

2019 First Edition



FUNDAMENTALS OF **SOFTWARE** ENGINEERING



ENGINEERING HANDBOOK

RAJAT GUPTA

Contents

CHAPTER . 1 . INTRODUCTION TO SOFTWARE ENGINEERING

1.1	INTRODUCTION	1
1.2	SOFTWARE CLASSES	2
1.3	TYPES OF SOFTWARE	2
1.3.1	System Software	2
1.3.2	Application Software	2
1.3.3	Utility Software	2
1.4	ROLE OF SOFTWARE	3
1.5	WHAT IS A GOOD SOFTWARE ?	4
1.6	PROGRAM VS. SOFTWARE	4
1.7	SOFTWARE CAN BE VIEWED AS A PRODUCT. HOW ?	5
1.8	LIMITATIONS OF SOFTWARE	6
1.9	SOFTWARE CRISIS	6
1.10	SOFTWARE MYTHS	7
1.10.1	Management myths	7
1.10.2	Customer myths	7
1.10.3	Practitioner's myths	7
1.11	WHY SOFTWARE NEEDS TO BE TREATED IN AN ENGINEERED WAY ?	8
1.12	WHAT IS SOFTWARE ENGINEERING ?	8
1.13	SOFTWARE ENGINEERING ACTIVITIES, SKILLS AND CHALLENGES	9
1.14	SOFTWARE ENGINEERING COMPONENTS	10
1.14.1	Systems Engineering Approach	10
1.14.2	Development Engineering Approach / Methodology	10
1.14.2.1	SSAD and OOSAD	11
1.15	WHAT IS A SOFTWARE PROCESS ?	13
1.16	SOFTWARE DEVELOPMENT PROCESS MODELS	13
1.17	SOFTWARE DEVELOPMENT LIFE CYCLE : (SDLC)	14
1.18	MODERN SOFTWARE DEVELOPMENT	14
	SUMMARY	15
	EXERCISES	16

CHAPTER . 2 . SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

2.1	DEFINITION	17
2.2	OBJECTIVES OF SDLC : SDLC	17
2.3	PHASES OF SDLC	17
2.4	DIAGRAMMATIC REPRESENTATION OF SOFTWARE LIFE CYCLE	18
2.4.1	User / Stakeholder's requirements	18
2.4.2	Feasibility study	19
2.4.3	Requirement analysis and specification	19
2.4.4	Design	19
2.4.5	Coding	20
2.4.6	Testing	20
2.4.7	Certification	20

2.4.8	Implementation	20
2.4.9	Maintenance and review	20
2.4.10	Special phases	20
	SUMMARY	22
	EXERCISE	22

CHAPTER . 3 . SOFTWARE PROCESS MODEL

3.1	INTRODUCTION	23
3.2	CATEGORIES OF SOFTWARE PROCESS MODELS	24
3.3	THE WATERFALL MODEL	24
3.3.1	System engineering	25
3.3.2	Requirement analysis	25
3.3.3	Design phase	26
3.3.4	Coding	27
3.3.5	Testing	27
3.3.6	Maintenance	27
3.3.6	Advantages	28
3.3.7	Disadvantages	28
3.4	PROTOTYPING MODEL	28
3.4.1	Reasons for using prototyping model	29
3.4.2	Type of prototyping	29
3.4.3	Evolutionary prototyping	29
3.4.4	Throwaway prototyping	30
3.4.5	Rapid prototyping techniques	31
3.5	THE RAPID APPLICATION DEVELOPMENT (RAD) MODEL	32
3.5.1	Process of the RAD	33
3.5.2	Advantages of RAD model	34
3.6	THE NEED FOR EVOLUTIONARY MODELS	34
3.7	THE INCREMENTAL MODEL	35
3.7.1	Advantages	35
3.7.2	Disadvantages	36
3.8	SPIRAL MODEL	36
3.9	COMPONENT ASSEMBLY MODEL	37
3.9.1	Advantages	38
3.9.2	Disadvantages	38
3.10	THE CONCURRENT DEVELOPMENT MODEL	39
3.10.1	Advantages	39
3.11	THE FORMAL METHODS MODEL	40
3.11.1	The merits of this model are given below	40
3.11.2	The demerits of this model are listed below	40
3.12	FORTH GENERATION TECHNIQUES (4GT)	40
3.13	COMPARISON AND SUITABILITY OF SOFTWARE LIFECYCLE MODELS	41
3.14	SELECTION OF A LIFECYCLE MODEL	43
3.14.1	Characteristics of requirements	43
3.14.2	Status of development team	44
3.14.3	Involvement of users	44
3.14.4	Type of project and associated risk	45
	SUMMARY	45
	EXERCISE	46

CHAPTER . 4 . FEASIBILITY STUDY

4.1	INTRODUCTION	48
4.2	SOFTWARE PROJECT MANAGEMENT	48
4.3	ROLE OF PROJECT MANAGER	48
4.4	ROLE OF SYSTEM ANALYST	49
4.5	PROJECT MANAGEMENT DIFFICULTIES	49
4.6	SOFTWARE PROEJCT PLANNING	50
4.7	SOFTWARE PROJECT MANAGEMENT PLAN (SPMP)	50
4.8	SOFTWARE PROJECT SCHEDULING	53
4.8.1	Project scheduling activities	53
4.8.2	Software project scheduling techniques	54
4.8.2.1	Work Breakdown structure (WBS)	54
4.8.2.2	Activity chart / Network	55
4.8.2.3	CPM and PERT	59
4.8.2.4	GANTT Charts	76
4.9	SOFTWARE PROJECT ESTIMATION	79
4.9.1	Software metrics in project estimation	80
4.9.2	Types of software metrics	81
4.9.3	Qualities of software metrics	81
4.9.4	Product metrics	82
4.9.4.1	Lines of codes (LOCs)	82
4.9.4.2	Function Points (FPs)	83
4.9.4.3	Feature Point metrics	87
4.9.5	Software project estimation techniques	87
4.9.6	Cyclomatic complexity	98
4.9.6.1	Program Control Flow Graph (CFG)	99
4.9.6.2	Advantages of cyclomatic complexity	100
4.9.6.3	Disadvantages	100
4.10	ESTIMATION ON STAFFING	104
4.10.1	Rayleigh's model	104
4.11	TEAM STRUCTURE	108
4.12	SOFTWARE RISK MANAGEMENT	109
4.12.1	Risk management activities	110
4.12.2	Risk control	112
	SUMMARY	113
	EXERCISE	113

CHAPTER . 5 . REQUIREMENT ENGINEERING

5.1	INTRODUCTION	116
5.2	PROBLEM ANALYSIS AND PRODUCT DESCRIPTION	117
5.3	REQUIREMENTS ENGINEERING (RE)	117
5.3.1	Requirements elicitation	118
5.3.2	Requirements analysis	120
5.3.3	Requirements specification	120
5.3.4	Modeling the system	120
5.3.5	Requirements validation	121
5.3.6	Requirements management	121

5.4	CONDUCTING A REQUIREMENTS STUDY	121
5.5	FACILITATED APPLICATION SPECIFICATION TECHNIQUES (FAST)	122
5.6	IMPACT OF PROTOTYPING ON REQUIREMENTS	123
5.7	USES OF THE SRS	124
5.8	WHAT OUGHT TO BE INCLUDED IN THE SRS ?	125
5.8.1	Behavioral Requirements	125
5.8.2	Non-behavioral Requirements	125
5.9	EXCLUSION OF PROJECT REQUIREMENTS FROM SRS	125
5.10	EXCLUSION OF DESIGN FROM SRS	125
5.11	EXCLUSION OF PRODUCT ASSURANCE PLANS FROM SRS	125
5.12	ATTRIBUTES OF HIGH QUALITY SRS	126
5.13	GENERAL FORMAT OF A SRS	128
5.14	STANDARDS IN SRS	128
5.15	AN APPROVED FORMAT FOR SOFTWARE REQUIREMENTS SPECIFICATIONS(S.R.S)	136
5.16	SRS : A LIVE EXAMPLE	141
	SUMMARY	155
	EXERCISES	156

CHAPTER . 6 . SOFTWARE DESIGN & CODING

6.1	INTRODUCTION TO SOFTWARE DESIGN	157
6.2	DEFINITIONS	157
6.3	DESIGN PROCESS	158
6.3.1	Interface Design	159
6.3.2	Architectural Design	159
6.3.2.1	Assessment of Architectural Design	160
6.3.3	Detailed Design	161
6.4	DESIGN CHARACTERISTICS	161
6.5	CRITERIA FOR QUALITY DESIGN	161
6.6	PRINCIPLES OF DESIGN	162
6.6.1	Modularity and Partitioning	162
6.6.2	Coupling	163
6.6.3	Cohesion	166
6.6.4	Span of Control	170
6.6.5	Module Size	170
6.6.6	Shared Use	171
6.7	IEEE RECOMMENDED DDS [DESIGN DOCUMENT SPECIFICATION] OR SDD [SOFTWARE DESIGN DOCUMENT]	171
6.7.1	Content description of SDD / DDS	172
6.7.2	Organisation of SDD	174
6.8	USER INTERFACE DESIGN	175
6.8.1	Graphical User Interface (GUI) vs. Character- based User Interface (CUI)	176
6.8.2	Classification of User Interface	178
6.8.3	Qualities of good User Interface Design (UID)	179
6.8.4	User Interface Design Principle	180
6.8.5	Elements for user interface Design	180
6.8.6	Graphical user interface	181

	6.8.6.1	Elements of GUI design	182
	6.8.6.2	Window Management System (WMS)	187
	6.8.6.2.1	X-Window system	189
6.9		SOFTWARE DESIGN METHODS	195
	6.9.1	Function-Oriented Design	196
	6.9.2	Data Structure Based Design	197
	6.9.2.1	Jackson Systems Development	197
	6.9.2.2	Warnier-Orr' System Design	199
	6.9.3	Object-Oriented Design Methods	201
	6.9.3.1	Benefits of OOD	202
	6.9.3.2	Types of OOD Methods	202
	6.9.4	Reuse-Based Design Methods	203
	6.9.5	Criteria for selecting a software Design Method	203
6.10		INTRODUCTION TO SOFTWARE CODING	204
	6.10.1	Coding Standards	204
	6.10.2	Coding Conventions	205
	6.10.3	Programming Style	207
	6.10.3.1	Importance of Programming Style	207
	6.10.3.2	General Program Style	207
	6.10.3.3	Good Programming Style	208
	6.10.3.4	Good Programming Style Aids	208
	6.10.4	System Verification	209
	6.10.4.1	Program Testing	209
	6.10.4.2	Reviews of Design and Code	210
	6.10.5	Code Inspections	210
	6.10.5.1	Code Inspection Process	211
	6.10.5.2	Checklist for Code Inspections	212
	6.10.5.3	Benefits of Code Inspections	213
	6.10.6	Code Reviews and Walkthroughs	213
	6.10.6.1	Rules for Code Reviews and Walk-throughs	214
	6.10.6.2	Benefits of Code Reviews and Walkthroughs	214
	6.10.6.3	Limitations of Code Reviews and Walkthroughs	214
	6.10.7	Coding Tools	215
	6.10.8	Documents Generated From Coding	215
		SUMMARY	215
		EXERCISE	216

CHAPTER . 7 . SOFTWARE TESTING

7.1	INTRODUCTION	219
7.2	TESTING AND SDLC : AN INTER-RELATIONSHIP	219
7.3	TESTING TERMINOLOGIES	220
7.4	DEFINITIONS OF SOFTWARE TESTING	221
7.5	PRINCIPLES OF TESTING	221
7.6	OBJECTIVES OF TESTING	222
7.7	LEVELS OF TESTING	222
	7.7.1 Unit testing	223
	7.7.2 Integration testing / Interface testing	225
	7.7.3 System testing	230
7.8	BLACK BOX (FUNCTIONAL) TESTING	231

7.9	WHITE BOX TESTING / STRUCTURAL TESTING	232
7.10	STATIC TESTING STRATEGIES :	235
7.11	FORMAL TECHNICAL REVIEWS	236
7.12	DEBUGGING	236
	7.12.1 Debugging process	237
7.13	SPECIAL SYSTEM TESTS	238
	SUMMARY	239
	EXERCISES	239

CHAPTER . 8 . SOFTWARE CERTIFICATION

8.1	INTRODUCTION	240
8.2	VERIFICATION AND VALIDATION	241
8.3	SOFTWARE QUALITY ASSURANCE	241
	8.3.1 SQA objectives	242
	8.3.2 SQA plan	242
8.4	SOFTWARE QUALITY	243
	8.4.1 Classification of software quality	243
	8.4.2 Software quality attributes	243
	8.4.3 McCall's quality factors	244
	8.4.3.1 Product operation quality factors	244
	8.4.3.2 Product revision factors	245
	8.4.3.3 Product transition quality factors	245
	8.4.4 Criteria for software quality	245
	8.4.5 Quality representation	245
	8.4.6 Importance of software quality	246
8.5	CAPABILITY MATURITY MODEL (SEI - CMM)	246
8.6	INTERNATIONAL STANDARD ORGANISATION (ISO)	249
	8.6.1 Need of ISO certification for software industry	249
	8.6.2 Steps for ISO 9000 certification	249
	8.6.3 Benefits of ISO-9000 certification	250
	8.6.4 Uses of ISO	250
	8.6.5 Comparison between ISO 9000 certification and SEI-CMM	251
	8.6.6 Classification of failures	252
	8.6.7 Limitation of ISO 9000 certification	252
8.7	RELIABILITY ISSUES 252	
	8.7.1 Software reliability specification	253
	8.7.2 Reliability terminologies	253
	8.7.3 Reliability metrics	253
	8.7.4 Measurement of Reliability and Availability	254
	8.7.5 Reliability growth modelling	256
8.8	PERSONAL SOFTWARE PROCESS	258
	8.8.1 PSP planning	258
8.9	SIX SIGMA	259
	8.9.1 Objectives	259
	SUMMARY	260
	EXERCISES	260

CHAPTER . 9 . SOFTWARE MAINTENANCE

9.1	INTRODUCTION	262
9.2	NEED FOR SOFTWARE MAINTENANCE	262
9.3	CATEGORIES OF MAINTENANCE	263
9.4	CHALLENGES IN MAINTENANCE	264
9.5	SOLUTION TO MAINTENANCE CHALLENGES	265
9.6	MAINTENANCE PROCESS	266
9.7	MAINTENANCE MODELS	267
9.7.1	Build and Fix model	267
9.7.2	Iterative enhancement model	268
9.7.3	Reuse - oriented model	269
9.7.4	Boehm's model	270
9.7.5	Taute maintenance model	270
9.8	MAINTENANCE COST ESTIMATION	272
9.9	CHARACTERISTICS OF SOFTWARE EVOLUTION	273
9.10	SOFTWARE CONFIGURATION MANAGEMENT	276
9.10.1	Version and Releases	277
9.10.2	Version and Release management	278
9.10.3	What is Milestone and Deliverable ?	278
9.10.4	Software Configuration Management activities	278
9.11	CHANGE CONTROL PROCESS	282
	SUMMARY	284
	EXERCISES	284

CHAPTER . 10 . SOFTWARE RE-ENGINEERING

10.1	INTRODUCTION	286
10.2	SOFTWARE RE-ENGINEERING PROCESS MODEL	287
10.2.1	Inventory analysis	288
10.2.2	Document restructuring	288
10.2.3	Reverse engineering	288
10.2.4	Code re-structuring	289
10.2.5	Data re-structuring	289
10.2.6	Forward engineering	289
10.3	ADVANTAGES OF SOFTWARE RE-ENGINEERING	289
10.4	REVERSE, FORWARD AND RE-ENGINEERING : A COMPARATIVE STUDY	290
10.5	IMPORTANCE OF REVERSE ENGINEERING	290
10.6	REVERSE ENGINEERING PROCESS	290
10.7	LEVELS OF REVERSE ENGINEERING	291
10.7.1	Redocumentation	292
10.7.2	Structural redocumentation	292
10.7.3	Design Recovery	292
10.8	REVERSE ENGINEERING TOOLS	292
	SUMMARY	293
	EXERCISE	293

CHAPTER . 11 . COMPUTER AIDED SOFTWARE ENGINEERING

11.1	INTRODUCTION	294
11.2	LEVELS OF CASE	295
11.3	ARCHITECTURE OF CASE ENVIRONMENT	295
11.3.1	User Interface / Interface Generator	296
11.3.2	Tools Management Services (Tools Set)	296
11.3.3	Object Management System (OMS)	296
11.3.4	Repository	296
11.4	BUILDING BLOCKS FOR CASE	297
11.5	CASE SUPPORT IN SOFTWARE LIFE CYCLE	297
11.6	OBJECTIVES OF CASE	299
11.7	CASE REPOSITORY	300
11.8	CHARACTERISTICS OF CASE TOOLS	302
11.9	CLASSIFICATION OF CASE TOOLS	302
11.10	CATEGORIES OF CASE TOOLS	303
11.11	ADVANTAGES OF CASE TOOLS	305
11.12	DISADVANTAGES OF CASE TOOLS	305
	SUMMARY	306
	EXERCISES	306

CHAPTER . 12 . UNIFIED MODELING LANGUAGE

12.1	INTRODUCTION	307
12.2	MODEL	308
12.3	THE UML	308
12.4	UML ARCHITECTURE	310
12.5	UML FOUNDATIONS	311
12.6	RULES OF THE UML	313
12.7	COMMON MECHANISMS IN UML	313
12.8	USE CASE DIAGRAM	314
12.9	CLASS DIAGRAM	316
12.9.1	Relationship in class Diagram	317
12.9.2	Extensibility mechanisms	322
12.9.3	Example of UML Class Diagram	324
12.9.4	Meta Model	324
12.10	INTERACTION DIAGRAMS	325
12.10.1	Sequence diagrams	325
12.10.2	Collaboration diagrams	327
12.11	STATE-CHART DIAGRAM	328
12.12	ACTIVITY DIAGRAM	330
12.13	OBJECT DIAGRAM	332
12.14	IMPLEMENTATION DIAGRAMS	332
12.14.1	Component Diagram	332
12.14.2	Deployment Diagram	333
12.15	PACKAGES AND MODEL MANAGEMENT	334
12.16	OBJECT CONSTRAINT LANGUAGE	336
12.17	MODELING PATTERNS & FRAMEWORKS IN UML	336

12.17.1	Patterns	336
12.17.2	Frameworks	338
12.18	UML COMPATIBILITY	339
	SUMMARY	340
	EXERCISE	340

CHAPTER . 13 . OBJECT ORIENTED SOFTWARE ENGINEERING

13.1	INTRODUCTION	341
13.2	OBJECT ORIENTED TERMINOLOGIES	342
13.3	OBJECT ORIENTED SDLC (SOFTWARE DEVELOPMENT LIFE CYCLE)	343
13.3.1	Objectives of Object Oriented SDLC	343
13.3.2	The Software Development Process	345
13.3.2.1	Object-Oriented Requirements Analysis (OORA)	346
13.3.2.2	Object-Oriented Analysis (OOA)	346
13.3.2.3	Object-Oriented Design (OOD)	347
13.3.2.4	Object-Oriented Programming (OOP)	347
13.4	MERITS OF OBJECT ORIENTED SOFTWARE	354
13.5	DEMERITS OF OBJECT ORIENTED SOFTWARE	354
13.6	DIFFERENCES BETWEEN OOA AND OOD	354
13.7	OOPS PROGRAMMING LANGUAGES	355
	SUMMARY	357
	EXERCISE	357

CHAPTER . 14 . SOFTWARE & TOOLS

14.1	INTRODUCTION	358
14.2	ANALYSIS TOOLS	359
14.3	DESIGN TOOLS	359
14.4	DEVELOPMENT TOOLS	359
14.5	TOOLS FOR SPECIAL PURPOSES	360
14.5.1	Tools for Documenting procedure and Decision making	360
14.5.2	Tools for Data Flow strategy or Data Flow Analysis	363
14.5.3	Tools for Proto-typing	396
	SUMMARY	398
	EXERCISE	398



Introduction to software Engineering

1.1 INTRODUCTION

Computers; Amazing machines !

We are living and breathing in the computer age and the computer has gradually become such a basic necessity of life that it is tough to imagine the life without it.

Computers are affecting every sphere of our life, in government, business, education, entertainment, defence, medical science, space, research, weather forecast, legal practice, even in our personal and day-to-day life.

☛ *To think of anything without computer is meaningless.*

A computer system can be viewed as a flexible electronic / mechanical device, responds inputs (data), processes and produces outputs (information).

Basically computer system is framed using the following elements.

- * Processing unit
- * Memory unit
- * Input unit
- * Output unit.
- * Program.

Now, you must be wondering, what is so special about this machine that people from diversified fields can use it so flexibly for entirely different functions ?

The answer is that the computer is programmable i.e. it all depends upon what program it is using for performing a particular function.

What is a program ? In very simple language we can say that a “program is a set of instructions tells the computer what to do.”

A computer system can be broadly disintegrated into two parts.

- * the hardware part
- * the software part.

➡ Software runs the Hardwares

Software is a general term, which is used to described a set of instruction (more precisely programs) written with the help of some predefined/planned format / procedures.

Software may be a program or a set of programs.

The importance of software can be viewed through an example, say human brain vs human body.

All parts of human body are activated / controlled by the human brain with predefined instructions (program) fed into it.

The attitude / activities and response of a person are truly based on the “mantras” (i.e. the software) given to the human brain.

➡ The change in Software influences Hardwares

1.2 SOFTWARE CLASSES

It is classified into two categories.

- Generic software
- Customised software

Generic software is designed for a wide range of market whose requirements are common, stable and well understood.

Example - Operating system software

Customised (Bespoke product) software is designed for a customer where domain, environment and requirements are unique to the customer only.

Example - Application software :

1.3 TYPES OF SOFTWARE

Software generally of three types :

- System software
- Application software
- Utility software

1.3.1 System Software

This consists of all the programs, languages and documentations supplied by the manufacturer of the computer system for its exclusive use.

Example - Operating system, BIOS programs, Platform oriented software, It also comes under system software

Example - Interpreter, Compiler.

1.3.2 Application Software

These programs are developed by professional groups for some specific application / functions. These are also called user-based software.

Example - Pay roll system, Banking software.

Embedded software.

1.3.3 Utility Software

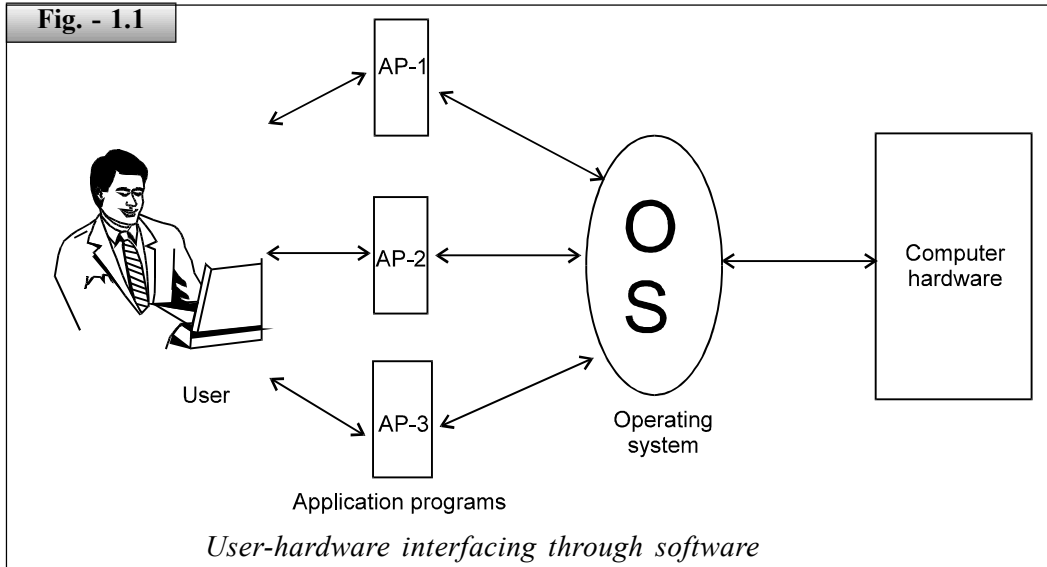
This may be considered as an application software / system support software which is very often used in the development of wide range of programs.

Example - MS-Office, Compilers, Interpreters, Debugger etc.

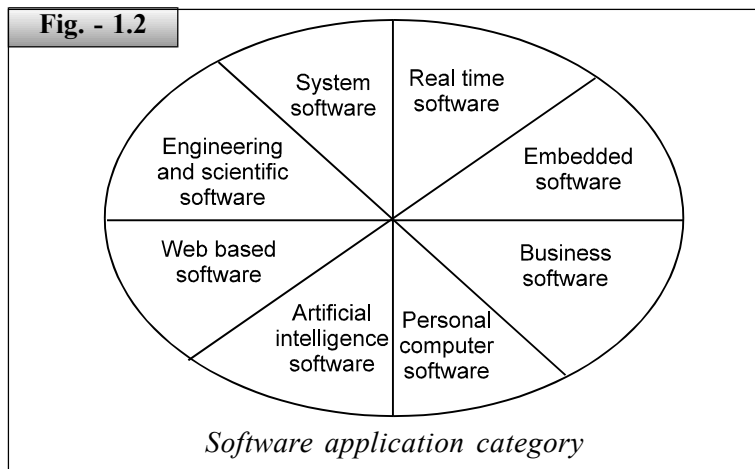
1.4 ROLE OF SOFTWARE

The key role of software is to perform tasks for the user by activating and controlling the computer hardware.

It can be shown in the following fig. 1.1



Software applications are categorised into five types for convenience. They are system software, business software, design and scientific software, embedded software and artificial intelligence software.



System software enables and provide services to software applications loaded on the computer system. It regulates the system performance and helps to run user-initiated applications.

Business software can be a generic product or customised product. Some are common to all industries and some deal with industry specific information processing requirements.

Design / scientific softwares deal with processing requirements in their specific field. These Softwares service the need of drawing, drafting, modeling, load calculation, building planning and designing using CAD/CAM, analysis of engineering data, statistical data for interpretation and decision making.

Embedded softwares are used to perform specific function under control conditions and further embedded into hardware as a part of large system. Ex. in Robotics

Artificial Intelligence software (AI) uses non-numeric algorithms, which use the data and information generated in the system to solve complex problems.

1.5 WHAT IS A GOOD SOFTWARE ?

A software can have number of attributes, which together will decide whether it is a good or bad one. The definition of a good software varies with respect to the person who evaluates it.

- The customer will decide on the basis of the cost-effectiveness of the software.
- The user group will consider its usability and reliability.
- The software engineer will look at its maintainability and efficiency.

The measure of good software is the customer satisfaction, cost and budget constraint fulfillment. Customer satisfaction depends on the degree to which customer requirements and expectations have been met.

The minimum essential attributes of a good software are maintainability, dependability, efficiency and usability.

Table- 1.1 : Generic comparison of software with hardware.

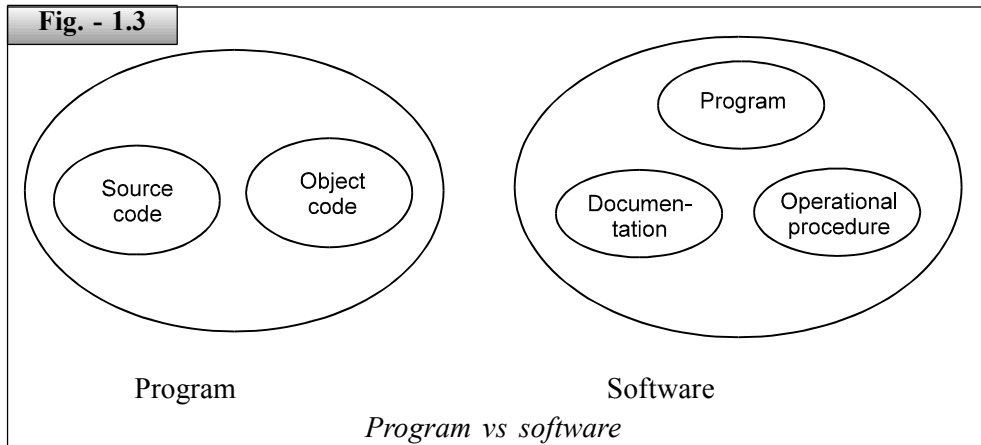
Generation	Hardwares	Softwares
1st Generation 1944 - 1955	Vaccum tubes	Machine language
2nd Generation-1956	Transistors	Symbolic language
3rd Generation 1964-1970	Integrated circuit	High level language FORTRAN, ALGOL. etc.
4th Generation 1971-1990	Large scale Integrated circuits.	PLI Basic PASCALS etc.
5th Generation 1990 +	Very Large Scale Integration	C, C++, Visual Basic, Fox-pro, DBMS, Prolog, LISP etc.

➡ The rate of advancement in software is more as compare to hardware

1.6 PROGRAM VS. SOFTWARE

People say “program is a software and software is a program or set of programs”. So how to distinguish ?

Fig. - 1.3



Simple, program comes under the software category or program is a subset of software. A program can be viewed as

- a set of instructions written for a specific task by individuals.
- small in size and limited functionality.
- it does not hold all the properties of what actually it is intended for.

[size, portability, compatibility, entertaining wide range of inputs, user friendly etc. and lot more...]

➡ Program = source code + object code

A software is a broad sense of developing programs that satisfies the criteria like

- user friendly
- portable
- maintainable
- fits to wide range of environments
- error / risk free
- cost effective etc.

Software consists not only program codes but also of all associated documents (design, testing operating procedures which includes user manuals and operational manuals.)

➡ Software = program + documentation + operating procedures

1.7 SOFTWARE CAN BE VIEWED AS A PRODUCT. HOW ?

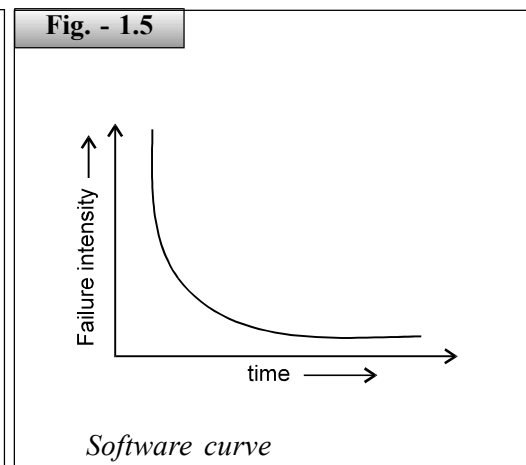
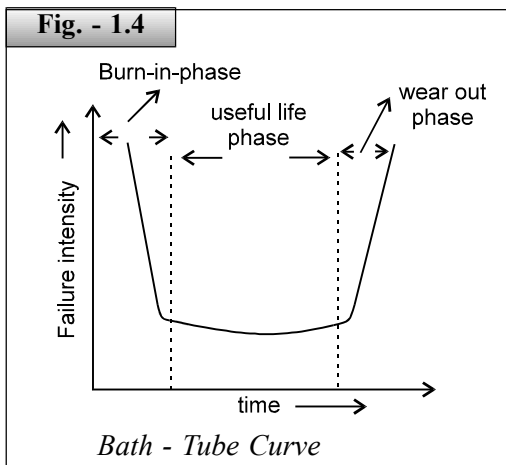
A product (consumable) which is available in the market is exhausted for the users. The demand of a product is based on its price, quality, durability. When the demand of the customers changes w.r.t their taste / use, manufacturers need to modify / redesign these existing products or to introduce new products time-to-time.

Due to the advancement and global use of computer systems in each and every field (as shown in the generic comparison), the software plays vital role for all the users (End user, Govt. organisations etc)

1.8 LIMITATIONS OF SOFTWARE

- Software is not scalable.
- It deteriorates, but never wears out.
- It's existence can be felt when it runs.
- Legacy software can not be easily migrated and maintained.
- Software is engineered not manufactured.
- Software is custom built.

Limitations of Software :



1.9 SOFTWARE CRISIS

The term “*Software crisis*” refers to a set of problems encountered in the development of software and to encapsulate all the ills of the software product.

It is also characterised by “an inability to develop software on time, budget and within the requirements.”

A list of problems and the reasons on software crisis is given below.

Problems :

- Schedule and cost estimates are inaccurate.
- Productivity is not in pace with the demand of customers.
- Quality of the software is poor.
- Communication between the user and developer is measurable.
- Low maintenance.

Reasons :

- There is delay in process or stage (analysis, design, coding and testing etc.) resulting out of schedule.

- No proper methods to estimate a software project.
- No adequate principles of communication between user and developer.

Software crisis counts the problem of :

- Software compatibility
- Portability
- Documentation
- Staffing and Co-ordination
- Maintenance
- Cost effectiveness
- User friendliness
- Availability of bugs
- Software product updation etc.
- Risk containment.

1.10 SOFTWARE MYTHS

These are something like traditional stories / beliefs concern with the use and development of softwares by the user / developers that affect the way. Myths may appear to be reasonable statements of facts but may not be sufficiently enough to be implemented. Some myths are :

1.10.1 Management myths

- We do have books full of standards and principles for building software.
☛ then, what's the use of a software manager ?
- It's late finishing a Software product, just add more programmer to catch up the project.
☛ Allas ! it's not building a house.
- Better to hand over the project to a third party and get relaxed.
☛ Dream rarely comes true.

1.10.2 Customer myths

- I got money, I can avail it.
☛ Does it require to peep inside the basket what it contains & what I need ?
- Hey, I purchased the product that will do for me for ever.
☛ Are you satisfied with a fixed recipe throughout a week ?
- Software is flexible, It can be modified and the change can be accommodated any time.
☛ It's not a magician pocket to get any thing out of it, rather it needs a framework, additional cost and time.

1.10.3 Practitioner's myths

- We write a program once, get it to work and relax.
☛ Do you feel the only responsibility of mother to give birth a child ?
- Developing a program results with a software.
☛ A house is not a house if it has four walls and a roof.

1.11 WHY SOFTWARE NEEDS TO BE TREATED IN AN ENGINEERED WAY ?

“Engineering” is a discipline, which is framed with the association of

- People
- Machines
- Resources
- Technology
- Methods.

for manufacturing / making of products as per the user’s need / demand of the market i.e.

- Timely produced
- User friendly
- Economic
- Reliable etc.

As software is a product, it needs to be treated from it’s inception to retirement stage in an engineered way satisfying all needs of the developers (before, during & after the development of software product) and the user’s using it.

1.12 WHAT IS SOFTWARE ENGINEERING ?

Software Engineering is a methodology that includes process, methods, tools and techniques for the manufacturing of a software product which is -

- Timely produced
- User’s friendly
- Reliable
- Cost-effective
- Portable
- Versatile
- Inter-operable
- Maintainable
- Reusable

Software Engineering Definitions :

There are number of definitions of Software Engineering traced by different research groups, development organisations, software developers etc.

Some of highlighted definitions are noted below.

Fritz Bauer (1969)

Software Engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

Dennis (1975)

Software Engineering is the application of principles, skills and art to the design and construction of programs and system of programs.

Boehm (1979)

Software Engineering is the practical of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them.

Fairley (1985)

Software Engineering is the technological and managerial discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within estimated cost.

IEEE (1991)

Software Engineering is the application of a systematic, disciplined and quantifiable approach to the development, operation and maintenance of software.

Morven Gentleman (1992)

Software Engineering is the use of methodologies, tools and techniques to resolve the practical problems that arise in the construction, deployment, support and evaluation of software.

Stephen Schach (1992)

Software Engineering is a discipline whose aim is the production of quality software, that is delivered on time, within budget and that satisfies its requirements.

Refael J. Barros (1997)

Software Engineering is the application of methods and scientific knowledge to create practical cost-effective solutions for the design, construction, operation and maintenance of software and associated products in the service of mankind.

Software Engineering is concerned with building of artifact.

Software Engineering is a scientific approach for conceptualisation, inception, design, development, testing, implementation, maintenance and reuse of software products through process, tools and technology.

1.13 SOFTWARE ENGINEERING ACTIVITIES, SKILLS AND CHALLENGES

Software Engineering executes a set of activities essential for good software development.

These are :-

- ☛ Requirement analysis and definition
- ☛ Software scope and determination of application boundaries.
- ☛ Software factorisation in components and configurations for development, testing and integration.
- ☛ Planning, scheduling, executing, monitoring and control of software development.
- ☛ Testing at all stages and phases for quality assurance as required by the user.
- ☛ Documentation of software for uses.
- ☛ Implementation through demonstration, installation and execution at the customer's end.
- ☛ Change management in *pre-and post implementation* phase.

The managerial activities are basically carried out by project manager and system analyst, what contribute to the efficiency and effectiveness of the software product to be developed.

Such as -

- Resource and effort estimation, and management.
- Risk assessment and management.
- Process management to maintain cost, time and budget as defined.
- Project management for achieving Software goals.

1.14 SOFTWARE ENGINEERING COMPONENTS

Any software product and its quality depend upon the system on which it is installed. A software engineer should first understand the system on which the software is to be run. The characteristics of a system have lot of bearing on software scope, design and quality.

The term “system” may be any business organisation and computer softwares used in the organisation.

Software Engineering approach has two components for understanding and developing a system. They are;

- ☛ System Engineering Approach
- ☛ Development Engineering Approach

1.14.1 Systems Engineering Approach

The overall activities like system study and analysis of a system is carried out using the approach / methodology called Systems Engineering Methodology (SEM)

The SEM Steps are :

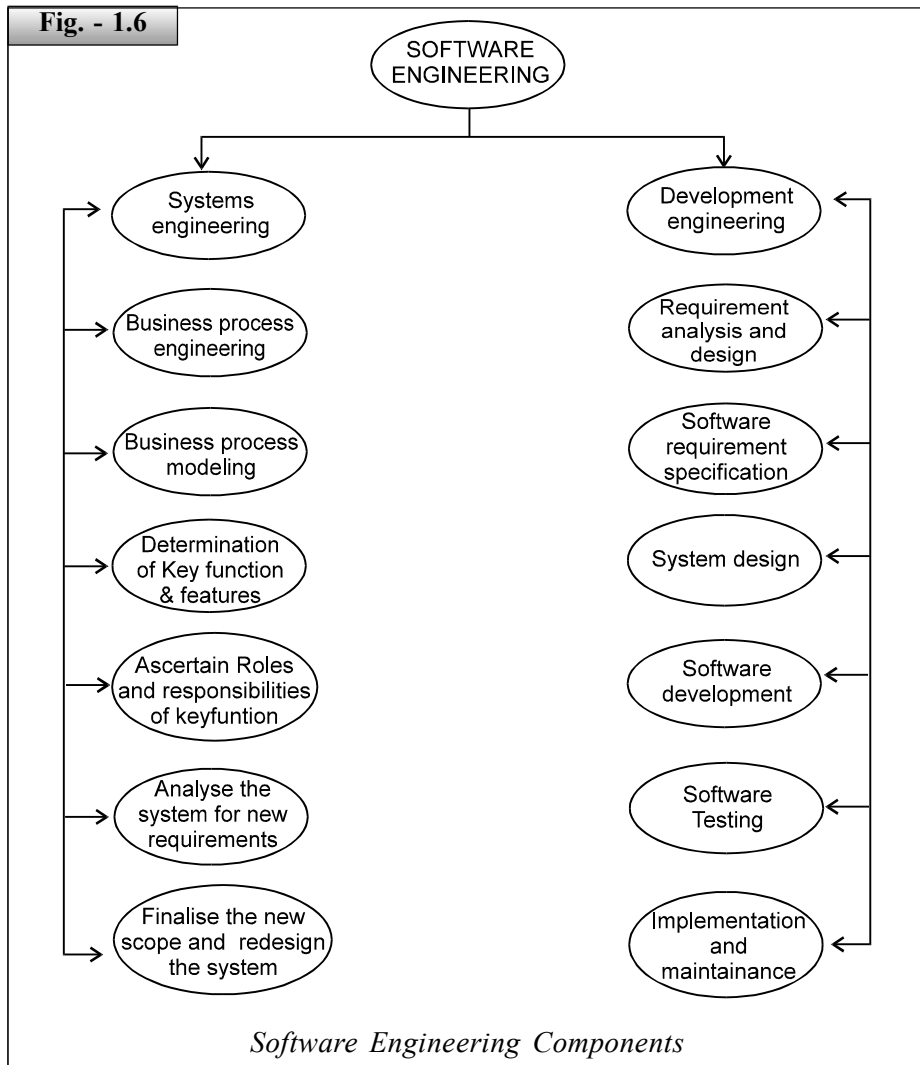
- Define objectives of the system.
- Define the application boundaries of the system.
- Factorisation of system into different components for understanding the system functions and features.
- Understanding the relationships between various components.
- Defining relationship in terms of inputs, outputs and processes.
- Understanding the role of hardware, software with the role of database and other software products used in the system.
- Identification of operational and functional requirements of the system.
- Use of modelling software for modelling the system.
- Interaction with customer, user and others affected by the system.

1.14.2 Development Engineering Approach / Methodology

It has a goal of translating the system requirements as software system goal, and proceeds to achieve through series of steps. The DEM has the following steps;

- Requirement definition and specifications.
- Design solution to deliver the requirements
- Determine the architecture for deliver of the solution.
- Software development planning.
- Software testing by components.
- Integration of system components.

- Integration testing for confirmation of delivery of requirements.
- Determination of implementation strategy
- Implementation.
- Change management process.
- Maintenance of the installed software.



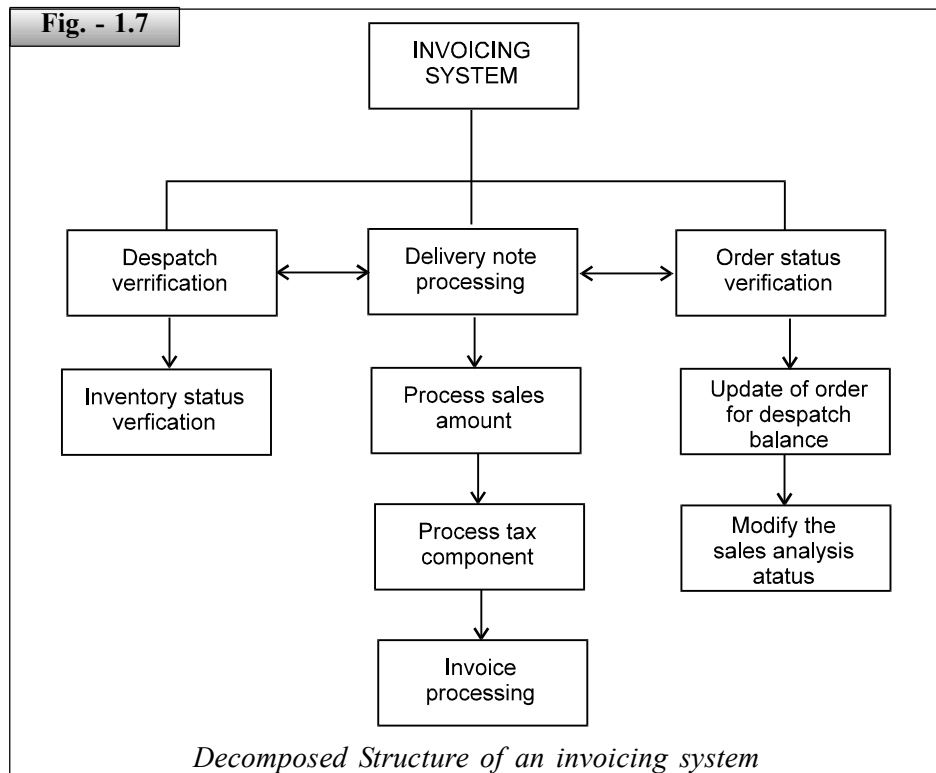
1.14.2.1. SSAD and OOSAD

Software development engineering is carried out in two ways.

- *Structured System Analysis and Design (SSAD)*
- *Object Oriented System Analysis and Design (OOSAD).*

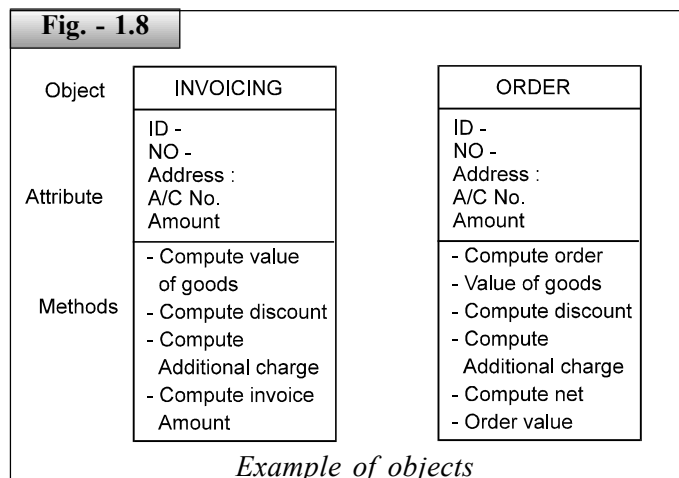
The SSAD approach, in which the system and its requirement are decomposed in structured manner into subsystems by functions.

Software development is carried out using the sub-systems structure tested, integrated and implemented. The software engineer's skill lies in decomposing the system in a structured fashion, that allows for understanding and developing a software with the required characteristics.



In contrast, the OOSAD development approach recommends an analysis of domain and the organisational business and builds objects of models independent of the system under consideration. The object represents a function, process or a document evolved for the organisation.

Each object has attributes to describe, methods to perform and relationship to other objects. The OOSAD principle is that when an object is developed, it is available for use in current, proposed and futuristic systems. It results with higher development efficiency and lower development cost.



In SSAD, the focus is on functions and the data structure designed for those functions. Functions, data and processing methods (software) are closely coupled.

In OOSAD, however, objects and processing methods (systems) are decoupled from data.

In SSAD, it is important to decompose the systems, where as in OOSAD, modelling the organisation and its business in objects.

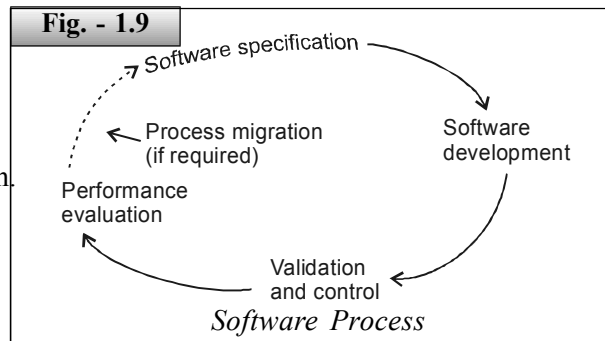
Both principles are similar in that the purpose of problem solving methodology and set of techniques and tools to assist software engineer to analyse, model, design and develop the system.

1.15 WHAT IS A SOFTWARE PROCESS ?

A process is a state of execution of particular task. It may also be a series of steps involving activities, constraints and resources that produce a specific output.

Software process is a set of activities and associated results, which produce a software product. The activities are basically carried out by the software engineers. There are four fundamental activities, which are common to all software processes.

- Software specification.
- Software development.
- Software validation and control.
- Software performance evaluation.



1.16 SOFTWARE DEVELOPMENT PROCESS MODELS

A process is intended to guide software developer team through a set of framework activities that are organised into a process flow, that may be :

- ☛ linear
- ☛ incremental
- ☛ Evolutionary (more in detail is described in chapter 3)

Process models provide stability, control and organisation to an activity. Software engineers and their managers adapt a prescriptive process model to their needs and then follow it. In addition, the people (customers) who have requested for the software products have to be a part of the development team.

A list of standard process models are :

- Iterative Waterfall / Linear Sequential Model
- Prototype Model
- RAD (Rapid Application Development) Model.
- Spiral Model etc.

1.17 SOFTWARE DEVELOPMENT LIFE CYCLE : (SDLC)

A systematic representation of different phases through which a software product undergoes from its inception to implementation and modification whenever required.

The highlighted phases / stages of the entire activity for software development are

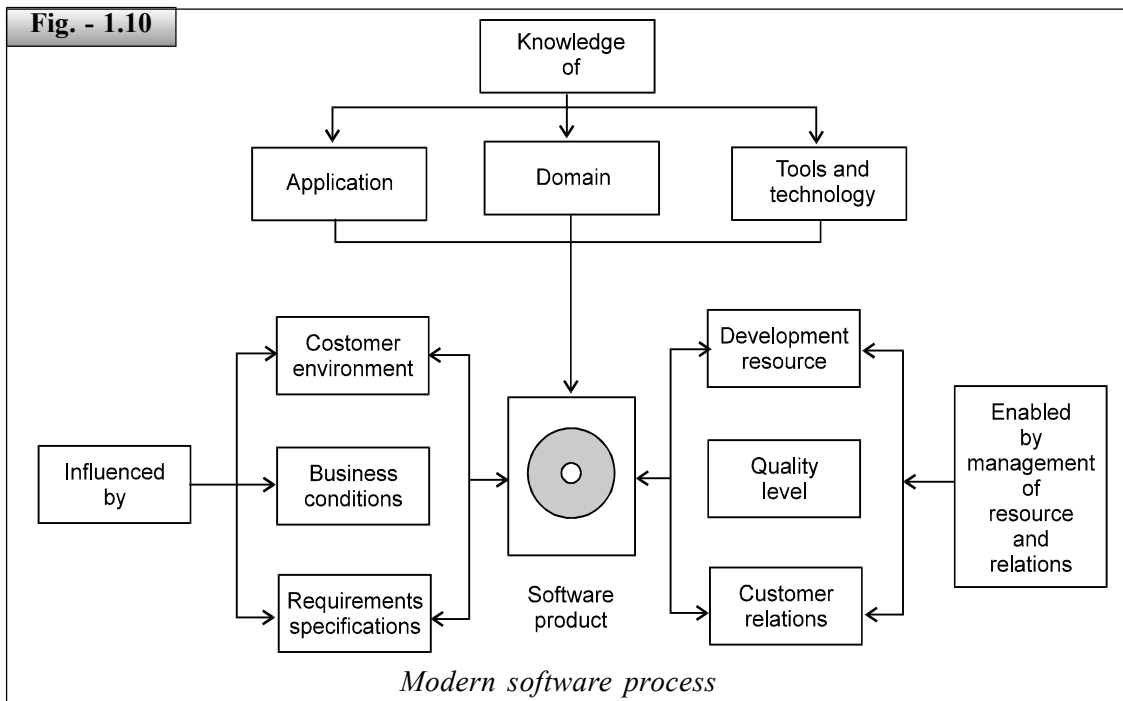
- User requirements
- Requirement and Analysis
- Coding
- Implementation
- Modification (whenever required) etc.
- Feasibility study
- Design
- Testing
- Maintenance and review

[The details are being discussed in chapter - 2.]

1.18 MODERN SOFTWARE DEVELOPMENT

Modern software development is complex for various reasons. It is technology driven and calls for knowledge on different fronts and management of complex business issues. Hence, software process management is a key management area in the development of effective software solutions.

The fig 1.10 shows the key area of management for increasing the effectiveness of process models.



Software process models will provide the best software solution if these key areas are managed properly. The organisation and its developers should have good knowledge of domain, application, tools and technology.

SUMMARY

- Use of software is increasing day-by-day in the field of industry, business education communication and many more, to improve the operational and management efficiency of conducting various activities. The importance of software can also be felt with comparison to hardware, what is run by the software itself.
- There are different classes of software based on their uses, such as- *generic* and *customized* software.
- The quality of software is accessed by the customer (user) with respect to its user friendliness, budget oriented, reliability, maintainability, portability and versatility etc.
- Software can also be viewed as a product like other but it needs to be treated or developed in an engineered way.
- Software cost now forms the major component of a computer system's cost. Software is currently extremely expensive to develop and is often unreliable. The goal of software engineering is to face this "software problem." In this chapter, we have discussed a few basic points regarding software and software engineering:
- Software is not just a set of computer programs but comprises programs and associated data and documentation. The main problems for software development currently are: high cost, low quality, and frequent changes causing rework.
- Software engineering is the discipline that aims to provide methods and procedures for developing software systems. The basic problem of software engineering is the problem of scale; the techniques used to solve small problems do not scale up to solve large and complex problems. And the main controlling factors are cost, schedule, quality, and consistency. The basic objective of software engineering is to develop methods for developing software that can scale up and be used to consistently develop high-quality software at low cost.
- The fundamental approach of software engineering to achieve its objective is to separate the development process from the products. Software engineering focuses on the process with the belief that the quality of products developed using a process are influenced mainly by the process. The process used for development need to be a phased process in order to achieve the software engineering objectives. As effective project management is critical to the success of a large development project, metrics-based project management is another basic approach software engineering uses.

We have considered a number of goals and problem areas in software development. Generally, software developers have a bad image, or reputation for producing software i.e.

- late
- over budget
- unreliable
- inflexible
- hard to use.

Because the problems are so great, there has been widespread talk of a crisis in software production. The general response to these problems has been the creation of a number of systematic approaches, novel techniques and notations to address the software development task. The different methods, tools and languages fit within a plan of action (called a process model).

EXERCISES

1. Define software.
2. What is software engineering.
3. What do you mean by the term “Software Engineering”? Describe the evolving role of software?
4. What are the different myths and realities about the software?
5. Give the various application areas of the software.
6. What is bathtub curve?
7. Discuss the characteristics of the software.
8. What characteristics of software make it different from other engineering products (for example hardware)?
9. Explain some characteristics of software. Also discuss some of the software components.
10. Comment on the statement “software does not wear out”.
11. Discuss about the evolution of software engineering as a subject in the last 50 years.
12. What are the different software components?
13. What are the symptoms of the present software crisis? What factors have contributed to the making of the present software crisis? What are possible solutions to the present software crisis?
14. What do you understand by software crisis?
15. What is software crisis? Give the problems of software crisis?
16. What do you mean by software myths?
17. Explain in detail software engineering process.
18. What is computer systems engineering? How is it different from software engineering?



Software Development Life cycle (SDLC)

2.1 DEFINITION

It's a strategy consisting of a set of well defined cyclic phases for a software product from its inception to implementation and its further modification (whenever required) as per the user's need.

When it is considered for any system of real world or computer system (*hardware*), it may also be called as *System Development Life Cycle*.

2.2 OBJECTIVES OF SDLC : SDLC

- helps understanding the whole process of designing / development of Software products.
- establishes a structured approach towards the development.
- enables resource planning for the developers in advance.
- controls, manages all the activities those are carried out during the entire development process of a Software product.

2.3 PHASES OF SDLC

It consists of 5 - 9 different phases (A phase can be identified as a definite stage with an entry (input) and exit (output) criteria through which the entire activities for the development of a software product are induced.

All the subsequent phases are associated with each other w.r.t their dependencies. (The exit (output) criteria of a particular phase can be treated as entry (input) for the next phase and so on...)

The different phases of SDLC are

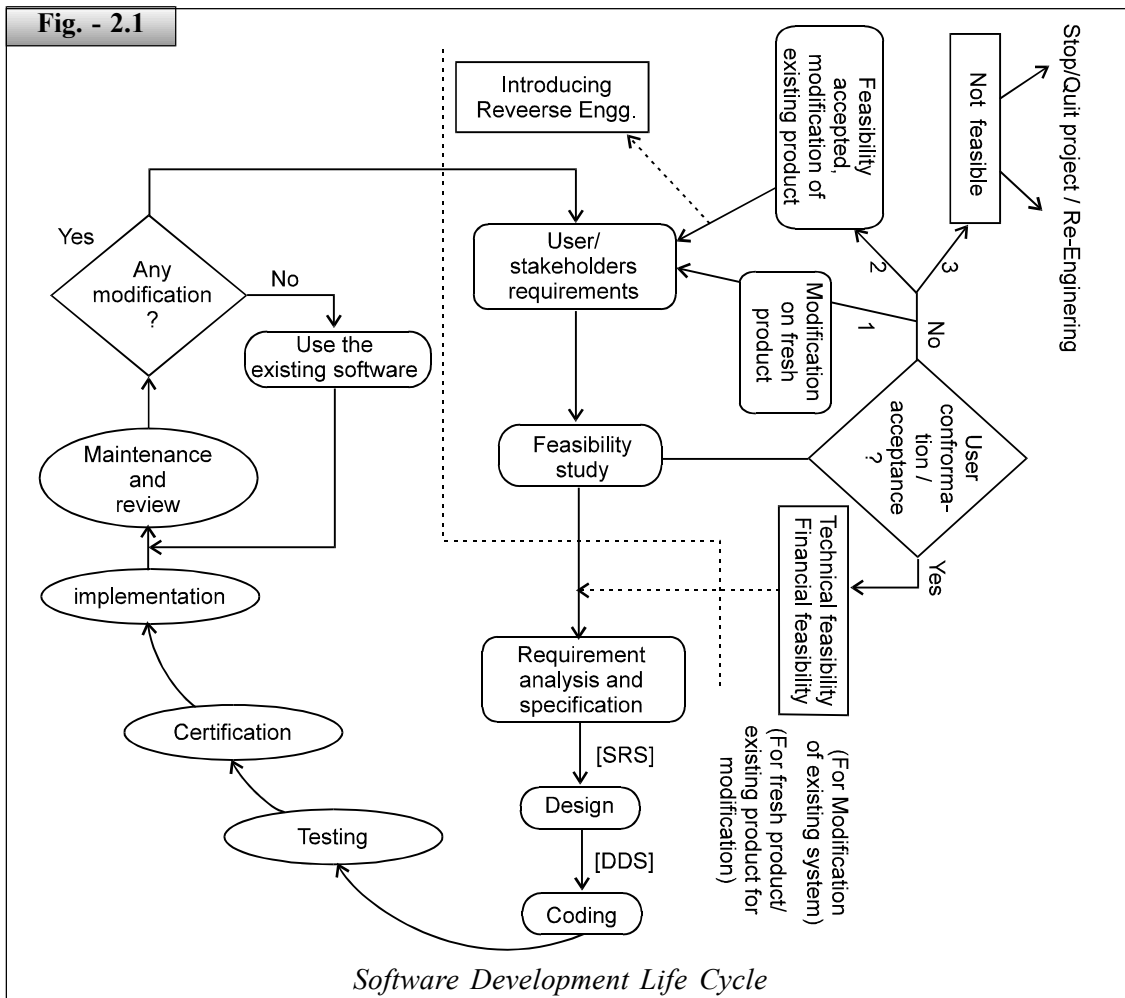
- User / Stake holder's requirements.
- Feasibility study
- Requirement Analysis and specification
- Design
- Coding

- Testing
- Certification
- Implementation
- Maintenance and Review.

Some other special phases (more to be called techniques) like, *Reverse engineering*, *Re-engineering* are introduced whenever desired.

2.4 DIAGRAMMATIC REPRESENTATION OF SOFTWARE LIFE CYCLE

The mapping of different activities (phases) performed on a Software product from its inception to retirement is shown in Fig. 2.1:



2.4.1 User / Stakeholder's requirements

Requirements are intended to meet the needs / objectives of an user / stakeholder (where user may be an individual, an organisation, software developers, system manufacturers etc.).

For Example : A small software product for an individual, like computer game, PDA (Personal Digital Assistance software)

- *An application product for an organisation e.g. ATM Banking, Railways, Inventory etc.*
- *Application /System software for the software developer group, system manufacturers, like MS-Office, Operating System software, interpreters, compilers etc.*

2.4.2 Feasibility study

It is a preliminary study which investigates the information needs of prospective users and determines the resource requirements, costs, benefits and technical /infrastructural requirements of a proposed project / an existing product under modification.

types of feasibility :

- Technical
- Financial / Economical
- Operational
- Behavioural

➤ **Technical feasibility** : It confirms whether the hardware and software are capable of satisfying the needs of a proposed system or an existing system (under changes) along with their higher degree of reliability and availability.

➤ **Financial / Economical feasibility** : It signifies the profitability of a proposed or implemented product / project w.r.t. cost saving, increased revenue, decreased investment and increased profit criterion.

➤ **Operational feasibility** : It identifies the willingness and ability of users or operators of a proposed / an existing product / project / system to operate and support i.e. end user acceptance, management support etc.

➤ **Behavioural feasibility** : Basically it is associated with real-time and embedded software development such as software for space-research, defence and scientific applications. It confirms about the friendliness and safetibility of the software product to be implemented in a particular / changeable environment.

2.4.3 Requirement analysis and specification

This phase starts after the successful completion of feasibility study and results with a well organised document (showing all logical requirements for the software product) called SRS (*Software Requirement Specification*)

Requirement Engineering (RE) is introduced for the determination of exact requirements of the user and to organise them into a specification document. The system analysts or engineers are responsible for the entire activities.

2.4.4 Design : This phase is fed with the SRS (from the previous phase) and results with another logical form of a document called DDS (Design Document specification).

There are different Software Engineering tools (*Data Flow Diagram, Decision Table, Data Dictionary, E-R diagram etc*) used at the time of designing document what identifies the activities like modularisation (considering coupling and cohesion criterion), determination of data, process, outputs of the modules and their relationship.

Different design strategies are also used in this phase (*Top-down, Bottom up etc.*)

2.4.5 Coding : It translates the *DDS* (from the design phase) into codes under a suitable chosen language environment. It emphasizes the improvisation of programming efficiency by reducing the elapsed (execution) time, identification/rectification of errors, and increasing the throughput (performance) and resource utilisation. It is maintained through LOCs (*Line of codes*), *FP* (*Function Point*), *Feature Point (FP)* etc.

2.4.6 Testing : This phase is associated with the activities like quality control measure, detection of errors on designed modules. During this phase specific test cases are executed and the actual results from the module under testing are compared with the expected outputs.

The final output of the testing phase is a test report and an error report. It does not show the absence of defects but the presence of software error.

2.4.7 Certification : The performance of the tested Software product is basically compared with the standards as framed by some internally recognised organisations like SEI, CMM and ISO etc.

SEI - Software Engineering Institute

CMM - Capability Maturity Model

ISO - International Organisation for Standardisation. It signifies the overall reliability of the software product as well as the organisational input in the user, using the product.

2.4.8 Implementation : It is mainly concerned with ascertaining site selection and preparation, file conversion and the tasks leading immediately to a fully operational system.

It also includes the final testing of the complete software product to the user satisfaction, producing security to the system.

2.4.9 Maintenance and review : It is an important phase of SDLC, it includes the correction of errors and the changes needed on the Software product.

It may be classified as -

- i) Corrective
- ii) Adaptive
- iii) Perfective
- iv) Preventive maintenance

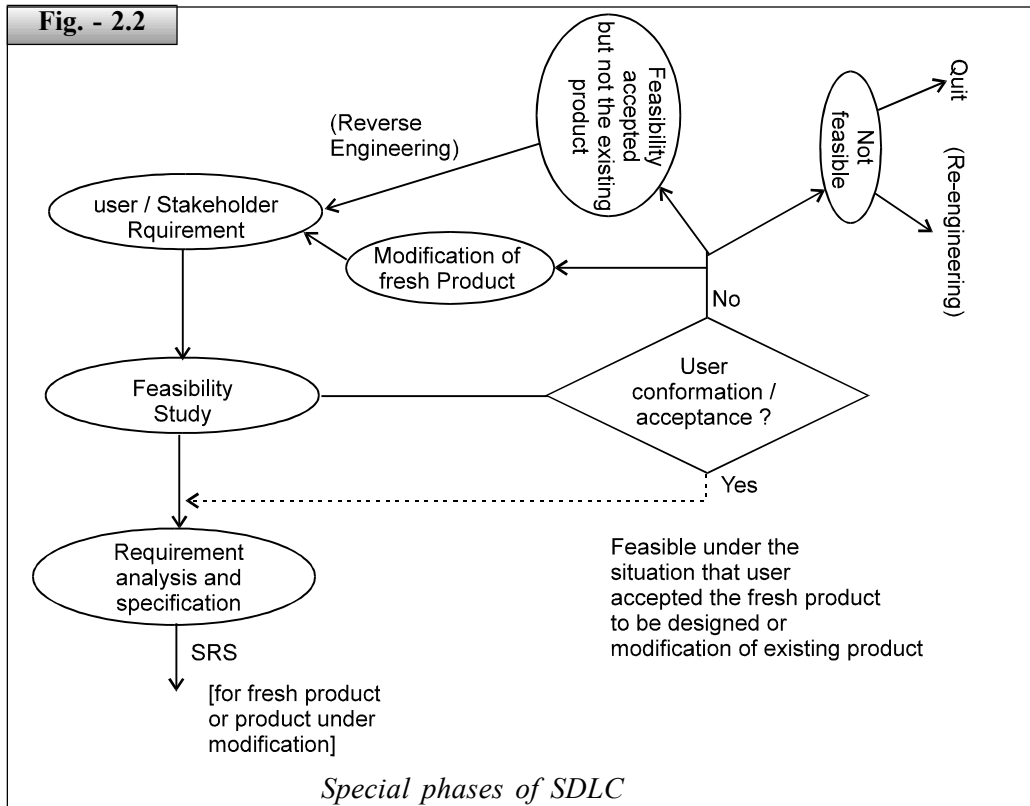
Review is a set of activities which is conducted by the software analyst / system analyst on the basis of following attributes.

- Case of use
- Level of Utilisation
- Response time
- Suitability of information
- Overall reliability

2.4.10 Special phases : [Techniques]

To have an overview on this type of phases, Let's consider the dotted portion of the SDLC as given in Fig. 2.1 :

Fig. - 2.2

**Possible Situations :**

- During / After feasibility study, the financial / technical or both report is submitted to the user / stakeholder.
 - If it (Feasibility report) is accepted by the user (Yes)
 - The fresh product will be entertained and *SRS* of the product is developed.
 - The product what needs (user) to be modified will be taken into consideration and *SRS* is developed.
 - If not accepted by the user (No), there is a chance of
 - Feasibility accepted / confirmed but customer needs some changes on the fresh product, that comes under the user / stakeholder's requirement category.
 - or
 - Feasibility accepted, user needs the modification of the existing product, then apply Reverse Engineering, that minimizes the cost of development, resources, manpower etc.
- (Reverse Engineering is discussed in detail in Chapter-10)
- or
- If the feasibility report is not accepted, then user has to think of the project to be abandoned or apply Re-engineering for the existing product what enables the better versions of an existing product without changing its core functionality.

SUMMARY

- There are nine distinct phases in the development of an information system. These phases constitute what is known as the system life cycle.
- A summary of what is done in each phase and the outputs obtained at the end of each phase is given in Figure 2.1.
- It should be remembered that in a design one may have to go back to an earlier phase in the design based on results obtained in a later phase. The phases are primarily intended as milestones to assess progress in design.

EXERCISE

1. What is the need of SDLC in software development process ?
2. Discuss SDLC in brief.
3. Give the basic phases in software development life cycle.
4. What are the different steps in software development life cycle? What are the end products at each step?
5. What are the important activities that are carried out during the feasibility study phase?
6. Explain the different categories of maintenance in the software development life cycle.
7. What is the role of testing phase "in software development life cycle?



Software Process Model

3.1 INTRODUCTION

In the early days of computing, software development was mainly an individual effort. There was no distinction between the programmer and the end-user of the application.

The end-user developed the application as a support to his / her own activity.

This kind of software development consisted only of coding in some language. It denotes only the development process. For small programs these activities may not be done accurately. But, for large systems, where the program development process includes a number of developers and time.

There is no need to break down the problem (Program) and documenting the various aspects of problem solving.

For any software system, of a nontrivial nature, each of the software development phase has to be exercised very carefully.

For large systems, each activity can be extremely complex and some formal mechanisms are needed to perform it efficiently and correctly.

Each of these activities is a major task for large software projects. So these activities can not be tackled in a single step and must be broken down into smaller steps.

Particularly, the problem for recognising the methods of software production process leads to the concept of structured models for describing it in a precise way with a view to make the process predictable and controllable.

Process models are the abstractions that assist the representation of the software process.

These are constructed with the builder's idea of what is needed in the final product. By defining the process models, it is beneficial to make the process more standardise.

The software process model enables the software developers to produce high quality, reliable and maintainable software in a systematic manner.

The process models follow up the *software life cycle* may completely or partially.

The nature of the developed software product may vary from product to product as the software process models are having different phases of activities.

Therefore, it is concluded that as the nature of the products vary, different software process models are required for software development.

3.2 CATEGORIES OF SOFTWARE PROCESS MODELS

Software process models can be categorised in to the following.

- 1) Linear sequential model
- 2) Iterative model
- 3) Evolutionary model
- 4) Formal methods model
- 5) System assembly from reusable components.

Lets examine each of these briefly :

1) **Linear sequential model :**

This model proceeds in a linear orderly fashion with transitions of well-defined deliverable at each stage.

Waterfall model and *RAD model* are referred as of linear sequential type.

2) **Iterative model :**

In this model, the process proceeds in the form of iterations. For each iteration, using the prototype, feedback about the model is obtained from the customer. This continues till the customer is satisfied with the model developed.

Prototype model is coming under the criteria of iterative model.

3) **Evolutionary model :**

This model is used when general objectives are known and detailed input / output are unknown.

Initially a core product is developed and the customer uses it.

As new requirement emerges, additional features are added to the existing system.

4) **Formal methods model :**

In this model, a formal mathematical system specification is developed and it is transformed in to a program by using various mathematical methods.

5) **Assembling a system using reusable components :**

It is applicable in an already existing system. In this model, the emphasis is given on the integration of reusable components rather than developing them from the scratch.

Out of all these above discussed process models, the *Linear sequential model*, *Iterative model* and the *Evolutionary model* are widely used for practical system development.

Hence we will be discussing these three approaches.

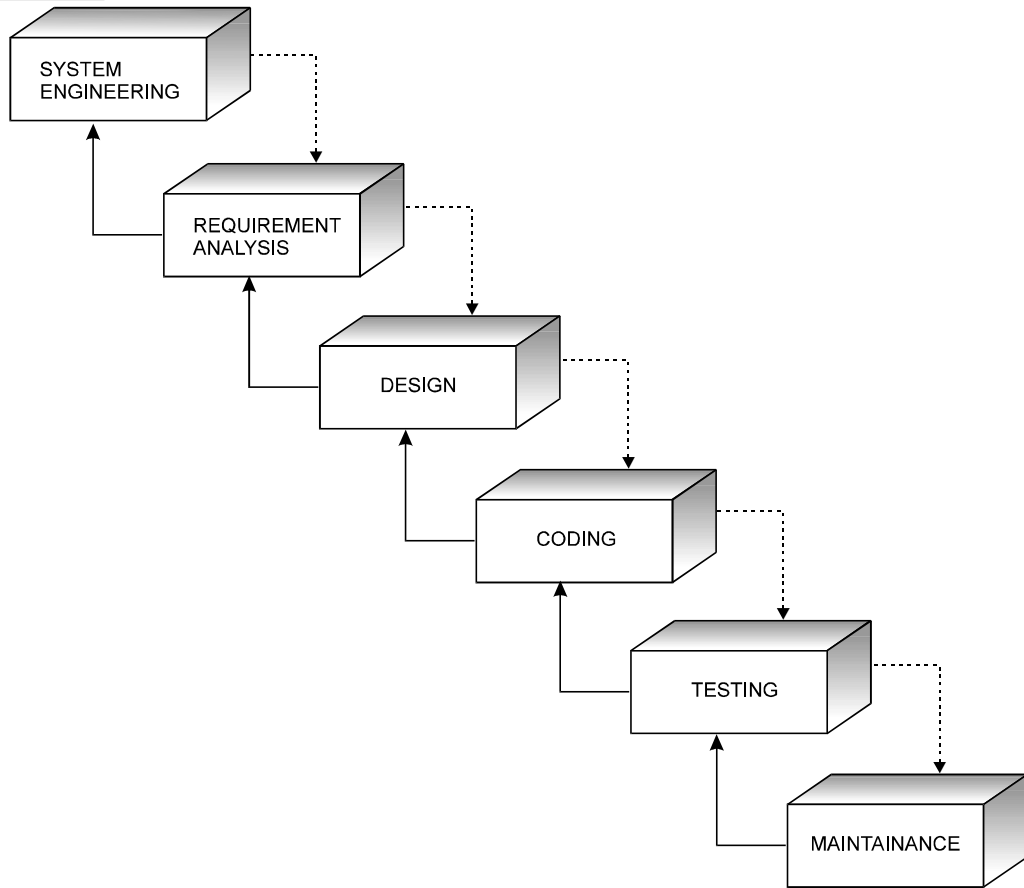
3.3 THE WATERFALL MODEL

The Waterfall model was proposed by Winston Royce in 1970. In the original model, the phases were iterative. In practice however, it becomes rigidly sequential, therefore, came to be known as the Linear sequential model.

The following figure depicts the *waterfall model* with iterative phases.

The principal stages of the *waterfall model* are :

Fig. - 3.1

*Iterative waterfall model*

3.3.1 System engineering

The software product is a part of large system. Therefore requirements are determined for all the system components and a part of these requirements are allocated for the software. This system view is needed when the software must interface with other elements like Hardware, people and databases.

3.3.2 Requirement analysis

Requirements are analysed and made out before proceeding to the other processes. Logical representation (*Graphic representation*) of the requirements analysis is required to avoid ambiguity in the requirements.

Extensive simulation and prototyping are some times used to capture and analyze the system requirements concerned with human interaction.

This phase exactly tells the requirement and needs of the project.

This is very important and critical phase in *waterfall model*. This purpose of a requirements analysis is to identify the qualities required of the application, in terms of functionality, performance, case of use, portability and so on.

- The requirements describe the “what” of a system, not the “how” ?
- This phase produces a large document, contains a description of what the system will do without describing how it will be done. The resultant document is known as *software requirement specification* (SRS) document.
- An SRS document must contain following :
 - * Detailed statements of problem.
 - * Possible alternate solutions to problem.
 - * Functional requirements of the software system.
 - * Constraints on the software system.
- The SRS document must be precise, consistent and complete.
- There is no scope of any ambiguity or contradiction in the SRS document.
- A SRS document may be organised as problem statement, introduction to problem, functional requirements of the system, nonfunctional requirements of the system, behavioural description and validation criteria.

3.3.3 Design phase

- The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.
- In technical terms, during the design phase the software architecture is derived from the SRS document.
- Two differently design approaches are available : i.e.
the traditional design approach and the object-oriented design approach.
- (i) **Traditional design approach** : The traditional design approach is currently being used by many Software development houses.
 - Traditional design approach consists of two different activities ; first a structured analysis of the requirement specification is carried out where the detailed structure of the problem is examined.
 - This is followed by a structured design activity.
 - During *structured design*, the results of structured analysis are transformed into software design.
 - Structured design is undertaken once the structured analysis activity is complete.
 - Structured design consists of two main activities : architectural design (also called *high level design*) and detailed design (also called *low - level design*).
 - High level design involves decomposing the system into modules, and representing the interfaces and the invocation relationships among the modules.
 - During detailed design, internal of the individual modules are designed in more detail, e.g. *the data structures and algorithms* of the modules are designed and documented.
- (ii) **Object-Oriented Design approach** : Various objects in the system are identified. After the identification of objects, the relationships among them are also explored. The OOD approach has several benefits such as lower development time and effort, and better maintainability.

3.3.4 Coding

- Coding is the phase in which we actually write programs under a suitable a programming language environment.
It was the only recognised development phase in early development processes, but it is one of several phases in a *waterfall process*.
- The output of this phase is an implemented and tested collection of modules.
- Coding can be subjected to company wide standards, which may define the entire layout of programs, such as the headers for comments in every unit, naming convention for variables and sub-programs, the maximum number of lines in each component and other aspects that the company deems worthy of standardisation.

3.3.5 Testing

- During the *testing* phase, the modules are integrated in a planned manner.
- The different modules making up a Software product are almost never integrated in one shot.
- *Testing* is carried out by a number of steps, during, each step the system is tested and a set of previously planned modules are added to it.
- When all the modules have been successfully integrated and tested, *system testing* is carried out.
- The objective of *system testing* is to determine whether the software system performs as per the requirements mentioned in SRS document. This testing is known as *system testing*.
- The *system testing* is done in three phases called “*Alpha*”, “*Beta*” and “*Acceptance testing*”.
- ☛ *Alpha testing* is conducted by the software development team at the developer’s site.
- ☛ *Beta testing* is performed by a group of friendly customers in the presence of the software development team.
- ☛ *Acceptance testing* is performed by the customer themselves. If the software is successful in *acceptance testing*, the product is installed at the customer’s site.

3.3.6 Maintenance

- *Maintenance* is defined as the set of activities that are performed after the system is delivered to the customer.
- *Maintenance* consists of correcting any remaining error in the systems, (*corrective maintenance*), adapting the applications to changes in the environment (*adaptive maintenance*), and improving, changing or adding features and qualities to the application (*perfective maintenance*).
- The cost of maintenance is often more than 60% of the total cost of software, and about 20% of maintenance costs may be attributed to each of *corrective and adaptive maintenance*, while over 50% is attributable to *perfective maintenance*.
- Based on this breakdown, we observed that evaluation is probably a better term than maintenance, although the latter is used more widely.

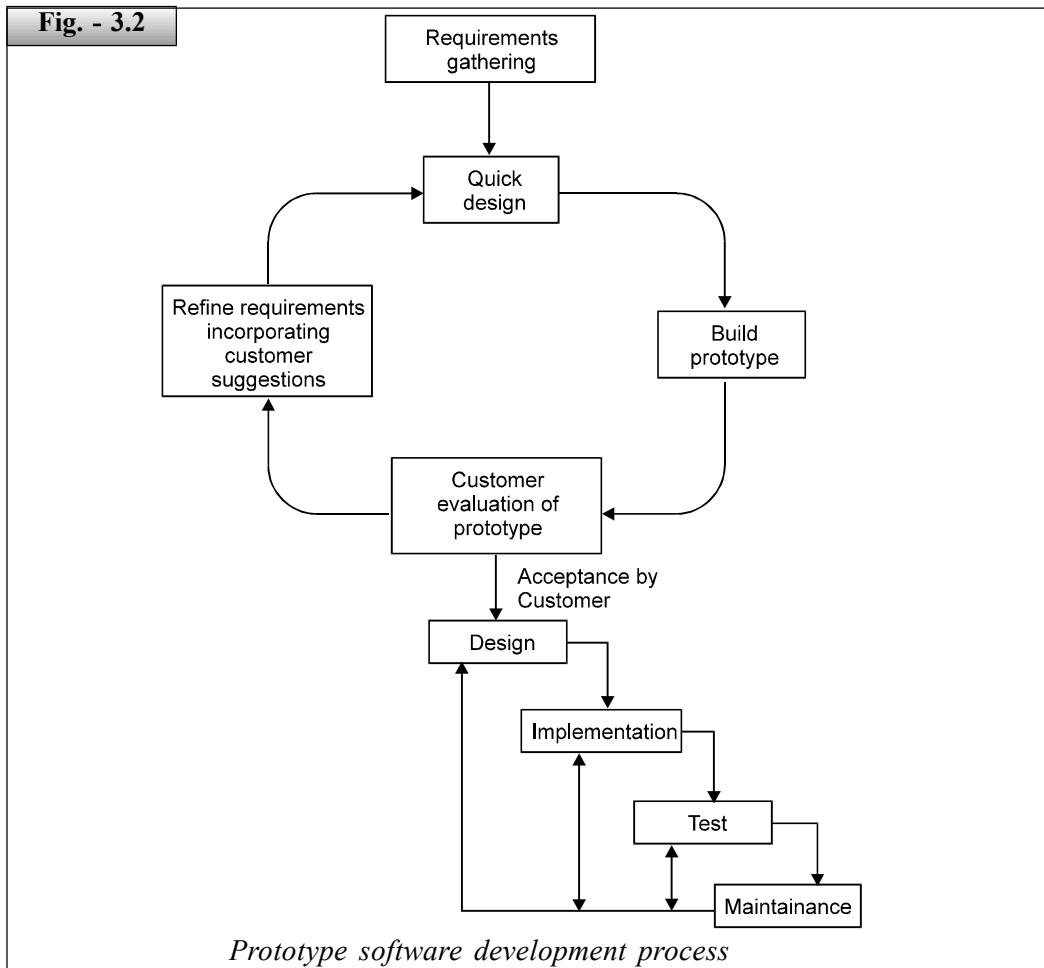
3.3.6 Advantages

- It enables maximum ordering in the *process implementation*.
- It provides a structured template for *software engineering*.

3.3.7 Disadvantages

- It is difficult for the customers to give all the requirements at one go, but this is a necessity for this model.
- It is difficult for the user to anticipate whether the final system constructed according to the *specifications* will eventually meet their *requirements*.
- Any change during the implementation can cause confusion, as the model is inherently sequential.
- Any working version can be seen only very late and hence in case of a serious error, the error has to be traced back to the requirements phase.
- Customers need to have patience for working with this model.

3.4 PROTOTYPING MODEL



- *Prototyping model* is based on the *iterative model*.
- A customer defines a set of general objectives for the software but does not identify detailed input, processing or output requirements.
- Customers' need for a 'quick design' and feedback led to the rise of this model.
- In this model, the prototype is developed based on *currently known requirements*.
- Development of this prototype also undergoes *design, coding and testing* but each of these phase is not done very formally or thoroughly.
- By using this *prototype*, the client can get a feel of the actual system.
- *Prototyping* is applicable for complicated and large systems when *requirements* are not known clearly.

3.4.1 Reasons for using prototyping model

- There are several purposes for a *prototype*.
- An important purpose is to illustrate the input data formats, messages, reports and the interactive dialogues of the customers. This is valuable for gaining better understanding of the customer's need.
- The *prototype model* is very useful in developing the *Graphical User Interface (GUI)* part of a system.
- The *prototyping model* can be used when the technical solutions are unclear to the development team.
- *Prototype* is the best or only way to solve technical issues like response time of a hardware controller or efficiency of sorting an algorithm.

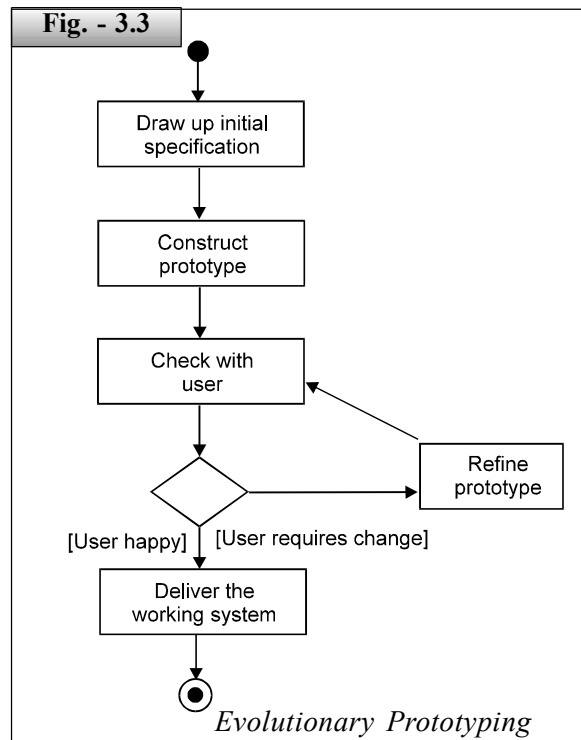
3.4.2 Type of prototyping

According to development approach, the *prototyping technique* is classified into two types :

- *Evolutionary prototyping*,
- *Throwaway prototyping*

3.4.3 Evolutionary prototyping

- This type of prototyping is based on the idea of developing an initial implementation, exposing it to user comment and refining it through repeated stage until an adequate system has been developed.
- *Evolutionary prototype* development is carried out within a systematic frame work, shown in the figure-3.3.



Evolutionary prototyping consists of several stages :

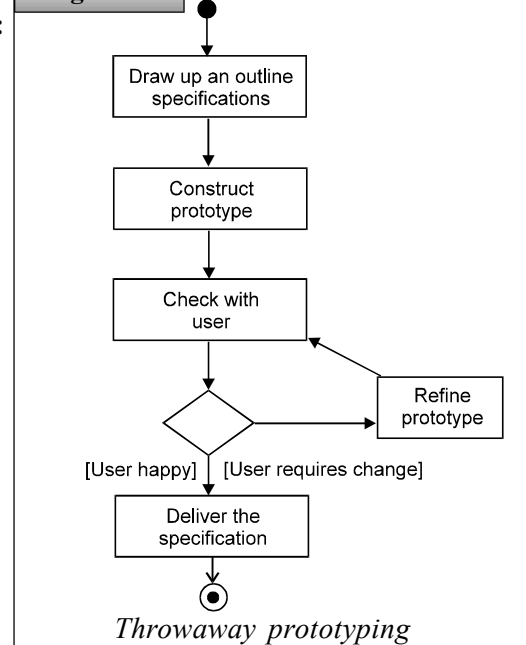
1. **Requirements definition** - a stage of thorough analysis is used to create an initial specification for the software.
2. **Prototype construction** - a prototype is built in a quality manner, including design, documentation and through verifications.
3. **Evaluation** - During evaluation, problem in the developer's perception of the customer requirements are uncovered. The prototype are the communication medium that enables the developer and customer to communicate with each other.
4. **Iteration** - Evaluation is carried out repeatedly until the prototype meets the objectives. The specification is updated with every iteration.

3.4.4 Throwaway prototyping

- In this type of prototyping model, the various versions of the system are constructed and then thrown away.
- A *throwaway prototype* implements only those requirements that are poorly understood. After the prototype is complete, the developer writes a full specification, incorporating what was learned and then constructs a full scale system based on that specification. Thus, the purpose of *throwaway prototyping* is the formulation of a validated specification.
- *Throwaway prototyping* is also called as rapid prototyping as it cost very little and take very little time to develop.
- *Rapid prototyping* seems to contradict the idea of using symmetric, careful methods during development; a prototype is produced in as quick a manner is possible.
- To be effective, *throwaway prototyping* is carried out within a symmetric framework, shown in figure 3.4.

The stages of throwaway prototyping are :

1. **Draw up an outline specification-** The first step in *throwaway prototyping* is the creation of an initial, often partial specification which contains area of uncertainty.
2. **Establish objectives** - The objective to develop a *throwaway prototyping* is to develop a system to prototype the user interface, to validate functional requirements, to explore certain new technologies or to demonstrate the feasibility of the application of management.
3. **Selection function** - Decide what to put into and what to leave out

Fig. - 3.4

of the prototype. It is controlled / determined by the objectives of the system.

4. **Construct prototype** - Fast, low cost construction is normally achieved by ignoring the normal quality requirements for the final product.
5. **Evaluate** - During evaluation, inconsistencies and short-comings in the developer's perception of the customer requirements are uncovered.
The prototype acts as an effective communication medium between the developer and customer.
6. **Iterate** - The prototype is rapidly modified, evaluation is carried out and the process repeated until the prototype meets the objectives.
7. **Deliver the specification** - The product of the prototyping process is a specification that meets the user's requirements. When the requirements are clearly established, the prototype is thrown away.
At this stage, a different software process model, such as the *waterfall model*, is employed to develop the software.
Users prefer to turn a throw-away prototype into a diversified system that is put into use.

The main reasons for this are as follows :-

- The characteristics such as performance, security and reliability are ignored during prototype development.
- During the prototype development, the change in the prototype indicates / points out the user needs. It is likely that these changes or modifications will have been made in an uncontrolled manner and not properly documented other than in the prototype code.
- The modification made during the development of prototype (*working model*) tends to degrade the architectural structure of the software.
It signifies that the maintenance of the software is difficult and an expensive task.

3.4.5 Rapid prototyping techniques

A *throwaway prototype* needs to be created quickly so that the user can evaluate its performance at an early stage. A prototype also needs to be altered quickly to incorporate the customer's needs as the prototype modifies to meet their requirements.

In this prototyping development, some specialised tools are utilised.

Some techniques for Rapid prototyping :

Here some techniques for fast prototyping are -

1) Use of high-level languages :

High level languages includes many facilities to buildup rapid prototyping as compared to other languages.

In this regard, *small-talk* is a language that can be used to prototype adventures *Graphical user Interface* (GUIs) with very little developer effort.

A demerit of *smalltalk* is that it can be a massive consumer of processor time and memory.

Therefore after prototyping it is necessary to rewrite the software codes in some other languages.

Hence the *small talk* is used for *throwaway prototyping*.

The “*Visual Basic*” software product has the features for *rapid prototyping* development, as it has a *GUI* platform with event driven facilities (*Drag and Drop from a palette*).

2) **Reuse of components :**

To provide the software as a realistic one, it is difficult task, because the requirements are incomplete.

For example, if a network solution is to be developed a *prototype* running on a stand-alone computer is developed. It simulates the complete system for the purpose of validation. But the developer is absolutely free from the discussion like consideration of network, volume of data and all possible problems associated with the performance of the software to be developed for a particular version.

3) **Error handling :**

In most of the cases, one-half of the whole software product is concerned with error handling.

It includes :-

- 1) Validation of user defined data feed to the system through the keyboard.
- 2) Efficiently handling the errors associated with input-output devices of the system.
- 3) Installation or development or use of exception handling software.
- 4) Installation or use of “*Fault tolerant*” software.

4) **Omission of features :**

Some of the features like logging of software, security and authentication are omitted in a prototype while developing it.

These above features requires high cost to be worked-out. Though these are accountable to the quality of the system, yet those are required to be omitted make the prototype development more quicker.

5) **Ignore functionality :**

This type of prototype is aimed to establish an acceptable user interface.

For example, suppose we were setting out to develop a new word processor. (word processor is a general purpose software that would be used by large no. of diverse people). We would, very quickly, create a mock-up of what would appear on the screen, while the actual functions of the word processor are not implemented.

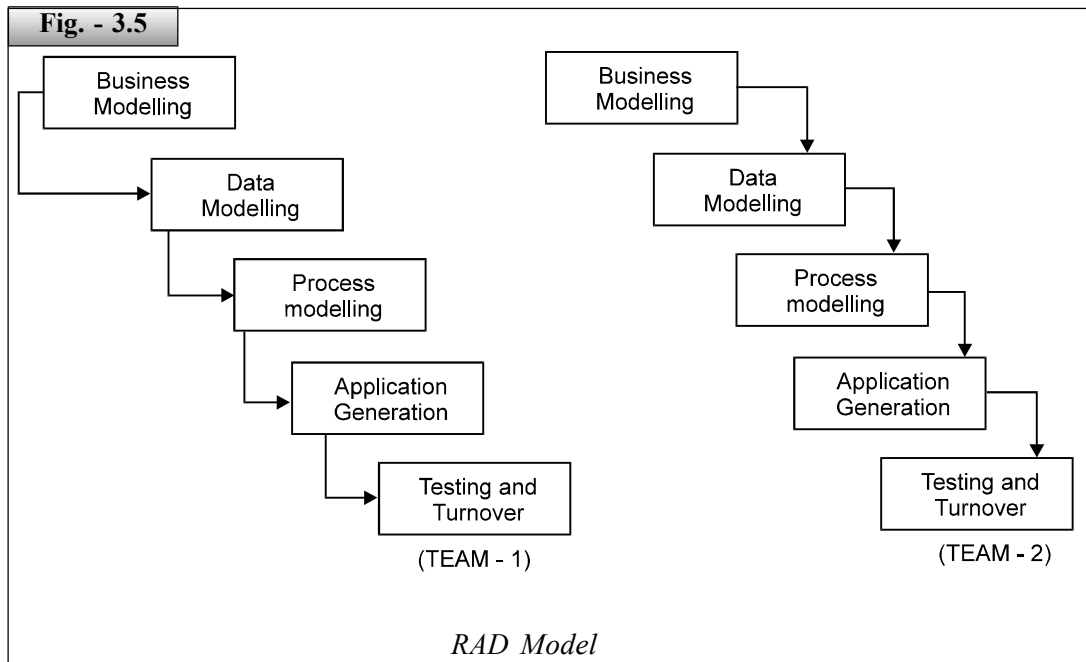
This type of prototype is often used during the design of user interfaces.

3.5 THE RAPID APPLICATION DEVELOPMENT (RAD) MODEL

As the time taken to develop Software using the *Linear Sequential Model (waterfall model)* is more, there was a need to develop Software within a shorter time frame, which ultimately led to formation of this model.

RAD is a *linear sequential model emphasizing* on short development cycles. It is a high speed adaptation of *linear sequential* model where rapidity is achieved by using

readymade components in the development of software product. This model is suitable for development of certain information system applications, where the business requirements are well defined.



3.5.1 Process of the RAD

Business Modelling

In this phase the information flow among the various business functions is modeled such that we know what information drives the business functions, information generated and the way it is processed.

Data Modelling

In this phase the characteristics of each object are identified and the relationship between these objects defined.

Process Modelling

In this phase the data objects of the *data modelling* phase are transformed to get the information flow for implementing the business function.

Application Generation

RAD uses *fourth generation techniques* like *automated tools* and *reusable components*, which are used to facilitate the construction of software.

Testing And Turnover

Since we reuse certain components that have already been tested, the overall *testing* time is reduced.

Problems with the RAD model

The *RAD model* is best suited for an extremely short development cycle where

requirements area is well understood and the project scope is constricted.

- It requires more human resources as multiple teams work concurrently.
- It requires commitment on part of the developers as well as customers to complete the system in a shortened time-frame.
- It is not suitable for systems that cannot be properly maintained.
- In case of high technical risk, it is not advisable to use the *RAD model* because it does not focus on minute details (For example, when a new application makes heavy use of new technology)

3.5.2 Advantages of RAD model

- By using the *RAD model* a product can be developed with in a very short time duration.
- A customer is involved at all stages of development it leads to a product achieving customer satisfaction.
- In order to increase productivity case tools and frame works are used.
- Feedback from the customers is available at the initial stages.

3.6 THE NEED FOR EVOLUTIONARY MODELS

Let us review some of the drawbacks associated with previous models.

- The *Linear sequential model (Waterfall model)* can only be applied for development via a straight line, i.e. until and unless the linear sequence is complete, the system is not ready for use.
- The *Prototyping model* cannot delivered a complete system in an operational mode, because it is primarily designed to bring forth the customer requirements in an easier and better fashion.

Coupled with these limitations there also was a need for

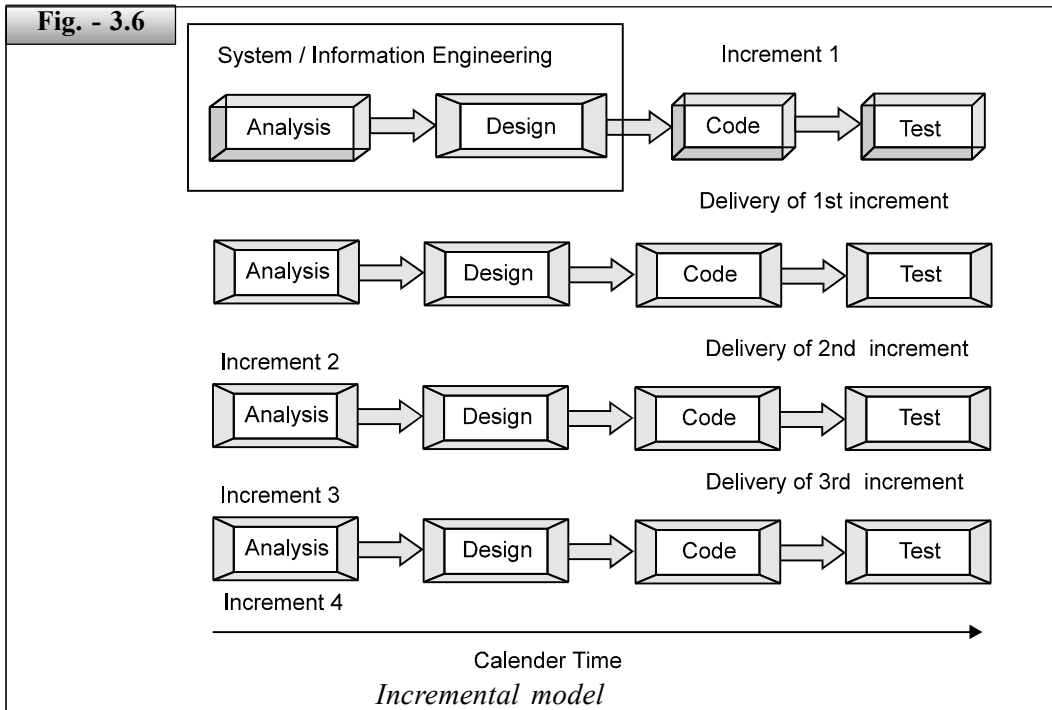
- * Tight schedule for marketing
- * Difficulty in understanding and developing the details of the product clearly
- * Change in requirements over a period of time.

The *evolutionary* model follows a flexible and non-monotonic approach. So, it is not only avoid of the earlier limitations, but also responds the needs mentioned above. The steps in the *evolutionary model* are as follows :

- Deliver something to the user.
- Get the feedback from the user.
- Adjust both design and objectives based on observed realities.
- In the *evolutionary model*, we will consider the following approaches :
 - Incremental model.
 - Spiral model
 - Component assembly model.
 - Concurrent development model.

3.7 THE INCREMENTAL MODEL

The *incremental approach* consists of stepwise development, where parts of some stages are postponed. This helps in producing some useful set of functions earlier in the development of the project. Here, each stage consists of expanding increments of an operational Software product. The increments may be delivered to the customer as they are developed.



This adds on to the value of the model by getting early user feedback. Thus instead of a two stage cascade of *code-and-test* and *integration-and-test* stages, which leads to monotonic application, we have a sequence of *code-and-test* and *integration-and-test* stages for various increments.

This incremental approach can be extended to all stages of the *life cycle*. Each increment is separately designed, coded, tested, integrated and delivered. In other words, we still follow the *waterfall model* but only for each separate increment. Increments are delivered one after another based on the feedback received from the customers. As users actually use the delivered parts, they begin to understand better what they actually need. Since each increment is simpler than the whole system, it is easier to predict the resources needed to accomplish the development task.

3.7.1 Advantages

The advantages of this model are given below :

- ☛ When limited resources in terms of manpower (staff) are present, *incremental model* is particularly useful. So even full staff is not available for all the increments, the project can be started.

-

3.7.2 Disadvantages

-

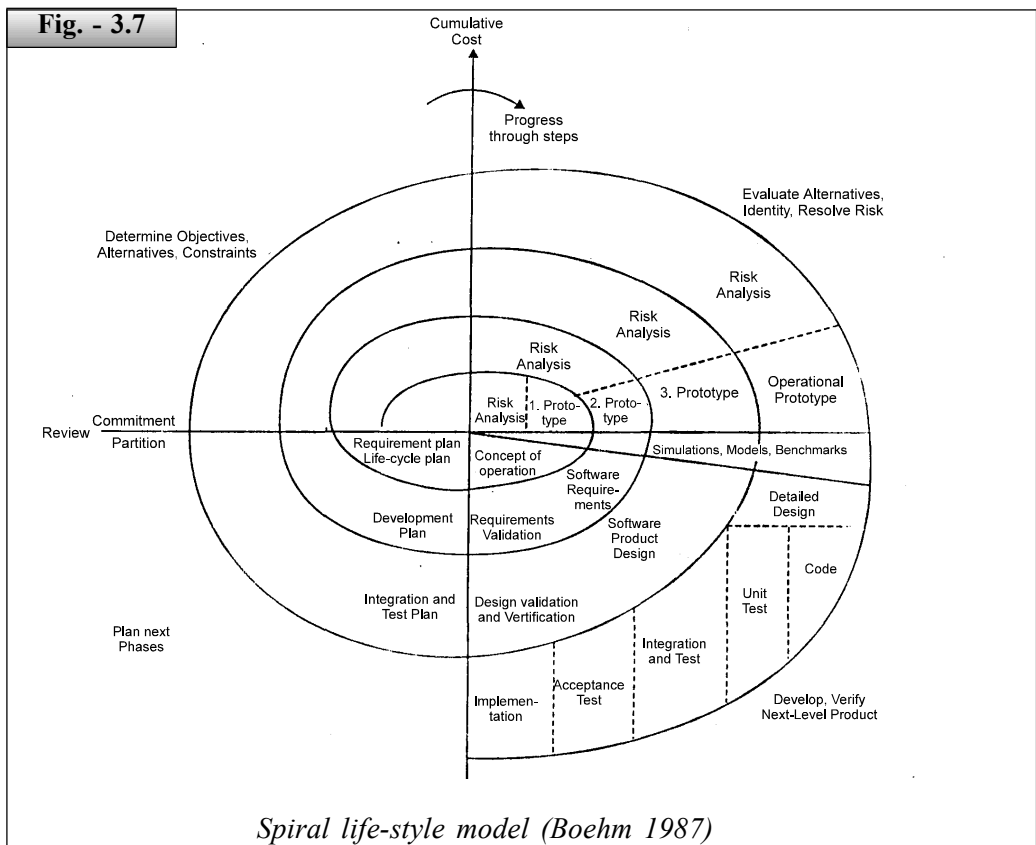
3.8 SPIRAL MODEL

During the development of software products, one of the most important factors which is inherent to the software project i.e. called un-certainty, which leads to high risk affecting the software operation or cost.

In the year 1986, *Barry Boehm* recognized this and tried to incorporate the “*Project risk*” factor into the *software life cycle* model and resulted with *spiral model*.

Spiral model is an “*evolutionary process model which is the mixture of iterative nature of prototyping with the systematic aspect of waterfall model*”.

The concrete look of the model is shown below figure 3.7.



Each phase is splitted arbitrarily into four highlighted activities.

- ☛ Planning
- ☛ Risk analysis
- ☛ Development
- ☛ Assessment

The radial dimension of the model represent cumulative cost. Each path around the spiral indicates increase in cost. The Angular dimension represents the progress of the activity. Each loop of the spiral from X-axis clockwise through 360° represents one phase.

Out of the four highlighted activities of a phase.

Planning includes determination of objectives, alternatives and constraints.

Risk analysis includes analysis of alternatives, identification and resolve of risks.

Development for product development and testing products.

Assessment for customer evaluation.

During the first phase, planning is performed, risks are analysed, prototypes are built and customers evaluate the prototype. During the second phase, a more refined prototype is built, requirements are documented and validated, and customers are involved in assessing the new prototype.

By the time, the third phase begins, risks are known and resolved with the development of next level of the software product. The product is thoroughly verified to eliminate high risks.

Finally in the fourth and final phase, adequate planning activities carried out for the next phase (as in SDLC) of the product.

Important features of this model is that each phase is completed with a review by the customer / user and the people concerned with the project (designers and programmers).

The number of loops (as shown in the figure 3.7) of the model not fixed, that varies w.r.t. the type of Software product.

Advantage :

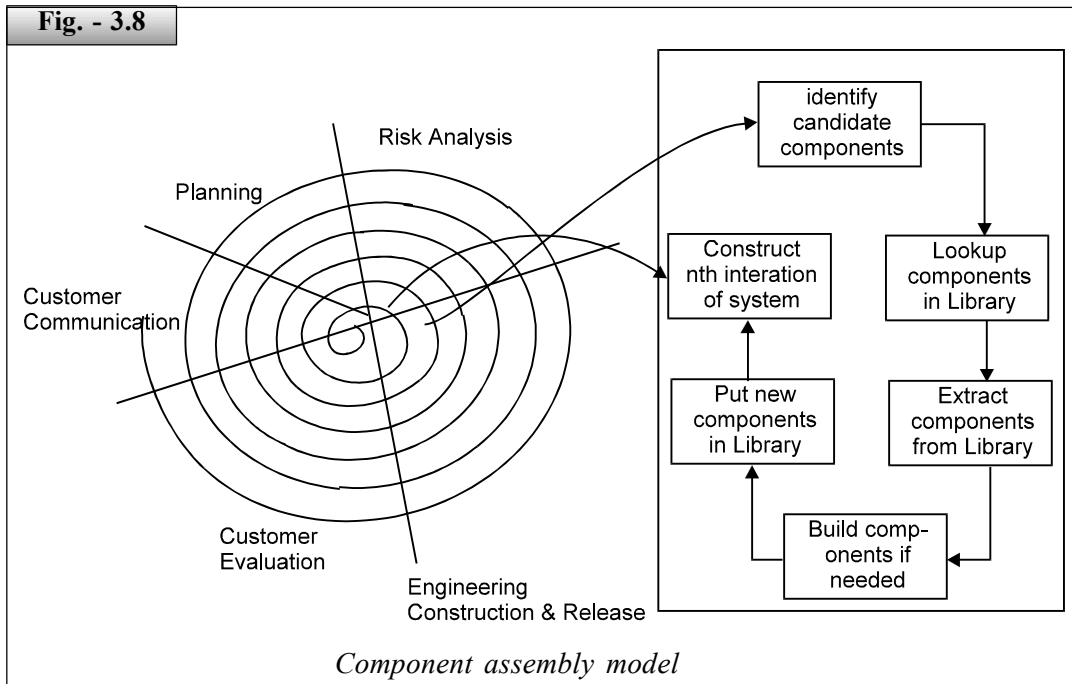
- It is a cyclic approach for incrementally growing a system's (software) degree of definition and implementation while decreasing the degree of risk.
- Ensures the customer / user commitment to feasible and mutually satisfactory solutions.
- It is a realistic approach to the development of large scale systems and Software.
- It incorporates software quality objectives into Software development.

Spiral model has some difficulties like, lack of explicit process guidance in determining objectives, constraints, alternative expertise on risk management what need to be resolved to treat this model universally applicable on *SDLC*.

3.9 COMPONENT ASSEMBLY MODEL

The *component assembly model* incorporates many of the features of the spiral model in terms of the iterative approach. It involves the concept of classes, which makes this model seem to be based on the object oriented paradigm. The various activities in using

this model begin with identifying the classes by examining the data used and algorithms to be applied. The corresponding data and algorithm are packed into a class. In a class library, classes created earlier are stored which can be reused. If they are not already present, they are created.



The first iteration of the application is composed, using existing classes and the newly created classes. The process flow then becomes spiral and ultimately enters the components assembly iteration during subsequent passes.

3.9.1 Advantages

The advantages of this model are listed below :

- It leads to software reuse (re-use of already existing classes)
- If results is reduction in development time of upto 70% and reduction in project cost of upto 84%, according to Quality System Management Associates, based on studies of reusability.
- System reliability is increased. Reused components exercised in working systems, are more reliable than newer components.

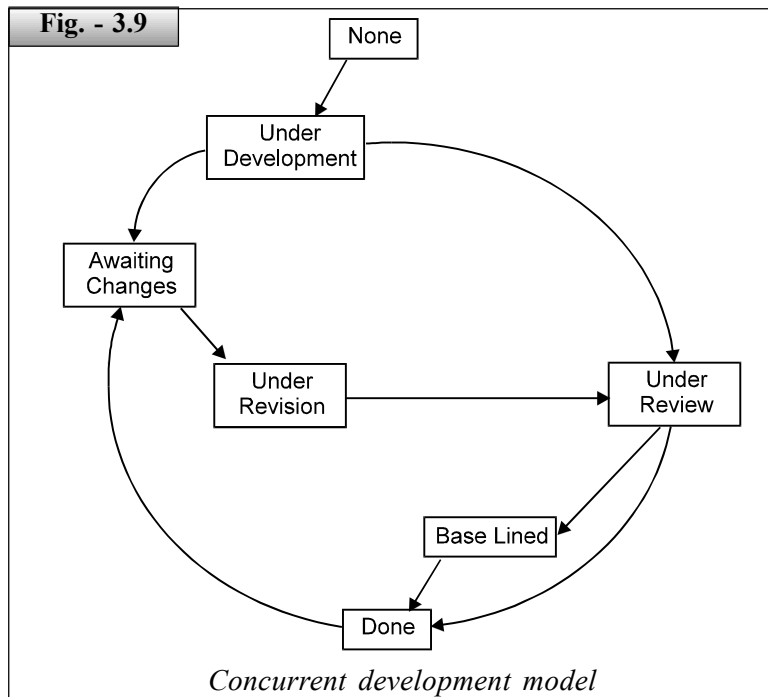
3.9.2 Disadvantages

- It is difficult to quantify what the cost reduction might be as there are costs associated with reuse. The components have to be discovered in a library, understood and sometimes adapted to work in a new environment. All this involves costs.
- Some Software engineers sometimes prefer to rewrite components, as they believe that they can improve reusable components.

3.10 THE CONCURRENT DEVELOPMENT MODEL

The *concurrent development model* is also known as *concurrent engineering*. It has been described in the following manner.

Project managers who track project status in terms of the major phases donot have any idea of the status of their projects. This is an example of tracking complex set of activities using simple models. Although a project may be is the coding phase, personnel on the project can be involved in several phases simultaneously.



The application development consists of a series of technical activities like *analysis*, *design*, *customer communication*, etc. What the concurrent development model proposes is that all these activities can be carried out concurrently, but each of which will reside in different state.

Concurrent development model defines a network of activities rather than confining the Software engineering activities to a linear sequence of events. Each activity on the network exists simultaneously with the other activities. Event generated within a given activity or at some place in an activity network trigger transitions among the states of an activity.

3.10.1 Advantages : The advantages of this model are given below :

- Applicable to all types of Software development and provides an accurate picture of the current state of a project.
- Particularly suited for client / server applications, has a set of functional components each of which can be designed and realised concurrently.

- ☛ Has been tested in operational systems and hence been exposed to realistic conditions.
- ☛ Overall process risk is reduced. If a component exists, there is less uncertainty in the costs of re-using that component than in the cost of development. This is particularly true when large components are reused.

3.11 THE FORMAL METHODS MODEL

The formal methods model comprises a set of activities that leads to mathematical specification of computer software. These method enables a software engineer to develop a computer - based system by applying rigorous mathematical notations.

3.11.1 The merits of this model are given below

- ☛ The formal specification provides insights in to software requirements and the software design. This reduces requirements errors and omissions.
- ☛ It is impossible to prove specification consistency and completeness. It is also possible to prove that an implementation conforms to its specification. The absence of certain class of errors may be demonstrated.
- ☛ Ambiguity, incompleteness and inconsistency can be discovered and corrected more easily.

3.11.2 The demerits of this model are listed below

- ☛ It is difficult to demonstrate that the relatively high cost of developing a formal system specification will reduce overall software development costs.
- ☛ Very few Software developers have the necessary knowledge to apply formal methods. Therefore extensive training is required.
- ☛ System customers are unlikely to be familiar with formal specification techniques.
- ☛ The development of formal models is currently quite time-consuming and expensive.

3.12 FORTH GENERATION TECHNIQUES (4GT)

These techniques involves specifying the software characteristics at a high level of abstraction. Then various tools are used to generate the source code. Software development environments supporting Fourth Generation Techniques paradigm include the following.

- Non-Procedural Languages for Data Base Query.
e.g. Structured Query Languages (SQL).
- Report Generation
- Data manipulation
- Screen interaction and definition.
- Code generation.

Like other paradigms, the 4GT begins with the requirements - gathering step. However these requirements cannot be directly translated into an operational prototype. The customers requirements might change. Hence, the customer developers dialogue remains an essential part of the 4GT approach. This is followed by a design strategy and the final implementation using a forth *Generation Language* (4GL).

The merits of this technique are given below :

- Automatic generation of code to generate the output.
- The time required to produce software is greatly reduced for small and intermediate applications.
- This technique coupled with *CASE Tools and code generators* Can help solve many software problems.

The demerits of the technique are also given below :

- Sometimes the resultant code generated by such tools is inefficient.
- This model demands greater analysis, design and through testing to save time through the elimination of coding.

3.13 COMPARISON AND SUITABILITY OF SOFTWARE LIFECYCLE MODELS

A comparison of the software lifecycle models is best accomplished by using the *waterfall model* as the basis for comparison. This is true for two reasons :

- ☛ *The waterfall model is most familiar to the majority of developers and*
- ☛ *Almost all other models were created to improve upon it.*

Besides comparison, suitability of all the important Software lifecycle models is also discussed below.

Waterfall lifecycle model

The waterfall lifecycle model is the most straightforward software development model as a series of processes that are expected to be carried out only once for the project. These are carried out in the proper order and the results documented for input to subsequent processes.

Because of the emphasis on thoroughness, documentation, and control. Waterfall model is best suited for large projects where requirements are known, stable and understood.

Its weaknesses include inflexibility for change, length of time before anything usable by the customer is produced and the implied requirement to complete every process correctly the first time.

Incremental lifecycle model

The incremental lifecycle model is similar to the waterfall in many respects, but it differs in that it produces some tangible results to the customer sooner. The initial processes of system requirements & feasibility, software requirements and general design are done in sequence, once for the overall project.

A partitioning into increments then occurs, where a number of different development; efforts, beginning with detailed design are identified. These increments can be planned as sequential or parallel efforts, depending upon the project characteristics and project constraints.

For the same reason as the *waterfall model*, *incremental lifecycle model* is suitable for large projects with requirements that are known, stable and understood.

Given these project characteristics, incremental lifecycle model is the best choice when early release of some parts of the software is beneficial or when earlier release of the entire system can be accomplished through a multiple teams working in parallel on

different increments.

When requirements are known and understood, but may not be stable, the incremental model is a logical choice because later releases can incorporate changes that surface during the earlier development efforts.

Use of this model requires careful partitioning of the system/product and well-defined interfaces between the increments, especially if they will be developed in parallel. Project managers using the incremental model must be aware of the need for additional attention to the coordination of multiple efforts. This includes additional attention to the coordination of multiple efforts. This includes additional effort that will likely be needed in the test process, where integration is addressed.

Evolutionary lifecycle model

The *evolutionary lifecycle model* applies in sequential aspects of the *waterfall model* and partitioning of the project borrowed from the incremental mode, but adds the evolution or the discovery of requirements.

Even though incremental development is planned when using the evolutionary model, the increments are not likely to be developed in parallel. This is because the purpose of each increment is to deliver a portion of the system with known requirements for subsequent increments.

Unlike the *waterfall and incremental models*, the system requirements & feasibility and software requirements processes are not done once for the entire project, but these are revised at the start of each *evolutionary development cycle* to incorporate the latest requirements.

Evolutionary lifecycle model is preferable lifecycle model when requirements are not fully known, but a subset of the requirements is known, stable and understood.

Benefits include the early delivery of some functions and the early testing of some assumptions before the entire system is built around them. The major weakness of this model is related to the inability to plan in detail at the outset of the project.

Because the requirements are not fully known, problems with the scope creep, inaccurate estimating and less than optimal architecture are possible. The predominately sequential nature of this lifecycle makes it not particularly rapid or cost efficient for complex systems.

Project managers using the evolutionary model must plan to revise the overall architecture as the system evolves. Flexibility must also be built -in at the individual software module level. Time that appears to be gained on the front end of a project because of the early release of the first increment may be spent on later increments on modification and integration efforts.

Spiral lifecycle model

At first glance, the spiral model appears to be the most radical departure from the waterfall model. This is because of the way it is diagrammed.

If the processes within the four cycles of the *spiral model* were drawn in a cascading fashion like the waterfall, the standard processes from requirements & feasibility through operation & maintenance would fall in the same order and would be done in sequence,

just as in the *waterfall* model.

The difference would be in the insertion of the non-standard processes of determining objects, alternatives, and constraints evaluating alternatives, and constraints, evaluating alternatives, identifying and resolving risks, and planning each cycle.

The non-standard processes may seem like a trivial difference, but as diagrammed, they account for approximately three-quarters of the project effort. However, this does not translate to increasing the project time, costs, etc. to for times what it would take using the waterfall lifecycle. Rather it represents a re-appropriation of project resources across additional processes that will dramatically reduce the amount of effort required on the standard processes.

The efficiencies achieved on the standard processes will be greatest on projects where several risks have been identified or are expected to surface during the course of the project. While all projects have some risks, not all require extensive use of risk resolving techniques such as prototyping, simulation, benchmarking, administering user questionnaires, etc. When extensive use of these techniques is anticipated, the spiral lifecycle model is indicated. Another indicator is when the developers and/or the customers know requirements. Development under the spiral model is cyclic rather than incremental, in that no software is release until the end of the final cycle.

The main benefit of the spiral model is in controlling project risks. Projects that would benefit the most are those that would be built on untested assumptions, if other lifecycle models were used. Use of the spiral model may prevent inadequate or unusable software from being built resulting in the avoidance of rework and increasing trust between the customers and the developers.

The shortcomings of the spiral are; that the additional processes are not as well defined as the standard processes, and there are not as many methods, techniques and tools to support them. In addition, the project manager must be prepared to manage from a risk-avoidance viewpoint. Developers, as well, must be prepared to identify, analyze and mitigate risks - a skill not always identified and cultivated.

3.14 SELECTION OF A LIFECYCLE MODEL

The selection of a suitable model is based on the following characteristics/categories :

- i) Requirements
- ii) Development team
- iii) Users
- iv) Project type and associated risks.

3.14.1 Characteristics of requirements

Requirements are very important for the selection of an appropriate model. There are number of situations and problems during requirements capturing and analysis. The details are given in Table 3.1.

Table 3.1 : Selection of a model based on characteristics of requirements

<i>Requirements</i>	<i>Waterfall</i>	<i>Prototype</i>	<i>Iterative Enhancement</i>	<i>Evolutionary development</i>	<i>Spiral</i>	<i>RAD</i>
Are requirements easily understandable and defined?	Yes	No	No	No	No	Yes
Do we change requirements quite often?	No	Yes	No	No	Yes	No
Can we define requirements early in the cycle?	Yes	No	Yes	Yes	No	Yes
Requirements are indicating a complex system to be built	No	Yes	Yes	Yes	Yes	No

3.14.2 Status of development team

The status of development team in terms of availability, effectiveness, knowledge, intelligence, team work etc., is very important for the success of the project. If we know above mentioned parameters and characteristics of the team, then we may choose an appropriate life cycle model for the project. Some of the details are given below.

Table 3.2 : Selection based on status of development team

<i>Development team</i>	<i>Waterfall</i>	<i>Prototype</i>	<i>Iterative enhancement</i>	<i>Evolutionary development</i>	<i>Spiral</i>	<i>RAD</i>
Less experience on similar projects	No	Yes	No	No	Yes	No
Less domain knowledge (new to the technology)	Yes	No	Yes	Yes	Yes	No
Less experience on tools to be used	Yes	No	No	No	Yes	No
Availability of training, if required	No	No	Yes	Yes	No	Yes

3.14.3 Involvement of users

Involvement of users increases the understandability of the project. Hence user participation, if available, plays a very significant role in the selection of an appropriate life cycle model. Some issues are discussed in Table below.

Table : 3.3 : Selection based on user's participation

<i>Involvement of Users</i>	<i>Waterfall</i>	<i>Prototype</i>	<i>Iterative enhancement</i>	<i>Evolutionary development</i>	<i>Spiral</i>	<i>RAD</i>
User involvement in all phases	No	Yes	No	No	No	Yes
Limited user participation	Yes	No	Yes	Yes	Yes	No
User have no previous experience of participation in similar projects	No	Yes	Yes	Yes	Yes	No
Users are experts of problem domain	No	Yes	Yes	No	No	Yes

3.14.4 Type of project and associated risk

Very few models incorporate risk assessment. Project type is also important for the selection of a model. Some issues are discussed in Table 3.4

Table 3.4 : *Selection based on type of project with associated risk*

<i>Project type and risk</i>	<i>Waterfall</i>	<i>Prototype</i>	<i>Iterative enhancement</i>	<i>Evolutionary development</i>	<i>Spiral</i>	<i>RAD</i>
Project is the enhancement of the existing system	No	No	Yes	Yes	No	Yes
Funding is stable for the project	Yes	Yes	No	No	No	Yes
High reliability requirements	No	No	Yes	Yes	Yes	No
Tight project schedule	No	Yes	Yes	Yes	Yes	Yes
Use of reusable components	No	Yes	No	No	Yes	Yes
Are resource (time, money people etc.) scarce ?	No	Yes	No	No	Yes	No

An appropriate model may be selected based on options given in four tables Firstly, we have to answer the questions presented for each category by circling a “yes” or ‘no’ in each table. Rank the importance of each category, or question within the category, in terms of the project for which we want to select a model. The total number of circled responses for each column in the tables decide an appropriate model. We may also use the category ranking to resolve the conflicts between models if the total in either case is close or the same.

SUMMARY

During the development of any type of software product, adherence to a suitable process model has become universal. Especially for development of any large software product, adoption of a suitable life cycle model is a prime necessity for the successful completion of the project.

We discussed only the central ideas underlying some important process models. Good software development organizations carefully document the precise process model they follow and typically include the following in the document:

- * Identification of the different phases
- * Identification of the different activities in each phase and the order in which they are carried out
- * The phase entry and exit criteria for different phases

- * The methodology followed to carry out the different activities

Adherence to a software life cycle model encourages the team members to perform various development activities in a systematic and disciplined manner. It also makes the management of software development projects easier.

The principle of detecting errors as close to their point of introduction as possible is known as phase containment of errors. Phase containment minimizes the cost to fix errors.

The essence and the strengths of the waterfall model are that:

- it divides a complex task into smaller, more manageable tasks
- each task produces a well-defined deliverable.
- each stage is carried out in sequence - there is no going back.

The goal of this approach is to maintain control during development.

The central goal of prototyping is to satisfy users' requirements. The key feature of the prototyping process model is the repeated demonstration of prototypes to users.

There are two approaches to prototyping - evolutionary and throwaway:

- in evolutionary prototyping an initial prototype evolves as requirements are clarified so that it becomes the final system
- throwaway prototyping uses rapid techniques to construct prototypes that are thrown away once users' requirements have been established.

It is important to establish the goal of using prototyping as part of a particular project. Although the goal is usually clarifying requirements, prototyping can also be used to design the user interface, demonstrate feasibility, verify that new technology will work or provide a training system.

The spiral model consists of a series of cycles. Each cycle consists of a series of steps. At every cycle, any risks to the successful progress of the project are assessed. Then an appropriate method is selected in order to minimize that risk. Thus the spiral model is essentially a cautious and robust approach to development.

The spiral model consists of a repeated cycle of small steps designed to assess and deal with risks at every cycle. Thus the spiral model is termed an iterative approach.

EXERCISE

1. Draw the schematic diagram of the waterfall model of software development? Also discuss its phases in brief.
2. Explain waterfall model in detail with the help of a diagram? State its advantages and also its limitations.
3. What is a prototype? Draw the schematic diagram of the prototyping model of software development? Also discuss its phases in brief?
4. What are the major advantages of first constructing a working prototype before developing the actual product?

5. Write a short note on Evolutionary development model. Also state its advantages?
6. Explain Iterative Enhancement model with the help of a suitable example.
7. Which life cycle model would you follow for developing software for each of the following applications? Justify your selection of model with the help of an appropriate reason:
 - (i) A game
 - (ii) A new text editor
 - (iii) A compiler for a new language
 - (iv) A software for hospital management
8. How are the risks associated with a project handled in the spiral model of software development?
9. What are the major phases in the spiral model of software development? Explain.
10. List the various models of software development and explain RAD model in detail.
11. Compare the spiral model with prototyping model by giving its advantages and disadvantages.
12. What is the need and use of an Evolutionary development model?
13. Discuss the merits and demerits of various models of software engineering.
14. Write a short note on comparison between various software development life cycle models.
15. Irrespective of whichever life cycle model is followed for a software product development, why is it necessary for the final documents to describe the product as if it were developed using the classical waterfall model?
16. Suppose you were to plan to undertake the development of a product beset with a large number of technical as well as customer-related risks, which life cycle model would you adopt? Justify your answer.
17. Suggest a suitable life cycle model for a software project which your organization has undertaken on behalf of a certain customer who is likely to change his requirements frequently. Justify your answer.



Feasibility study

4.1 INTRODUCTION

This is the vital phase of SDLC. The primary aim of feasibility study phase is to determine whether a software project would be financially and technically feasible. But in general, for any commercial product, feasibility study aims to verify how the product is financially viable, technically feasible, socially acceptable and commercial profitable.

It involves the detail analysis of the problem and gathering adequate information relating to the software product.

There are different activities carried out during feasibility study.

Now the question arises :

- Who does it ?
- How it is done ?

to explain it, some strategic managerial activities need to be studied.

4.2 SOFTWARE PROJECT MANAGEMENT

Software project which is treated in an engineered way is distinct from other types of engineering in many way.... as

- Software product is intangible
- No standard process for development.
- Software products are often changable.

Software project management is a set of activities to ensure that a Software product that meets the quality specification as per the requirements, proceed on schedule, within the budget and also within the technological domain.

4.3 ROLE OF PROJECT MANAGER

Software project manager takes the overall responsibility of steering a project to success. But it is difficult to describe the job responsibilities of a project manager. It ranges from invisible activities like building up team morale to highly visible customer satisfaction.

The activities can be broadly classified into project planning, project monitoring and project controlling.

Project planning activity is conducted before the development, where as the project maintaining and controlling activities are carried out during the development of the project.

4.4 ROLE OF SYSTEM ANALYST

In more details, software project management is intended to the planning, organisation, technical and economical control of project execution.

The major activities here is

- Proposal activity
- Project planning
- Project scheduling
- Project tracking or monitoring
- Project controlling.
- Selection and evaluation of personnels.
- Project report writing.

Proposal writing is a skill that is achieved by experience. It is a critical task as the existence of many Software organisations depend on having enough proposals accepted and contracts awarded. There is not set guidelines for this task.

Project planning is concerned with identifying the activities, milestones and deliverables produced by a project. Basically, it is conducted before the development activities.

Project scheduling is one of the most vital tasks involving separation of total work (project) into different tasks and assesment for the completion of tasks.

Some tasks can be carried out in parallel. A clear co-ordination needed to run the tasks ensuring the optional use of workforce.

Project tracking / monitoring is a continuous project activity. The managers must keep track of the progress of the project and compare the actual and planned project and cost.

Project controlling is to ensure that the development proceeds as per the plan which is again in a continuous fashion.

Personnel selection and evaluation is really a challenging task which is usually conducted by project manager(s) to select manpower to work on them project with skill and appropriate experience.

Project report writing is the responsibility of project manager to write reports both for the clients and organisation. The report must be concise, coherent that possesses abstract critical information about the whole project.

4.5 PROJECT MANAGEMENT DIFFICULTIES

Software project management is a very difficult and challenging task. These are the major difficulties.

- ☛ Software systems uniqueness
- ☛ Possibility of multiple solutions.

- ☛ Individuality of programmers.
- ☛ Speed of technological advancements.
- ☛ Invisibility of software products.

4.6 SOFTWARE PROJECT PLANNING

The real purpose of software project planning is to establish reasonable plans for performing the Software Engineering and managing the Software product.

It involves developing estimates for the work to be performed, managing the software project activities and addresses the commitments to the customer according to the resources, constraints and capabilities of the software project. Software project planning is broadly classified into 5 different categories like -

- i) Quality plan
- ii) Validation plan
- iii) Configuration plan
- iv) Maintenance plan &
- v) Staffing plan.

A list of activities are carried out in software project planning as follows :

- The software developer group initiates the planning in the early stages of the overall project planning.
- Identification of commitments made to individuals & groups are reviewed w.r.t the documented procedure.
- A software life cycle with predefined stages of manageable size is identified or defined.
- The software development plan is developed according to the documented procedure.
- The plan for software project is documented.
- Software work products that are needed to establish and maintain control of the project are identified.
- Deriving the estimates for the size of software work products.
- Estimates for the project effort, costs, resources according to the documented procedure.
- Scheduling of software project.
- Risks associated with the cost, resource, schedule and technical aspects of the project are identified, assessed and documented.
- Plans for the software engineering facilities and support tools for the project to be carried out.

4.7 SOFTWARE PROJECT MANAGEMENT PLAN (SPMP)

Once the software project planning is complete, the software project managers arrange the project planning results in form of a document called software project management plan.

We discuss here the IEEE SPMP as follows. It's a framework by IEEE cos 8.1 in the year 1987.

1. *Introduction*

- Project overview
- Project deliverables
- Evolution of SPMP
- Reference material
- Definitions & acronyms

2. *Project organisaiton*

- Process model - organisational structure
- Organisational application boundaries and interfaces
- Project responsibilities

3. *Managerial process*

- Management objectives and priorities
- Assumption, dependencies and constraints.
- Risk management
- Monitoring and controlling mechanisms.
- Staffing plan.

4. *Tehnical process*

- Methods, tools and techniques
- Software decoumentation
- Project support functions

5. *Work packages, shcedule and budget*

- Work packages
- Dependencies
- Resource requirements
- Budget and resource allocation
- Schedule.

6. *Additional components*

7. *Index*

8. *Appendices*

All the above highlighted elements of IEEE SPMP are briefly described.

1. *Introduction :*

It provides an overview of the project and the product to be developed.

- 1.1 *Project overview* : A brief description about the project objectives, product to be delivered, the activities and their resulting work products along with the list of resources, master schedule and master budget.
- 1.2 *Project deliverables* : The items to be delivered to the client are list with dates.

- 1.3 *Evolution of SPMP* : In this section, no concrete plan can be costed rather it requires continual update of experience and of charge within both the clients and the development group. Some formal procedures and mechanisms for changing the plan (if required) are described.
- 1.4 *Reference material* : All referenced documents used in SPMP are listed.
- 1.5 *Definitions and acronyms* : It ensures that the information under this section used in SPMP are well understood to everyone.

2. **Project organisation :**

It provides the information that how the product is to be developed, both from the viewpoint of software product and the organisational structure of the developers.

- 2.1 *Process model* : It is specified in terms of activities like designing, testing, review of the product along with the project management or configuration management.
- 2.2 *Organisational structure* : It is important to demonstrate the lines of authority and their responsibility with the developer organisation.
- 2.3 *Organisational application boundaries and interfaces* : It clearly identifies the interaction of clients with the developer organisation. Administrative and management boundaries between the project and other entities (support groups) are clearly defined. It also includes configuration management and SQA (Software Quality Assurance).
- 2.4 *Project responsibilities* : The responsibilities of all individuals related to the project are identified.

3. **Managerial Process :**

This section describes how a project is to be managed.

- 3.1 *Management objectives and priorities* : The philosophy, goals and priorities for management are described under this section. It may include frequency and mechanism of reporting, priorities among requirements, schedule and budget for the project, risk management procedures etc.
- 3.2 *Assumption, dependencies and constraints* : Any assumptions, constraints in the specification document appear under this section.
- 3.3 *Risk management* : A clear list of various risk factors along with their tracking mechanism of a project are listed out.
- 3.4 *Monitoring and controlling* : Reporting mechanism for the project are described in detail including review and audit.
- 3.5 *Staffing plan* : The numbers and types of personnel required are listed together with the duration for which they are needed.

4. **Technical process :**

Technical aspects of the project are specified under this section.

- 4.1 *Methods, Tools and Techniques* : Technical aspects of hardware and software described in detail covering operating system, the system on which a software product will be installed, development techniques, testing methods, technical

standards to be used, programming language, documentation standards, CASE tools, coding standards are included.

- 4.2 *Software documentation* : It includes documentation requirements, namely milestones, baselines and review for software documentation.
- 4.3 *Project support functions* : Detail plans for the supporting functions such as configuration management, quality assurance, test plan etc.

5. Work packages schedule and budget :

This section emphasises on the work packages, their interdependencies, resource requirements and associated budgeting allocations.

- 5.1 *Work packages* : It specifies the work packages with their associated work products broken down into activities and tasks.
- 5.2 *Dependencies* : The inter dependencies among the work packages and dependencies on external events are specified.
- 5.3 *Resource requirement* : The resources required for the entire project are presented as a function of time.
- 5.4 *Budget and resource allocation* : The allocation of resources and budget (cost) to the various project functions, activities and tasks is performed.
- 5.5. *Schedule* : It is a master plan tells about the approximation of completion of project on time and within the budget.

6. Additional components :

Additional components may required to be entertained in the plan like.

- Hardware procurement - Lower level management plan.
- Installation - Security plan.
- Product maintenance - Training plan
- Independent verification and validation.

NOTE : *The SPMP document may vary based on the nature of software project and the organisation policies.*

4.8 SOFTWARE PROJECT SCHEDULING

Software project scheduling is one of the most difficult jobs of a software manager, because most of the information used in it is empirical and based on a particular individual's experience.

There are many techniques for estimating the time and resources required for a project. *"estimates of the time and resources required for a development project are almost always optimistic, especially when the system being developed is technologically advanced or is significantly different from other projects that the organisation has developed"*.

4.8.1 Project scheduling activities

It involves :

- Identification of tasks and activities within the project.
- Identification of dependencies among the activities.

- Estimate the resources (time) for each activity.
- Allocate people.
- Determine the order of entire activities.
- Estimate the resource (time) for entire project.

4.8.2 Software project scheduling techniques

The following are important software project scheduling techniques.

- Work Breakdown Structure (WBS)
- Activity chart
- Project Evaluation Review Technique (PERT)
- Gantt chart
- Critical Path Method (CPM)

4.8.2.1 Work Breakdown structure (WBS)

It's a hierarchical decomposition or breakdown of project or major activity into successive levels, in which each level is a finer breakdown of the preceding one.

It can be shown with the help of following figure 4.1.

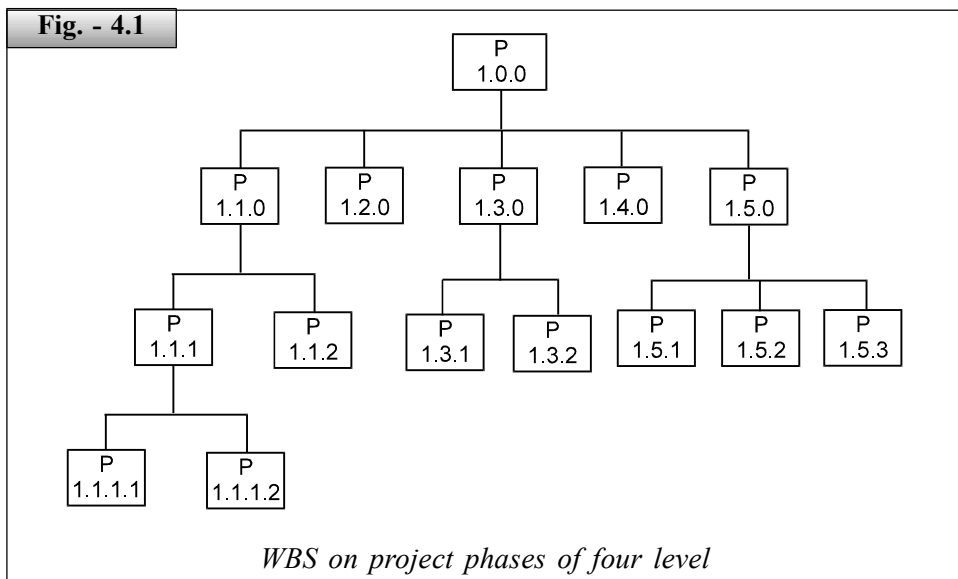


Figure shows the WBS diagram of a project phases of four levels.

P(1.0.0) - The whole project at *Level - 1* :

P(1.1.0), P(1.2.0), P(1.3.0), P(1.4.0), P(1.5.0) - Decomposition of project (1.0.0) into *level-2* :

P(1.1.1), P(1.1.2) - decomposition of P(1.1.0) at level 2 into level 3 and so on...

Mainly the techniques implemented here are based on splitting the goal of the project into several intermediate goals. Each intermediate goal can in turn be split further. This process is continued / repeated until each goal is small enough to be well understood.

Finally each goal can be individually planned for its

- *resource requirements*
- *assignment of responsibilities and*
- *scheduling etc.*

Uses of WBS

The potential use of WBS include :

- * Dividing the project into phases.
- * Dividing the project into responsibility areas within the organisation.
- * Dividing the schedule of the project into subschedules whose inter-relations are known.
- * Factors forming a good reason to filling the cost of the project by defining clear target to it.
- * For hierarchical outlining and coding for the work to be done.
- * For enabling integration, planning and managing of the project from both financial and scheduling perspectives.

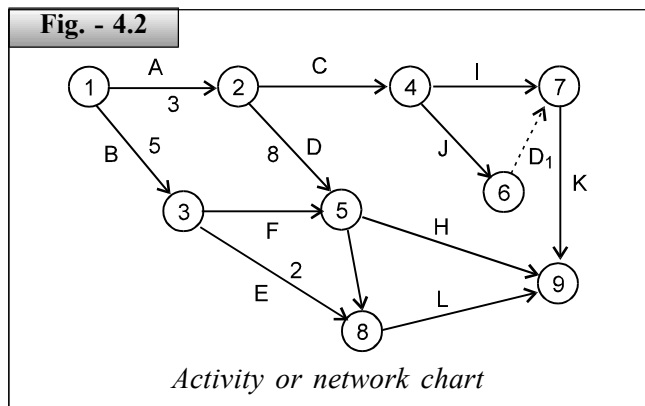
4.8.2.2 Activity chart / Network

This approach was originated from US navy in 1958. This was used for evaluating the performance of large development project.

The technique consists of three stages

- **Planning**
 - * Identify tasks and estimate duration of times.
 - * Arrange the feasible sequences.
 - * Draw diagram.
- **Scheduling**
 - * Establish time table of start and finish time.
- **Analysis**
 - * Establish float
 - * Evaluate and revise as necessary.

The activity / network chart is needed to complete a project, showing the order in which the tasks need to be completed and the dependencies between them.



Where A, B, C L	=	Activities
D ₁	=	Dummy activity
(1), (2) (9)	=	Nodes
2, 3, 5, 8	=	Activity duration.

There are different network analysis techniques like :

PERT - Project Evaluation and Review Technique.

CPM - Critical Path Method

RAMS - Resource Allocation and Multiproject Scheduling.

MAP - Man power Allocation Procedure

LCS - Least Cost Scheduling

GERT - Graphical Evaluation and Review Technique.

Here, we discuss some rules for constructing network / network diagram.

Rules for network construction

- 1) No event should occur until every activity preceding it has completed.
- 2) An activity can't occur twice i.e. path of an activity can't form a loop.
- 3) Each activity must start from and terminate in an event.
- 4) Time flows from left to right.
- 5) Dummy activities should be introduced if absolutely necessary.

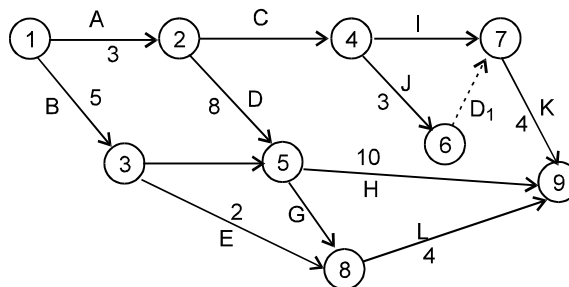
How to have numbering for the events ?

There is a rule which was devised by *D.R. Fulkerson* for numbering the events after the network is drawn.

The steps are :

- *Event number should be unique.*
- *It should be carried out in a sequence from left to right.*
- *The initial event which has all outgoing arrows with no incoming arrow, is numbered 0 or 1.*
- *The head of an arrow should always bear a number higher than the one assigned at the tail of the arrow.*

Example : 4.1



- (1), (2) ... (9) - Activity nodes (—>)
 A, B ... L - Activities

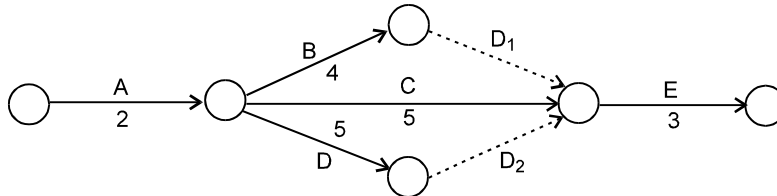
2, 3, 5, ... Activity duration

D_1 - Dummy activity.

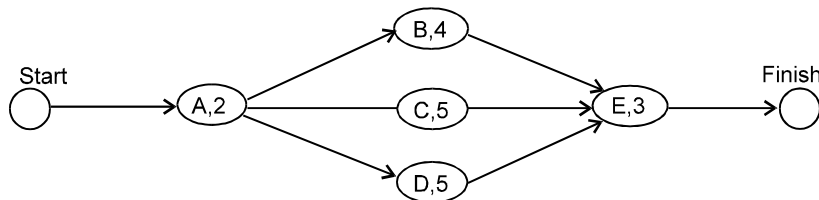
Activity - On - Node diagram (A_0N diagram)

A_0N diagram is also called as precedence diagram. In A_0N diagram the activities are denoted by nodes while precedence relationships are represented by arrows as follows.

a) Arrow diagram.



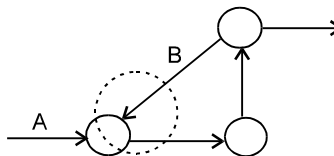
b) A_0N diagram.



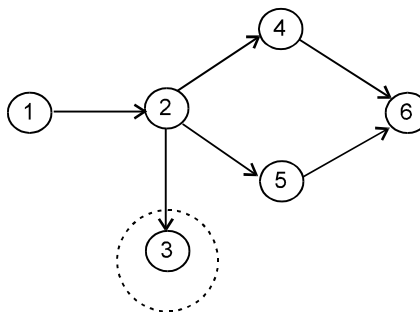
Common Errors in network drawing :

There are three types of errors are most common in network drawing.

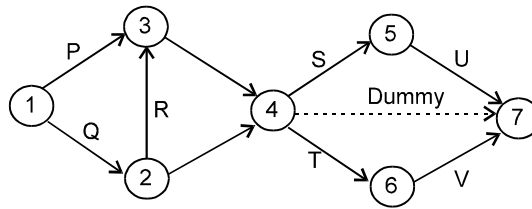
(i) Formation of loop.



(ii) Dangling (hanging)



- No activity should end without being joined to the end event, (3) is not joined to (6)
 - A dummy activity should be introduced from node (3) to node (6) to avoid dangling.
- (iii) Redundancy :

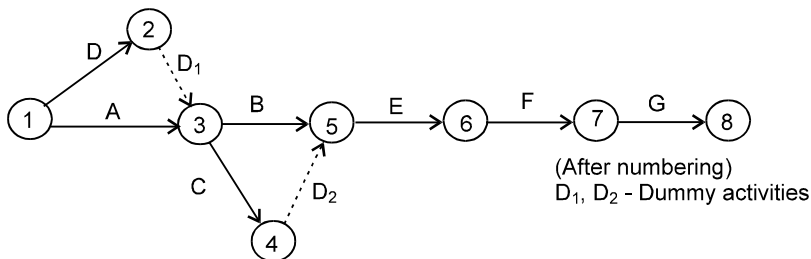
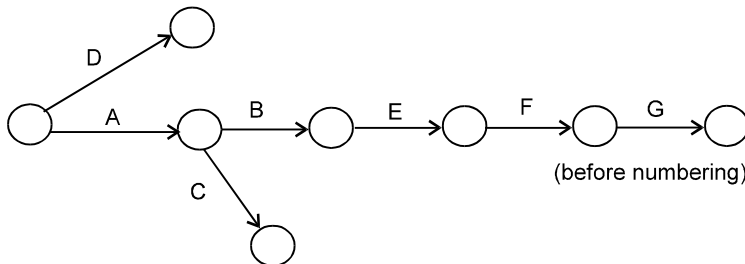


Here, the dummy activity is redundant and can be eliminated and the network is redrawn.

Example : 4.2 Draw a network diagram for the following table.

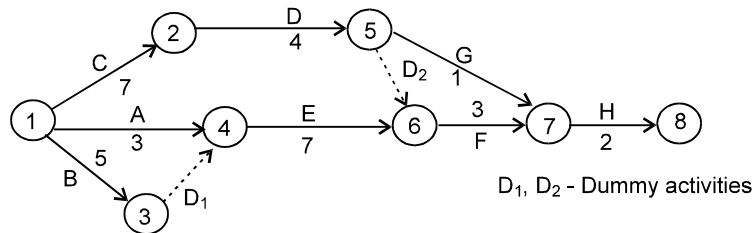
Activity	Immediate Predecessor
A	—
B	A, D
C	A, D
D	—
E	B, C
F	E
G	F

Network diagram



Example-4.3 : Draw the network diagram

Activity	Time	Immediate predecessor
A	3	—
B	5	—
C	7	—
D	4	C
E	7	A,B
F	3	E,D
G	1	D
H	2	F, G



4.8.2.3 CPM and PERT

One of the most challenging jobs that any project manager can take on is management of large scale project that requires co-ordinating numerous activities throughout the organisation. For this two closely related, operation research techniques available to assist the project manager. These techniques use networks to help and display co-ordinating of all the objectives.

There are different types of large projects including sophisticated software products are handled efficiently using CPM and PERT techniques.

Terminology :

- Event :** An event is a specific instant of time which marks start and end of an activity. It is represented by a circle and the event number is written within the circle.
- Activity :** An activity is an element of a project and it may be a process (A project has no. of operations/ tasks called activities)

An activity is shown by an arrow ($\longrightarrow$) and it begins & ends with an event.

Activities are classified as :

- Critical activity :** In a network diagram, critical activities are those which if consume more than their estimated time, the project will be delayed. An activity is called critical if the sum of its earliest start time and time taken by it is equal to the *latest finishing time*. A critical activity is marked either by a thick arrow.

- *Non-critical activity* : These are the activities, have provision (*float or slack*) so that, even if they consume a specific time over and above the estimated time, the project will not be delayed.
- *Dummy activity* : When two activities start at the same instant of time, the head events are joined by dotted arrow called dummy activity. It does not consume any time. It may or may not be critical. Dummy activity becomes a critical activity when its earliest start time (EST) is same as its latest finish time (LFT).
- *Critical Path* : It is that sequence of activities which decide the total project duration. Critical path is formed by critical activities. A critical path consumes maximum time. A critical path has zero float.
- *Duration* : Duration is the estimated or actual time required to complete a task or an activity.
- *Total project time* : It is the time which will be taken to complete a project and is found from the sequence of critical activities. It is also the duration of critical path.
- *Earliest Start Time (EST)* : It is the earliest possible time at which an activity can start and is calculated by moving from first to last event in the network diagram.
- *Earliest Finish Time (EFT)* : It is the earliest possible time at which an activity can finish.

$EFT = EST + \text{Duration of that activity.}$

- *Latest Finish Time (LFT)* : It is calculated by moving backward i.e. from last event to first event of the network diagram. It is the last time of the head event.
- *Latest Start Time (LST)* : It is the latest possible time by which an activity can start.

$LST = LFT - \text{Duration of that activity.}$

- *Float or Slack* : Slack is with reference to an event and float is with respect to an activity. In other words, slack is used with PERT and float with CPM but in general practice, they may be used inter-changeable. It may be float or slack, it means spare time, a margin of extra time over and above its duration which a non-critical activity can consume without delaying the project. Float is the difference between the time available for completing an activity and the time necessary to complete the same.

- (i) *Total float* : It is the additional time which a non-critical activity can consume without increasing the project duration.

$\text{Total float} = (LST - EST)$

or

$(LFT - EFT)$

It can also be negative.

- (ii) *Free float* : It all the non-critical activities start as early as possible. The surplus time is the free float.

$\text{Free float} = \text{EST of tail event} - \text{EST of head event} - \text{Activity duration.}$

- *Independent float* : The use of independent float of an activity does not change the float in other activities. Independent float can be used to advantage if one is interested to reduce the effort on a non-critical activity in order to apply the effort on a critical activity.

Thereby reducing the project duration. The independent float is associated with an activity, is not reduced by delaying previous operations whereas such a reduction can be noticed with free float.

Independent float = EST of tail events

- LFT of head event – Activity duration.
(if it comes to negative, is taken as zero.)

A. Critical Path Method (CPM)

It's a technique which is applied during a project activities, to schedule and control the project and experience a good amount of saving.

CPM technique : CPM employs the following steps to accomplish project planning.

1. Break down the project into various activities systematically.
2. Level all the activities.
3. Arrange all the activities in a logical sequence.
4. Construct the arrow diagram.
5. Number all the nodes (events) and activities.
6. Find the time for each activity.
7. Mark the activity times in the arrow diagram.
8. Calculate early and late, start and finish times.
9. Tabulate various times and mark EST and LFT on the arrow diagram.
10. Calculate total float for each activity.
11. Identify the critical activities and mark the critical path on the arrow diagram.
12. Calculate the total project duration.
13. If it is intended to reduce the total project duration, crash the critical activities of the network.
14. Optimise the cost.
15. Update the network.
16. Smooth the network resources.

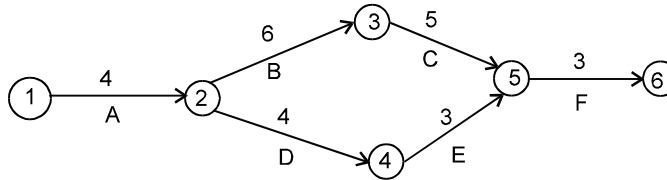
Example -4.4 :

Draw network diagram and calculate the critical path, EST, LST, EFT, LFT, Total float, Free float, independent float.

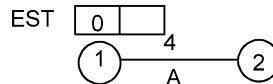
The activity and duration given below.

<i>Activity</i>	<i>Duration (days)</i>
1-2 (A)	4
2-3 (B)	6
3-5 (C)	5
2-4 (D)	4
4-5 (E)	3
5-6 (F)	3

Solution : Network Diagram is :



EST is calculated by starting from event 1 i.e., activity A and giving it a time 0.



Now EST of activity B i.e., node 2

= 0 + duration of activity

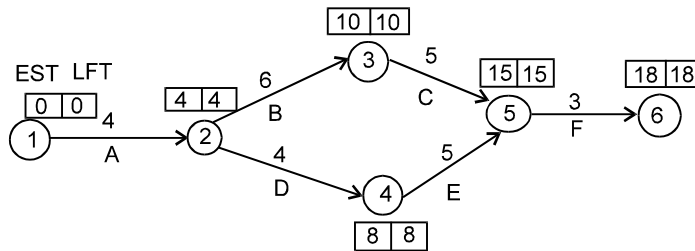
= 4

EST of activity i.e., node 3

= EST of activity B i.e., node 2 + Duration of activity B

= 4+6

=10 and so on.



EST of activity F i.e., node 5 can be found by following two path.

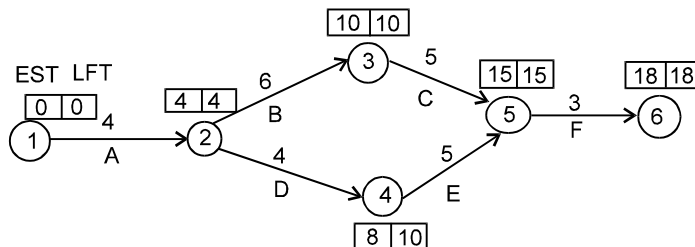
1-2-3-5, 1-2-4-5

The path 1-2-3-5 gives =15 days, where 1-2-4-5 gives 11 days.

Naturally maximum value (15) is selected because until activity C is completed which ought to finish on 15th day, activity F cannot be started.

The EST of other activities is calculated by proceeding similarly in forward direction.

LFT is calculated in a similar manner as EST but proceeding backward from the last event to the first one.



LFT of activity F i.e., node 6 is 18 (i.e., in finish node $EFT = LFT$)

LFT for activity C and E (i.e., node 5)

$$= 18 - \text{Duration of activity F}$$

$$= 18 - 3$$

$$= 15$$

Similarly proceeding backward we have LFT of activity B (i.e., node 3) and D (i.e., node 4) at activity A i.e., node 2.

There are two paths i.e., one from node 3 and other from node 4 i.e., 4 and 6. Choose the minimum days i.e., 4, and start node will be 0. From the above diagram now we can solve our problem directly.

Activity	Duration	EST	LST	EFT	LFT	Total float	Free float	Independent float
1-2(A)	4	0	0	0	4	0	0	0
2-3(B)	6	4	4	10	10	0	0	0
3-5(C)	5	10	10	15	15	0	0	0
2-4(D)	4	4	8	8	12	4	0	0
4-5(E)	3	8	12	11	15	4	4	0
5-6(F)	3	15	15	18	18	0	0	0
		Directly from Network diagram	LFT-Duration	EST+Duration	Directly from Network diagram	LST-EST or LFT-EFT		

Total float = (LST-EST) or (LFT - EST)

for example total float for activity D = (8-4) or (12-8) = 4

Free float = EST of tail event -EST of head event -activity duration.

For example free float for D = 8-4-4 =0

Independent float = EST of tail event -LFT of head event -activity duration.

for example, Independent float for activity 0 = 8-4-4 = 0

Critical path 5 one, whose TF = 0 i.e., 1-2-3-5-6.

It consumes the maximum time. The total duration is 4+6+5+3 = 18 days.

The critical path can be shown as a dark linear parallel line in the network diagram.

B. PERT (Project / Program Evolution and Review Techniques.)

PERT was first used in 1957 as a method of planning and controlling the polaris missiles program by *BOCZ Allen and Humilton* together with *US Naval Department*. The aim was to finish the project in advance.

PERT describes the basic network technique which includes planning, monitoring and control of projects.

PERT is suitable in those projects where activity times cannot be reliably predicted.

PERT deals with probabilistic model.

In *CPM* we assume that the expected time is actual time taken, where as in *PERT* the probability distribution of activity time is based upon three different time i.e. optimistic time (t_o), pessimistic time (t_p) and most likely time (t_m).

The PERT planning technique consists of the following steps.

- The project is broken into different activities systematically.
- Activities are arranged in logical sequence.
- The network diagram is drawn. Event and activities are numserverd.
- Using three times estimate, the expected time for each activity is calculated.
- Standard deviation and variance for each activity are computed.
- Earliest starting times and latest finishing times are calcaulted.
- Expected time, earliest starting time, and latest finishing times are marked on the network diagram.
- Slack is calculated.
- Critical path(s) are identified and marked on the network diagram.
- Length of critical path or total project duration is found out.
- Lastly, the probability that the project will finish at due date is calculated.

Estimation of activity time :

For dealing with uncertainties associated with different activities, PERT approach computes expected time for each activity form the following three time estimates :

(a) Optimistic time (t_o)

It is the shortest possible time in which an activity can be completed if everything goes exceptionally well.

(b) Most likely time (t_m) :

It is the time in which the activity is normally expected to complete under normal contingencies.

(c) Pessimistic time (t_p) :

It is the time in which the activity will take to compelte in case of difficulties i.e. if mostly the things go wrong. It is the longest of all the three time estimates.

The t_0 , t_m and t_p are combined statistically to develop the *expected time* (t_e) for an activity. The fundamental assumption in PERT is that the three times estimates form the end points and mode of *Beta distribution*.

It is further assumed that t_m and t_0 are about equally likely to occur where as the probability of occurrence of t_m is 4 times that of t_0 or t_p .

$$\text{Therefore, } t_e = \frac{t_0 + 4t_m + t_p}{6}$$

PERT using statistical probability concept, employs standard deviation and variance as measures of variability.

They are given by

$$\text{Standard deviation} = \left(\frac{t_p - t_0}{6} \right)$$

$$\text{Variance} = \left(\frac{t_p - t_0}{6} \right)^2$$

$$= (S_t)^2$$

Example - 4.5 A small consultancy project consist of following activities given below :

Activity	t_0	t_m	t_p
1-2	2	5	14
1-6	2	5	8
2-3	5	11	29
2-4	1	4	7
3-5	5	11	17
4-5	2	5	14
6-7	3	9	27
5-8	2	2	8
7-8	7	13	31

- Calculate the values of expected time (t_e), standard deviation, and variance for each activity.
- Draw network diagram and mark t_e .
- Calculate EST and LFT and mark them on diagram.
- Calculate total slack for each activity.
- Identify the critical path and Find length of the critical path.
- Find critical path(s).
- Calculate variance of critical path.
- Calculate the probability that job in critical path will be finished by the due date of 38 days.

- (i) Calculate approximate probability that jobs on the next most critical path will be completed by the due date of 38 days.
- (j) If the project due date changes to 35 days what is the probability of not meeting that due date.
- (k) Find the due date which has a probability 94.5% of being met.

Solution :

- (a) We know that :

$$t_e = \frac{(t_0 + 4t_m + t_p)}{6}$$

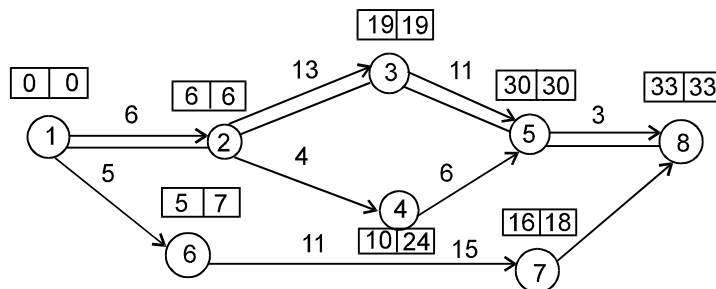
$$\text{Standard deviation} = \frac{t_p - t_0}{6}$$

$$\text{Variance} = \left(\frac{t_p - t_0}{6} \right)^2$$

$$= (\text{Standard deviation})^2$$

Activity	t_0	t_m	t_p	t_e	s_t	varr.
1-2	2	5	14	6	2	4
1-6	2	5	8	5	1	1
2-3	5	11	29	13	4	16
2-4	1	4	7	4	1	1
3-5	5	11	17	11	2	4
4-5	2	5	14	6	2	4
6-7	3	9	27	11	4	16
5-8	2	2	8	3	1	1
7-8	7	13	31	15	4	16

(b) and (c) follows figure.



(d)

Activity	EST	LST	Total slack (LST-EST)
1-2	0	0	0
1-6	0	2	2
2-3	6	6	0
2-4	6	20	14
3-5	19	19	0
4-5	10	24	14
6-7	5	7	2
5-8	30	30	0
7-8	16	18	2

(e) Critical path is 1-2-3-5-8 , Length is $6+13+11+3 = 33$ days.

(f) There are 3 paths 1-2-3-5-8 (33) critical path.

1-6-7-8 (31) Next most critical path.

1-2-4-5-8 (19) Next most critical path.

(g) The variance of critical path 1-2-3-5-8 is $4+16+4+1 = 25$.

(h) The probability that the project will meet the due date is calculated by the formula.

$$\begin{aligned}
 Z &= \frac{\text{Due date} - \text{Expected date of completion}}{\sqrt{\text{Variance}}} \\
 &= \frac{38 - 33}{\sqrt{4 + 16 + 4 + 1}} \\
 &= \frac{5}{5} = 1
 \end{aligned}$$

For the value of $Z=1$ the corresponding standard normal distribution table is 0.841 i.e., 84.1%.

(i) Next most critical path is 1-6-7-8 of 31 days.

Varriance of path = $1+16+16 = 33$

$$\begin{aligned}
 \therefore Z &= \frac{\text{Due date} - \text{Expected date of completion}}{\sqrt{\text{Variance}}} \\
 &= \frac{38 - 31}{\sqrt{33}} \\
 &= \frac{38 - 31}{5.74} \\
 &= 1.22
 \end{aligned}$$

$\therefore$ From the normal distribution table, $Z = 1.22$ is 0.888

The probability of meeting due date in next most critical path is 88.8%.

$$(j) \quad \text{Again} \quad Z = \frac{35 - 33}{5} \\ = 0.4$$

From the normal distribution table $Z=0.4$ is 0.655

The probability of not meeting the due date is $1-0.655 = 0.345$ i.e., 34.5%

$$(k) \quad Z = 94.5\% \text{ or } 0.945 \text{ the value of } Z = 1.6$$

$$\text{Therefore } 1.6 = \frac{\text{Due date} - 33}{5}$$

$$\therefore \text{Due date} = 41 \text{ days.}$$

Standard Normal Distribution Table

Z	Probability of meeting Due or scheduled date	Z	Probability of meeting Due or scheduled date
2.8	0.997	-0.2	0.421
2.6	0.995	-0.4	0.345
2.4	0.992	-0.6	0.274
2.2	0.989	-0.8	0.212
2.0	0.977	-1.0	0.159
1.8	0.964	-1.2	0.115
1.6	0.945	-1.4	0.081
1.4	0.919	-1.6	0.055
1.2	0.885	-1.8	0.036
1.0	0.841	-2.0	0.023
0.8	0.788	-2.2	0.014
0.6	0.726	-2.4	0.008
0.4	0.655	-2.6	0.005
0.2	0.579	-2.8	0.003
0.0	0.500		

Difference between PERT and CPM.

The fundamental network of PERT and CPM are though identical, yet there are (certain) differences in details as mentioned below (Table 4.1)

Table 4.1 : Comparison of CPM with PERT

	PERT	CPM
1.	A probabilistic model with uncertainty in activity duration expected time is calculated from t_0 , t_m and t_p .	A deterministic model with well-known activity (single times based upon past experience. It assumes that, the expected time is actually the time taken.
2	An event-oriented approach.	An activity-oriented system.
3	PERT terminology uses words like network diagram, events, and slack.	CPM terminology employs words like arrow diagram, nodes, and float.
4	The use of dummy activities is required for representing the proper sequencing.	The use of dummy activities is not necessary. The arrow diagram thus becomes slightly simpler.
5	PERT basically does not demarcate between critical and non critical activities.	CPM marks critical activities.
6	PERT finds applications in projects where resources (men, materials and specially money) are always made available as and when required.	CPM is employed to those projects where minimum overall costs is of primary importance. There is better utilization of resources.
7	Especially suitable in deference projects and R & D where activity time cannot be reliably predicted.	Suitable for problems is Industrial setting, plant maintenance, civil construction projects, etc.

Project Crashing :

Crash time is the minimum possible time in which the activity can be completed and the cost associated with this is crash cost.

Normal cost is the cost incurred when activity is completed in its normal routine time.

$$\text{Cost slope} = \frac{\text{Crash cost} - \text{Normal cost}}{\text{Normal time} - \text{Crash time}}$$

$$= \frac{\Delta C}{\Delta T}$$

Example - 4.6 :

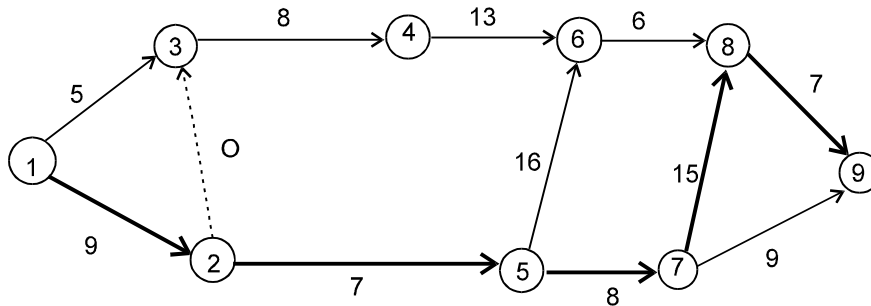
The network of PERT as shown in the table below contains normal and crash time for each activity along with normal cost and crash cost. Find optimum scheduled and optimum project duration.

Activity	NORMAL		CRASH		ΔT	ΔC	$\frac{\Delta C}{\Delta T}$
	Time	Cost	Time	Cost			
1-2	9	7000	4	10,000	5	3000	600
1-3	5	6000	3	8000	2	2000	1000
2-3	0	0	0	0	0	0	0
2-5	7	9000	2	11,500	5	2500	500
3-4	8	2500	6	3000	2	500	250
4-6	3	10,000	9	16,000	4	6000	1500
5-6	16	12,000	11	16,000	5	4000	800
5-7	8	12,000	7	14,000	1	2000	2000
6-8	6	10,000	6	10,000	0	0	---
7-8	15	6,000	8	7400	7	1400	200
7-9	9	6,000	6	12,000	3	6000	2000
8-9	7	6,000	5	7800	2	1800	900
Total – 86,500		1,15,700					

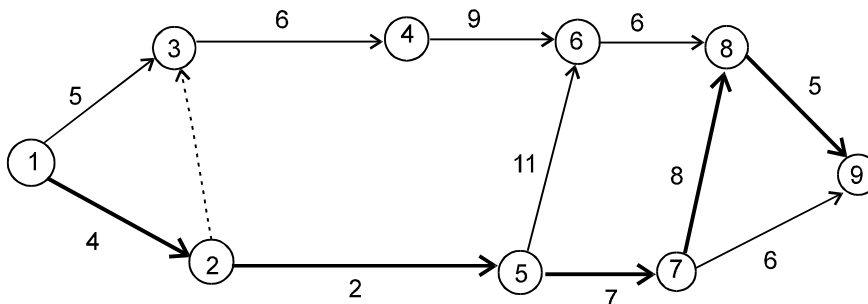
The indirect cost given as Rs. 1500 per week.

Solution :

Remark :



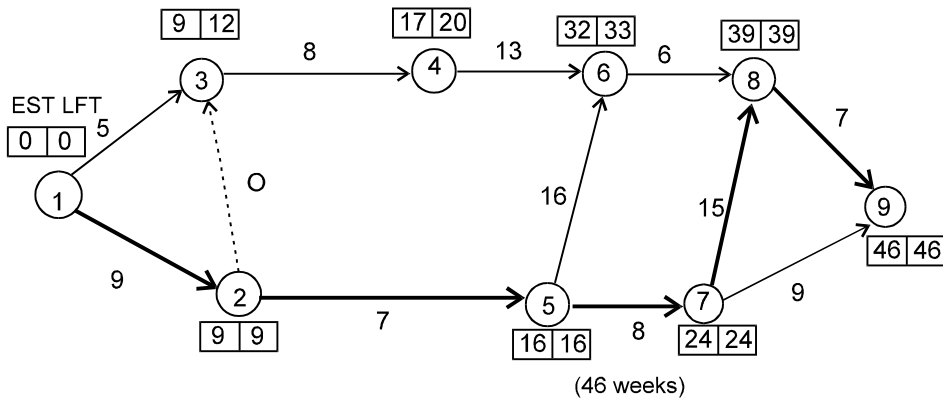
Normal project duration (critical path) is 1-2-5-7-8-9 = 46 weeks.



Crash time project duration is = 30 week.

Now optimum length schedule will be lie some where between 46 and 30 weeks

Now start from the diagram with normal expected time.



$$\text{Cost slope} = \frac{\Delta C}{\Delta T} = \frac{\text{Crash cost} - \text{Normal cost}}{\text{Normal time} - \text{Crash time}}$$

The critical path is 1-2-5-7-8-9 = 46 weeks.

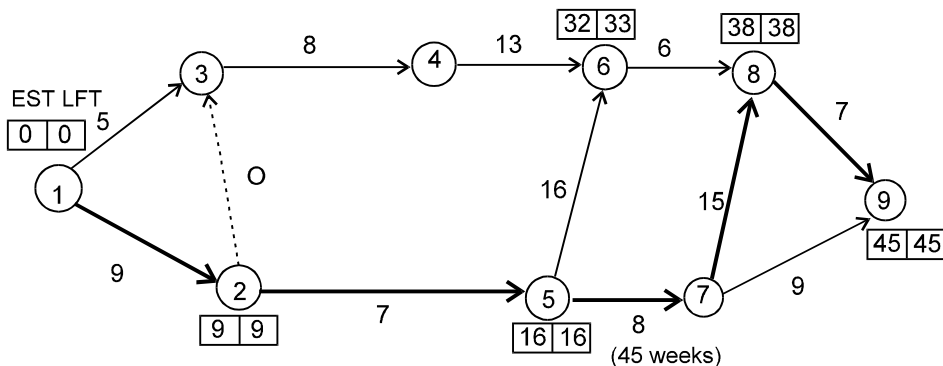
Hence total project cost = Direct cost of all activities + Indirect cost for 46 weeks.

$$= 86,500 + 46 \times 1500 = 1,55,500$$

Now we are to identify those activities on the critical path which can be crashed (only those critical activities with cost slope less than or equal to the indirect cost are crashed in case an activity with cost slope greater than indirect cost is crashed, the saving in indirect cost will be less than the cost of crashing).

The activity with least expenses or with the least cost slope will be chosen first.

Activity 7-8 with $\frac{\Delta C}{\Delta T} = 200$ is the least expensive and can be crashed by 7 weeks at an additional cost of Rs. 1400 ($= 7 \times 200$). The new critical path is 1-2-5-6-8-9.



Thus by crashing 7-8 by 7 weeks the project's time period of completion has come down from 46 to 45 days, i.e., reduced by 1 day.

$$\text{Direct project cost} = 86,500 + 7 \times 200 = 87,900$$

$$\text{Indirect cost} = 45 \times 1500 = 67,500$$

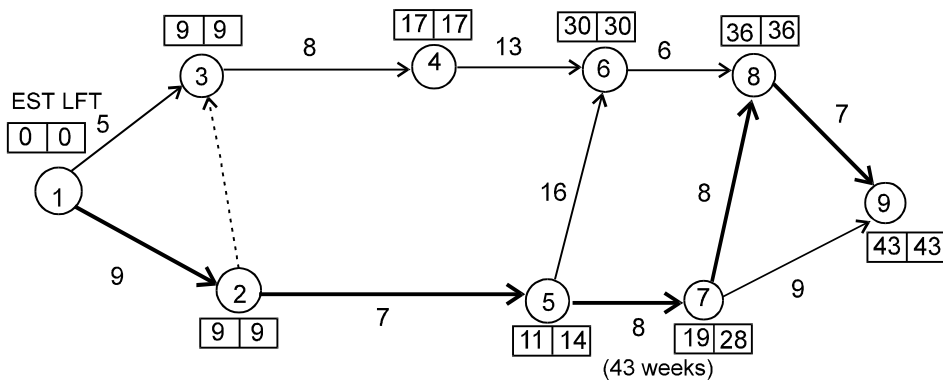
$$\text{Total project cost} = 1,55,400$$

On the critical path 1-2-5-6-8-9 activity with the least cost slope is 2-5 with cost slope of Rs. 500. It is crashed by 5 weeks. The critical path now becomes 1-2-3-4-6-8-9 = 43 weeks.

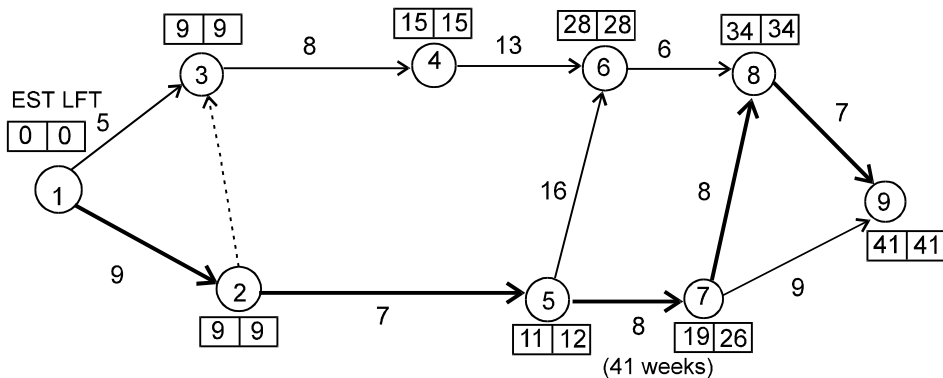
$$\text{For this path direct cost} = 87,900 + 500 = 90,400$$

$$\text{Indirect cost} = 43 \times 1500 = 64,500$$

$$\text{Total cost} = 1,54,900$$



On the critical path 1-2-3-4-6-8-9 the activity with the least cost slope is 3-4 = 250. This activity can be crashed by 2 weeks. After crashing the new network is shown below:

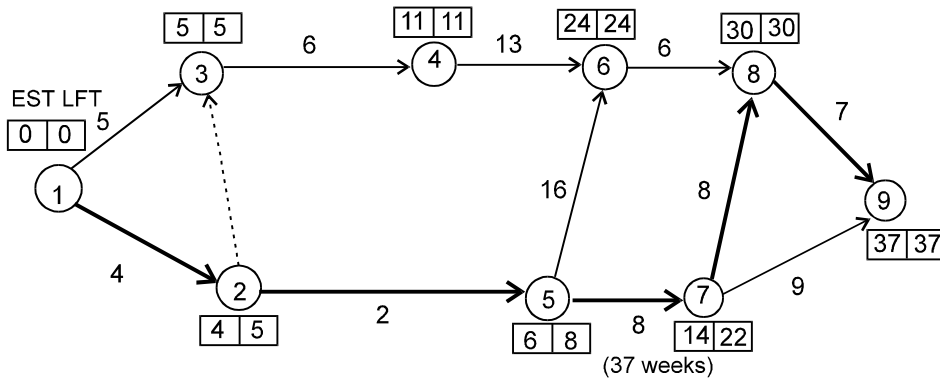


$$\text{Direct cost} = 90,400 + 2 \times 250 = 90,900$$

$$\text{Indirect cost} = 41 \times 1500 = 61,500$$

$$\text{Total cost} = 1,52,400$$

Critical path is the same 1-2-3-4-6-8-9 = 41 weeks. On this path the least expensive activity is 1-2, i.e., cost slope is 600 and it can be crashed by 5 weeks. This change brings the critical path is 1-3-4-6-8-9 = 37 weeks.



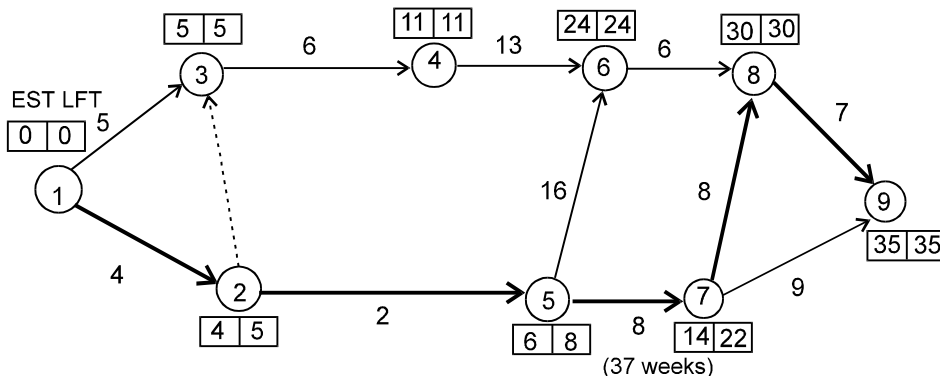
For this path

Direct cost = 90900 + 5 × 600 = 93900

Indirect cost $= 37 \times 1500 = 55500$

Total cost = 1,49,400

Next activity to be crashed on 1-3-4-6-8-9 is 8-9 for 2 weeks at a cost of 900 (Rs) per week. This change makes the new project duration of 35 weeks but the critical path remains the same.

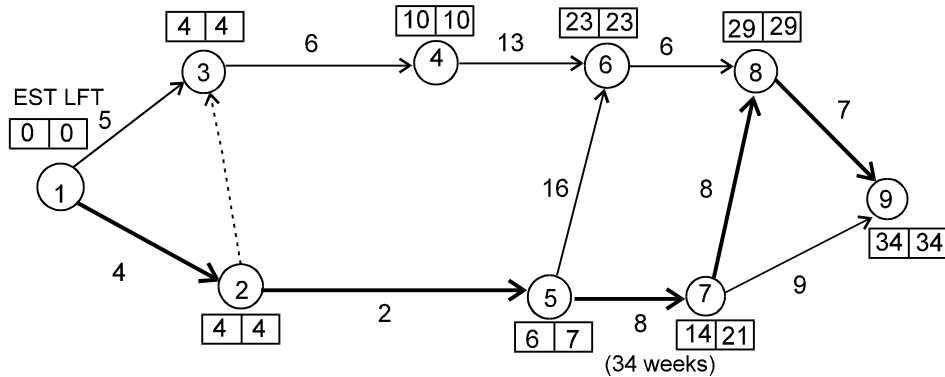


Direct cost $= 93900 + 2 \times 900 = 95700$

Indirect cost $= 35 \times 1500 = 52500$

Total cost = 1,48,200

Now on the path 1-3-4-6-8-9 is activity 1-3 with a cost slope of 1000 is the minimum expensive and this activity can be crashed for 2 weeks. Now the new critical ; path is 1-2-3-4-6-8-9 = 34 weeks.

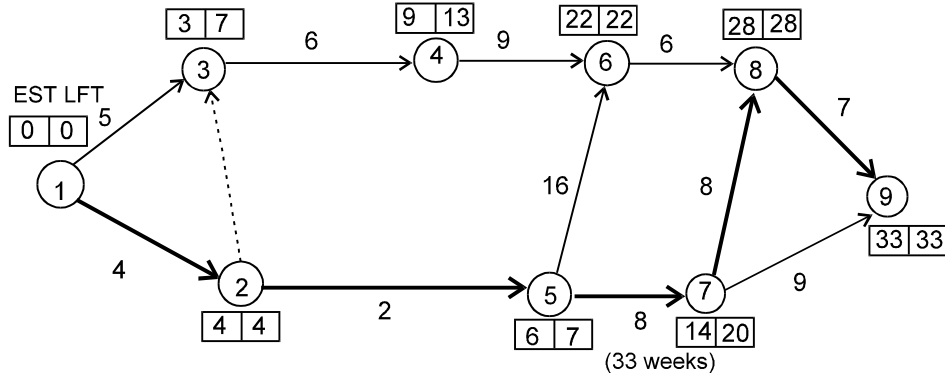


$$\text{Direct cost} = 95700 + 2 \times 1000 = 9,7700$$

$$\text{Indirect cost} = 34 \times 1500 = 51,000$$

$$\text{Total cost} = 1,48,700$$

On this critical path there is only one activity 4-6 which is not crashed. Crashing, it by 4 weeks at a cost of Rs. $4 \times 1500 = 6000$ we get ;the new critical path 1-2-3-4-6-8-9 = 33 weeks.

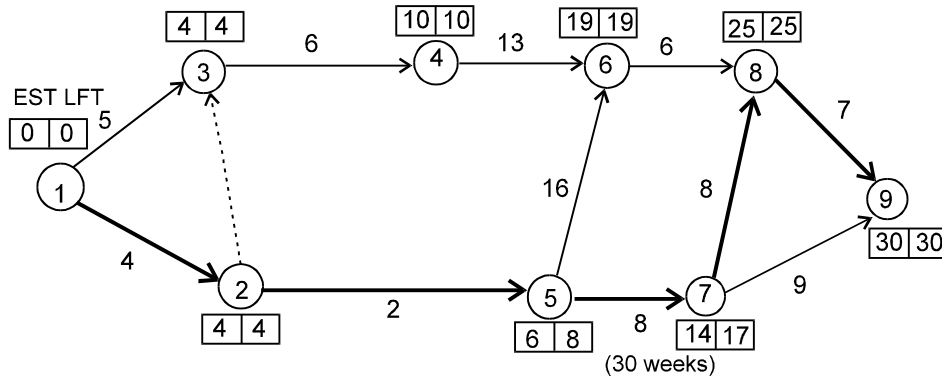


$$\text{Direct cost} = 97700 + 4 \times 1500 = 1,03,700$$

$$\text{Indirect cost} = 33 \times 1500 = 49500$$

$$\text{Total cost} = 1,53,200$$

On the critical path 1-2-5-6-8-9 activity 5-6 is uncrashed. It can be crashed for 5 weeks with a cost slope of 800 per weeks. This crashing gives the new critical path of 1-2-3-4-6-8-9 = 30 weeks.



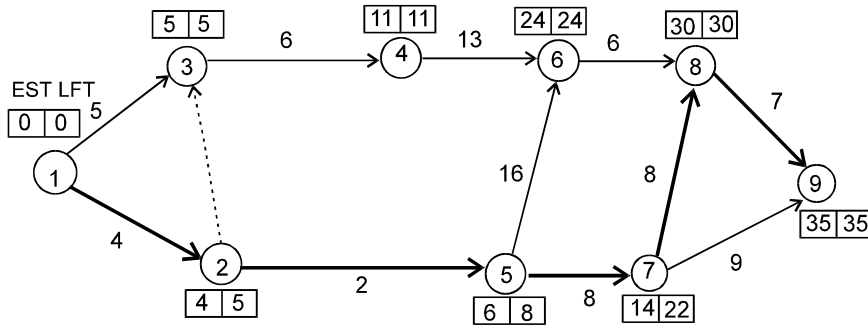
$$\begin{aligned}
 \text{Direct cost} &= 1,03,700 + 5 \times 800 \\
 \text{Indirect cost} &= 33 \times 1500 = 45000 \\
 \text{Total cost} &= 1,52,700
 \end{aligned}$$

We see that all the activities on the critical path which could be crashed as per the given condition, have been crashed and it is not possible to further crash the project network. It is also to be noted that crashing of the non-critical activities will not change the project completion time and hence they will not change the project completion time and hence they will not be crashed. Now a summary is prepared below.

SUMMARY

<i>Activity</i>	<i>Weeks</i>	<i>Project</i>	<i>Direct</i>	<i>Indirect</i>	<i>Total</i>
<i>Crashed</i>	<i>Saved</i>	<i>Duration (weeks)</i>	<i>Cost</i>	<i>Cost</i>	<i>Cost</i>
NIL	0	4	86500	69,600	1,55,000
7-8	1	45	87900	67,500	1,55,400
2-5	2	43	90600	64,500	1,54,900
3-4	2	41	90900	61,500	1,52,400
1-2	4	37	93900	55,500	1,49,400
8-9	2	35	95700	52,500	1,48,200
					(Least)
1-3	1	34	97700	51,000	1,48,700
4-6	1	33	103700	49,500	1,53,200
5-6	3	30	107700	45,000	1,52,700

From the summary table, we observe that the least total cost is Rs. 14.8,200 corresponding to a project duration of 35 weeks.



4.8.2.4 GANTT Charts

We have already discussed some highlighted techniques of project scheduling like critical path method (CPM) and project evaluation and review techniques (PERT)

Let's discuss, an important technique to keep the information for

- What is happening ?
- When is happening ?
- Where it is happening and
- Who are there ?

concerned with a project in a graphical form [horizontal bar chart] is called Gantt chart. Basically it provides a graphical illustration of a schedule that helps to plan, co-ordinate, and track specific tasks in a project. It shows the relationship of activities w.r.t. time.

Gantt chart was developed as a production tool in the year 1917 by Henry L. Gantt, an Americal engineer and social scientist. It indicates different activities, responsibilities, estimated or planned activities with their dates (Actual start and target dates)

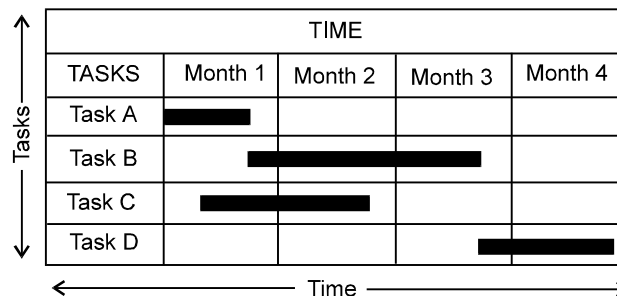
It plots activities as bars on a time scale.

Gantt chart helps in scheduling activities, co-ordinating the activities, evaluating the progress, allocating the resources etc.

It provides an easy understandable overview of a project for those without having any technical background. It also, reflects the status of each project at any instant of time, represents overlapping of activities or parallel tasks.

Examples of Gantt charts :

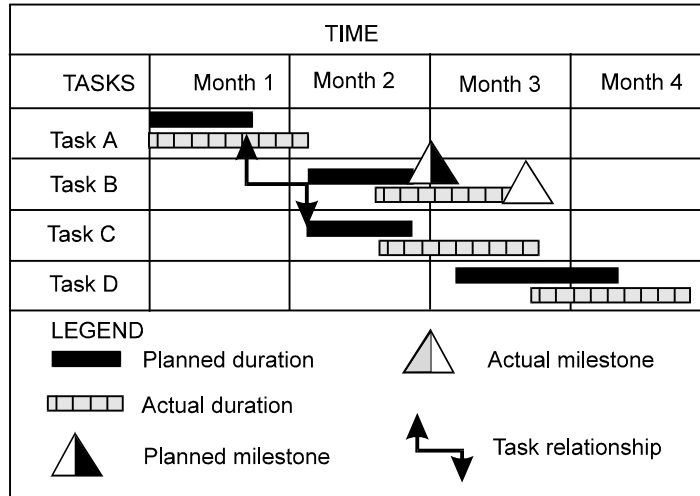
Example - 4.7 : A simple chart : It can be shown as follows :



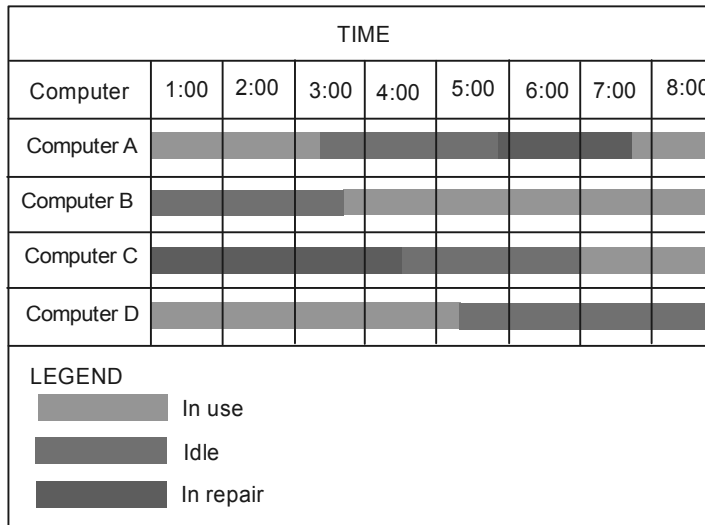
Simple Gantt Chart

Example-4.8 :

Typically, Gantt charts indicate the exact duration of specific tasks, but they can also be used to indicate the relationship between tasks, planned and actual completion dates, cost of each task, the person or persons responsible for each task, and the milestones in a project's development. Such a Gantt Chart is shown in following figure.

*Gantt Chart***Example - 4.9 :**

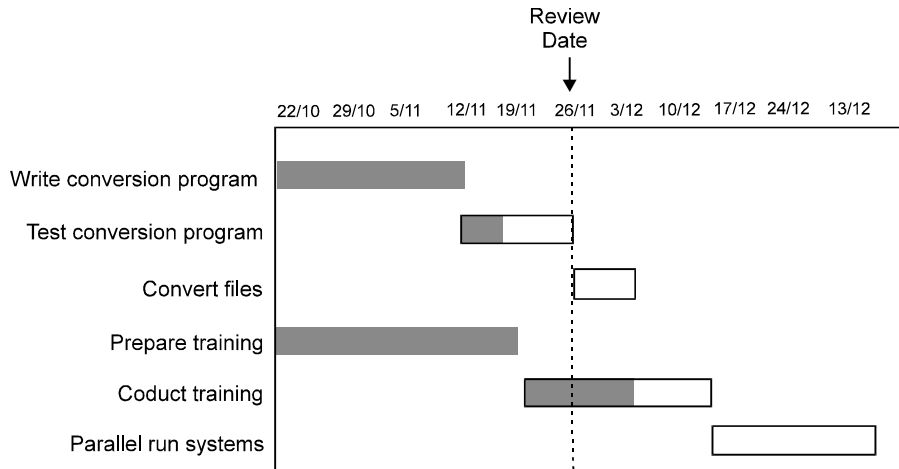
Gantt charts are also used to indicate the usage statistics of computers in a software development organisation. Such a chart is shown in the following figure.

*Gantt Chart***Example - 4.10 :**

There are no widely accepted standards for Gantt charts. Automated tools are available which produce Gantt charts directly from activity networks or from a full

description of the tasks. Automated tools have features which assist the planning function including

- Display of original and latest time for task;

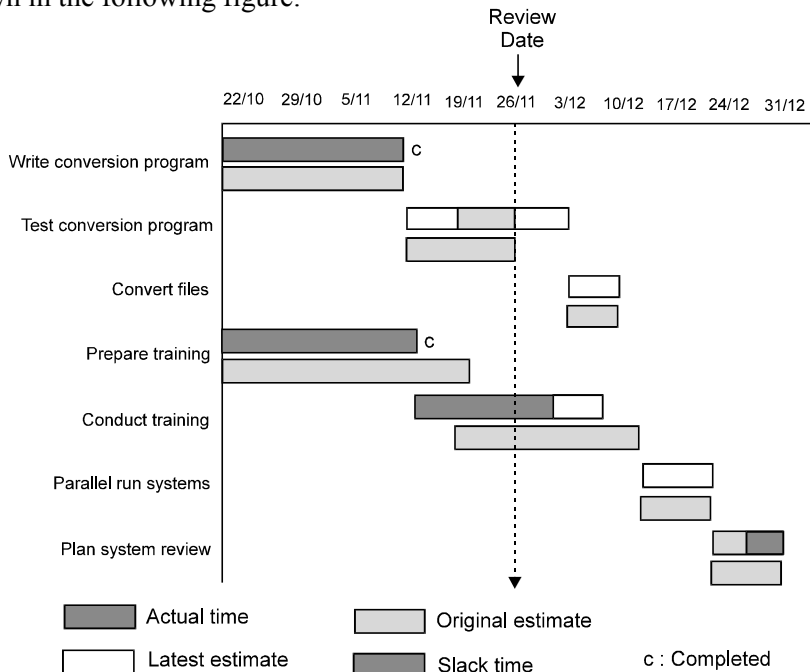


Example of a Gantt Chart

- Display of person(s) allocated to tasks;
- Integration with other planning techniques (i.e. networks and milestones).

A Gantt Chart generated by automated tool is shown in the above figure.

Alternative Gantt Chart incorporating features commonly present in automated tools is shown in the following figure.



Example of a Gantt Chart showing Project Management Tool Features

(This chart shows which tasks are ahead of schedule and which tasks are behind schedule?)

Reasons to Use Gantt Charts

Gantt Charts are convenient in the means of representing graphically schedules, appointments and events. Such charts help members of an organization to plan activities better.

Information about *WHAT* is happening, *WHEN* is happening, *WHERE* is happening, and *WHO* will be present, is valuable to concerned people in the organization.

It is good because:

- It plots activities as bars on a time scale.
- It indicates different:
 - (a) *activities*
 - (b) *responsibilities*
 - (c) *estimated or planned activities' start and target dates*
 - (d) *actual start and target dates*
- It aids in:
 - (a) *scheduling activities*
 - (b) *coordinating activities*
 - (c) *evaluating progress*
 - (d) *reallocating resources in case of problems*

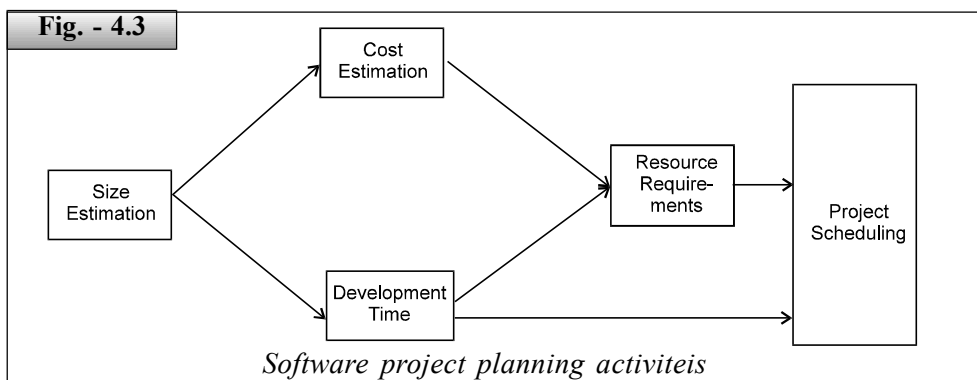
Advantages of Gantt Charts

- ☛ Provide an easily understandable overview of a project for those without any technical background.
- ☛ Gantt charts are relatively easy to create and maintain.
- ☛ Able to reflect the status of each project task at any point in time
- ☛ Able to represent overlapping or parallel tasks

Disadvantages of Gantt Charts

- ☛ Does not show a task interrelationship.
- ☛ These are not ideal for project control.

4.9 SOFTWARE PROJECT ESTIMATION



Software project management starts with a set of activities, collectively called as project planning. Before any project can begin, the project manager, software project team must have to have knowledge about the type of works to be done, amount of resources to be utilised throughout the project (to be conducted) and amount of time (elapsed) that will be required to finish the project.

(elapsed time : time duration from start to finish a project)

So the term estimation can be defined in general that “*an attempt to determine (approximation) how much money, effort, resources and time it will take to complete a task (may be a building, any cultural program, a product or a software- based system or product)*”

Unless we know about the detail requirements (time, money, effort resources etc.), it is meaningless to start a project.

The real question here is the term “approximate” in the second paragraph.

How can we rely on it ?

It has been reported (*BEN 03*) that 40 percent of software developers, they struggle with estimation and that software size and development time are very difficult to estimate accurately.

Yes, its really a hard job, because we really want to know the actual requirements (estimation) until the project has been completed. But, we can have a better and close picture about estimation of a project, through experienced persons (project managers), using systematic approaches, taking solid historical data into consideration and implementing some standard methods of estimation.

Here in case of software project estimation we generally take three things into consideration i.e. -

*cost
effort and
duration.*

4.9.1 Software metrics in project estimation

The term metric(s) is used to measure the different aspects of any product (in a quantitative term) that need evaluation on an ongoing basis for estimation. Metrics is associated with two other different terms.

* measure * measurement.

☛ *Measure is an attribute of an entity.*

[for example, Length, breadth and height are the measures of a room]

☛ *Measurement is an activity to determine the measure.*

[Ex : Using an instrument say tape, to measure the length / breadth / height of a room]

- Now, we can say, metrics are built using the measures.

e.g : Size and area of a room are expressed using the units of measurement (e.g : foot, metre, inch etc.)

* Metrics are independent of the units of measure and measurement.

All engineering disciplines have also metrics (such as metrics for weight, density, wave length, pressure, temperature) to quantify various characteristics of their products.

Likewise, software engineering professionals have evolved different metrics (associated with measures, measurement) to describe software and its related attributes.

So, the question arises, what is software metrics ?

Software metrics are built out of several measures, and provide the software developers and insight into the various aspects of software product : namely

- Software processes.
- Software products and so on....

Software metrics are useful to build standards for planning and control, quality productivity and estimation of resources, cost, efforts, software size and time for software development.

The metrics differ from one type of software to other. The metrics of business software is different from that of engineering and scientific softwares.

Software metrics have two categories based on the types of measures used in :

- * *Direct measure* : Cost, effort, LOCs, response time, resources and so on.
- * *Indirect measure* : Functionality, quality, complexity, efficiency, dependability, maintainability, ease of use, quality of documentation etc.

4.9.2 Types of software metrics

There are basically three different software metrics, which are as follows :

1. *Product metrics* : This type of software metrics describe the characteristics of the product such as size, complexity, design features, efficiency, reliability, performance etc. product metrics are used to build the artifacts of the software system requirement specification documents, system design document etc.
2. *Process metrics* : This type describes the effectiveness and quality of the processes that produce the software products.
e.g : effort required in a process, time to produce the product, effectiveness of defect removal during development, no. of defects formed during testing, process maturity etc.
3. *Project metrics* : Project metrics describes the project characteristics and execution.
i.e. number of developers,
staffing policy,
cost and schedule, and
productivity etc.

This metric is used by the project manager to control the project in terms of cost, time and effort through measurement of skills, customer relations, technology of development, software solution design etc.

4.9.3 Qualities of software metrics

In general, software metrics should possess the following qualities.

- *Simple and definable* :
It should be clear enough how the metrics are to be calculated.

- *Easily obtainable:*
The metrics should be available whenever required at a reasonable cost.
- *Valid :*
The metrics should be effective to what they are meant for.
- *Robustness :*
The metrics should not be sensitive to any minor product changes.
- *Programming Language independent :*
The metrics should be based as the analysis model, the design model or structure of the program itself, but not the programming environment.
- *High-quality feedback :*
The metrics should lead to a higher-quality and product.

4.9.4 Product metrics

Generally product metrics are the measurements taken from an actual body of the source code (i.e. the size of a software product)

These are viewed as either w.r.t. the internal or external attributes of a software product.

i.e. *Internal* : Purely in terms of the product itself.

External : In terms of the behaviour of the product to its environment.

Categories of product metrics

There metrics can be categorized as

1. Size metrics [Software or program size]
 - *Lines of codes (LOCs)*
 - *Function Points (FPs)*
 - *Feature Points (FPs)*

4.9.4.1 Lines of codes (LOCs)

This metric is the simplest one to measure the product size in number of lines of codes. (LOCs lines of codes or KLOCs (Kilo/thousand lines of codes), or KDIC (Kilo Delivered Instruction Code).

Basically, in this metric the lines of the source codes (high-level language programs) are counted and the length of the program (product) is calculated.

A technique called, modularisation is used to break a large program into smaller modules (concrete and meaningful). So that the length of each module can be easily determined, where the past experience on developing software products are also taken into consideration.

Limitations :

- It is purely defined on codes and characterised only on the size (i.e. length) of the program / product.
- It does not entertain the functionality or complexity of the product.
- It is language dependent as the lines of codes varies from language to language (no of lines under 'C' language differ from using COBOL language)

- It does not measure the size of a product specification.
Using *Lines of codes (LOCs)* technique can also be compared with the following examples.
- while purchasing a TV set from a vendor, how many of us really ask about the number of transistors, IC's, resistors and other components what the TV set is made up of instead of its external attributes / behaviour like Screen size, Look, Picture quality, sound system and other facilities (functionalities) available with the TV set itself?
- To build of a house, we concentrate on the no of rooms as well as the room size not the no. of bricks used for the rooms.

4.9.4.2 Function Points (FPs)

It is a popular method for estimating the size of a software product w.r.t. its "functionality.

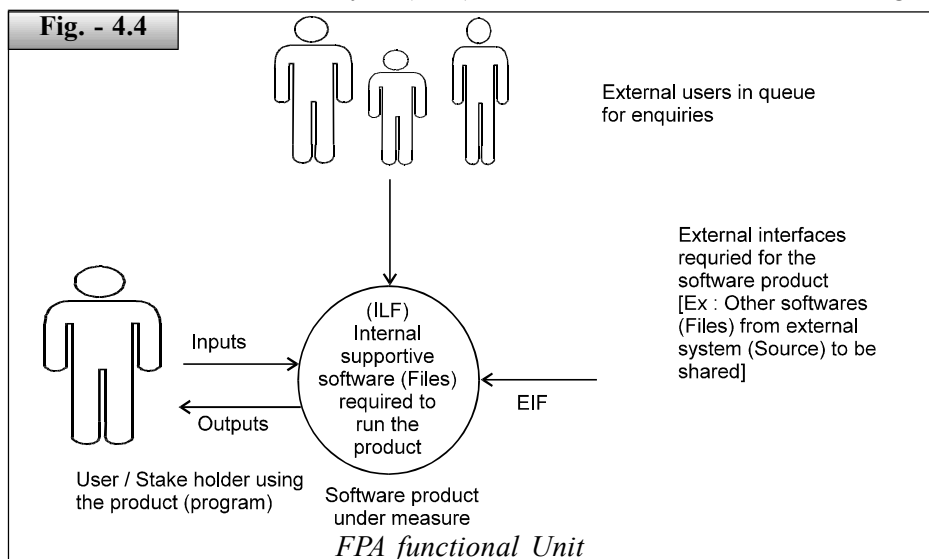
Now it is clear, how much we look for a product (Ex. A TV set) w.r.t. its use (as per our requirement), not the components being used to manufacture it.

Function point metric was proposed by *Allan Albrecht (1979)*. It uses a measure of the functionality exhibited by a software product as a major value from the user's point of view i.e. on the basis of what the user requests and receives in return.

Function point (FP) approach is to identify and count the number of unique function types. like

- *External inputs (File names, data etc.)*
- *External output (Reports, message etc.)*
- *Enquiries (interactive inputs)*
- *Internal logical fiels (Invisible to internal to the system)*
- *External interface files.*

The Function Point Analysis (FPA) functional units are shown in the fig. 4.4.



Advantages :

- Function point approach is independent of language environment, tools, or methodologies i.e. it does not depend on programming language, DBMs, other supporting hardware.
- Function points can be estimated from requirement specification or design specifications, which enables to estimate the development effort at the early stage of development.
- Function points are purely based on the user's requirements, any change of requirements can easily be followed by a re-estimate.
- User's of all range (technical / non-technical) can interact and understand the software product w.r.t its functional measure.
- It resolves many inconsistencies that arise while using lines of codes (LOCs) as a product size measure.

Function Points (FPs) calculation :

The following facts, figure and policies are taken into consideration for calculating function points. (Shown in the table-4.2)

- All(five) functional units are ranked according to their complexity (degree of use). i.e. low, average or high, using a set of standard (predefined)
- The weighting value (based on the complexity) that varies from 3(for simple external inputs) to 15(for complex internal files).

Table -4.2 : Functional units with weighting factors

Functional Units	Weighting Factor		
	Low	Average	High
External inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Enquiries (EI)	3	4	6
External Logical Files (ILF)	7	10	15
External Interface Files (EIF)	5	7	10

- The Un-adjusted Function Points (UFP) i.e. the arbitrarily calculated function points, is completed by multiplying each function unit count with their weighting factor and adding of all the values i.e.

$$UFP = \sum_{i=1}^5 \cdot \sum_{j=1}^3 Z_{ij} \cdot W_{ij}$$

- where
- i = Functional units varies from 1 – 5
 - j = Weighting factor varies from 1 – 3 (Low, average and high unit)
 - z = no. of functional unit counts.
 - w = weighting factor values for functional units.

- A complexity adjustment factor (CAF) or Technical complexity factor (TCF) is calculated as

$$\text{CAF or TCF} = 0.65 + (\text{sum of factor}) / 100$$

$$= (0.65 + 0.01 \times \sum_{i=1}^{14} F_i)$$

where, F_i ($i = 1$ to 14) is the degree of influence and are based on the response to the questions (1 – 14) as shown in the table 4.3.

Table-4.3 : Influence factor and the response

Rate each Factor on a scale 0 to 5					
0	1	2	3	4	5
No influence	Incidental	Moderate	Average	Significant	Essential
Number of factors considered (F_i) Total = 14.					
1	Does the system require reliable backup and recovery ?				
2	Is data communication required ?				
3	Are there distributed processing functions ?				
4	Is performance critical ?				
5	Will the system run in an existing heavily utilized operational environment ?				
6	Does the system need on-line data entry ?				
7	Does the on line data entry require the input transaction to be built over multiple screens or operations ?				
8	Are the master files updated on line ?				
9	Is the inputs, outputs, files or inquires complex ?				
10	Is the internal processing complex ?				
11	Is the code designed to be reusable ?				
12	Are conversion and installation included in the design ?				
13	Is the system designed for multiple installations in different organizations ?				
14	Is the application designed to facilitate change and ease of use by the user ?				

On the basis of function points, several other metrics can also be computed.

- * Productivity = Function points / Person - month
- * Quality = No. of defects / Function points.
- * Documentation = No. of pages / Function points.

Example - 4.11 : Consider a project with the following functional units.

No. of user inputs	= 50
No. of user outputs	= 40
No. of user enquires	= 35
No. of user files	= 06
No. of Ext. Interface	= 50

Assume the complexity adjustment factors and weighting factors are average.
Calculate FP for the project.

Solution : We know $UFP = \sum_{i=1}^5 \cdot \sum_{j=1}^3 Z_{ij}, W_{ij}$

$$\Rightarrow UFP = 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7 \\ = 200 + 200 + 140 + 60 + 28 \\ = 628$$

$$CAF = (0.65 + 0.01 \sum Fi)$$

or

$$TCF = 0.65 + 0.01 (14 \times 3) \\ = 0.65 + 0.42 = 1.07$$

$$\text{So, } FP = UFP \times CAF \\ = 628 \times 1.07 = 672 \text{ (Ans)}$$

Example - 4.12

Calculate the FP value for a software product with following characteristics.

No. of user inputs : 32

No. of user outputs : 60

No. of user enquiries : 24

No. of Internal files : 8

No. of external interface : 2

With the complexity adjustment values (average for all)

Solution : Let's calculate the UFP

$$UFP = \text{Sum of the product of all functional units and their weighting factor} \\ = 32 \times 4 + 60 \times 5 + 24 \times 4 + 8 \times 10 + 2 \times 7 = 618$$

where the weighting factor for all the functional units is average.

(i.e. input - 4, output - 5, enquires - 4)

Files - 10, External Interface - 7

now Function Point (FP)

$$= UFP \times TCF \text{ or } CAF \\ = 618 \times (0.65 + 0.01 \times \sum fi) \\ = 618 \times (0.65 + 0.01 \times 30) = 587.10$$

Example - 4.13

An application has the following :

inputs = 10 (Low)

outputs = 12 (high)

Int. Logical Files = 20 (low)

Ext. Interface = 15 (high)

enquiries = 12 (avg.)

and

the CAF or TCF = 1.10

Findout UFP and FP for the application.

Solution :

$$\begin{aligned}
 \text{here UFP} &= \sum_{i=1}^5 \sum_{j=1}^3 Z_{ij} W_{ij} \\
 &= 10 \times 3 + 12 \times 7 + 20 \times 7 + 15 \times 10 \\
 &\quad + 12 \times 4 \\
 &= 452 \\
 \text{FP} &= \text{UFP} \times \text{CAF} \\
 &= 452 \times 1.10 = 497.2
 \end{aligned}$$

4.9.4.3 Feature Point metrics

The extension of function point metric is called *Feature point metric*. In function point metrics, it is assumed that the effort required to design and develop any two functionalities of the system is same. but it differs, as there are different levels (easy / moderate / difficult) of developing functionalities.

This is overcome by taking the algorithmic complexity of the software product as an extra parameter into consideration in Feature point metric. The size of a product using this metric indicates that the functionalities having more complexity will need more effort i.e. the size of a product is larger compared to simpler functionalities.

As the FP is language independent, it can be easily calculated from the SRS during project planning.

4.9.5 Software project estimation techniques

While handling a software project (what ultimately results with software products) by the developer group, it is most important to estimate various project parameters those are essential in project planning and scheduling activities.

The important project parameters like the project size, effort required to conduct the project, time duration and also the cost of the project. There are basically three different policies for project estimation.

- Empirical estimation techniques.
- Heuristic techniques.
- Analytical estimation techniques.

1. Empirical estimation techniques :

These is one of the oldest techniques, and used to estimate the cost of large scale software projects. These techniques are related to a proposed project to similar projects for which historical cost data are already available.

The empirical data that support estimating a software project are generally derived from a limited sample of projects for this reason, empirical techniques are not appropriate for all classes of software and in all development environments. So the results obtained from these techniques can be taken as a predicated results.

It can be estimated as

$$E = A + B \times (e_v) C$$

where A, B, C are empirically derived constants,

E is effort in person - months and e_v is the estimation variable either in LOC or FP. There are two important empirical estimation techniques such as.

- **Expert Judgement Techniques :**

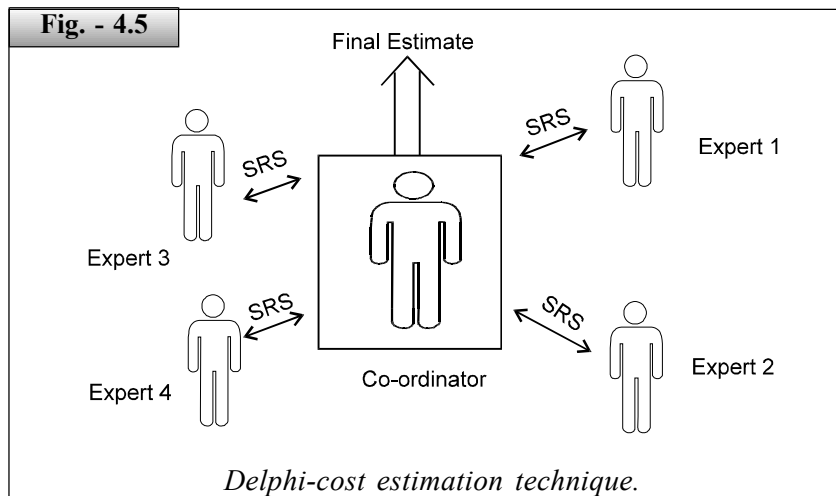
This approach is a guess work to estimate the project size after through analysis of the project (Problem). It is done by an expert, who estimates the cost of the different components of the project and combine them to get overall estimate.

But practically this technique is not error free rather contains human error, because of the limited experience and knowledge domain of the expert.

However, the human error can be minimised by entertaining a group of experts during estimation.

- **Delphi cost Estimation :**

It is an approach to minimize the short comings of the expert judgement committee approach. This approach is basically carried out by a team consists of a group of experts and a co-ordinator.



- The co-ordinator provides each expert a copy of the SRS.
- All experts individually estimate on SRS (No communication is entertained among the experts)
- Co-ordinator collects and summerizes all the estimates, if is required, the summerised estimate copies are again provided to the experts for further estimation and so on. This iterative process continues till to get the Final estimate (after compilation) on which the co-ordinator can rely.

2. Heuristic estimation techniques

These estimation techniques are used to find out the relationship among different project parameters (independent and dependent) through mathematical expressions.

Independent parameter are the basic parameters e.g. size and the dependent parameters are to be derived through mathematical expression e.g. effort, project duration, resources etc.

Heuristic estimation techniques uses two diff. classes i.e.

- Single variable model
- Multivariable model.

For single variable

$$\text{Estimated paramter} = C_1 * e^{d_1}$$

where C_1 , d_1 are constants (historical data) and 'e' is the characteristics of the software product (independent) has previously estimated.

For multi-variable,

Estimated Resource = $C_1 * e^{d_1} + C_2 * e^{d_2} + \dots$ where C_1 , d_1 , C_2 , d_2 are constants (historical data) and e_1 , e_2 are basic characteristics of the software product previously calculated.

COCOMO : The Constructive Cost Model

The cost estimation task for any project is mostly done through algorithmic cost modelling, where mathematical formulae are used.

COCOMO, the constructive cost model has gained popularity in the field of project cost estimation, was proposed by *Boehm, in the year 1981*. It is also called as COCOMO'81 model. This model is derived from the analysis of 63 different complete projects.

The following facts are considered under this cost estimation model.

- Any software development project is classified into :
- *Organic* [Application programs]
- or
- *Semi-detached* [Utility programs]
- or
- *Embedded* [System programs]

based on its development complexity.

- The software product is considered w.r.t its characteristics, the development team and the environment.
- Application oriented programs are like MS-Office, and other data processing program where as utility programs are compiler linkers, loaders and the operating system and real time system are system programs.
- The product development complexity for the above three classes can be mathematically shown in a ratio of 1:3:9 Respectively. [Embedded, Organic, Semi-detached].

Organic Project :

This type of project deals with the development of application programs, where a small team of project development with experienced team members (already developed similar projects earlier)

Semi-detached project :

A project can be considered as semi-detached, where a mixture of experienced and in-experienced team members handling the project to develop utility programs.

Embedded project:

In this case, experienced team members are needed, to develop system based software products those are heavily coupled with typical hardware.

Levels of COCOMO

Boehm has proposed three different levels for constructive cost model, they are :

- * *Basic COCOMO.*
- * *Intermediate COCOMO*
- * *Detailed COCOMO.*

Table 4.4 : Comparison Table for COCOMO modes

Mode	Project size	Nature of the Project	Innovation	Deadline of the project	Development Environment
Organic	2 – 50 KLOC	Small size project, experienced developers in the familiar environment, Ex – Pay roll, inventory projects	Little	No tight	Familiar and in-house
Semi-detached	50-300 KLOC	Moderate size project, average size team, average pass experience on similar projects. Ex – Utility program, compilers, database systems.	Medium	Medium	Medium
Embedded	>300 KLOC	Large project, Real time systems, complex interfaces, very less past experience Ex : ATM, Air Traffic Control software	Significant	Tight	Complex hardware/ customer interface

Basic COCOMO

It is a single valued static model that calculates software development effort and cost as a function of the program size expressed in estimated lines of codes (LOCs) It can be mathematically expressed.

$$E = a_b (KLOC)^{b_b} \quad \text{person-month}$$

$$D = C_b (E)^{d_b}$$

where, E is the effort applied in person - months.

D is the development time in months.

and a_b , b_b , c_b and d_b are the coefficient constants based as the category of the project.

Ex : $(a)_b \rightarrow$ Coefficient
 $\rightarrow$ Basic COCOMO

Table-4.5 : Basic COCOMO Co-efficients.

Project Category	Co-efficient			
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

When effort and development time are known, the average staff size to complete the project can be calculated as.

$$\text{Avg. staff size (SS)} = \frac{E}{D} \text{ Persons}$$

and Productivity level (P)

$$= \frac{KLOC}{E} \text{ KLOC / Person - month}$$

Example - 4.14

A project is estimated to 400 KLOC. Find out the effort (E) and development time (D) for organic class.

Solution :

We know the mathematical expression for effort $(E) = a_b (KLOC)^{b_b}_{(PM)}$

$$\text{dev. time (D)} = C_b (E)^{d_b}_{(Month)}$$

here the size in KLOC (given) = 400 referencing to the Basic co-efficient table, the coefficients for the organic project class.

$$\text{i.e. } a_b = 2.4 \quad b_b = 1.05 : c_b = 2.5 \text{ and } d_b = 0.38$$

$$\text{therefore } E = 2.4 \times (400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5 \times (1295.31)^{0.38} = 38.07 \text{ M}$$

Example - 4.15

A project size of 200 KLOC is to be developed. software development team has average experience on similar type of projects. The project schedule is not very tight.

Calculate the effort, development time, average staff size and productivity of the project.

Solution :

Based on the development team experience, project schedule and the project size (200 KLOC)

The project is of semi-detached class, hence, the coefficients.

a_b , b_b , c_b and d_b are 3.0, 1.12, 2.5 and 0.35 respectively (as referenced from the table)

So, Effort(E) = $a_b \times (\text{KLOC})^{b_b}$

$$= 3.0 \times (200)^{1.12}$$

$$= 1133.12 \text{ person – Month}$$

development and time (D) = $C_b (E)^{d_b}$

$$= 2.5 \times (1133.12)^{0.35} = 29.3 \text{ Months.}$$

Likewise,

average staff size (SS)

$$= \frac{E}{D} = \frac{1133.12}{29.3} = 38.67 \text{ persons.}$$

$$\text{Productivity (P)} = \frac{\text{KLOC}}{E} = \frac{200}{1133.12} = 0.1765 \text{ KLOC / PMs}$$

$$= 176 \text{ LOC/PM.}$$

Intermediate COCOMO

This model computes the software development effort as “a function of program size and a set of “Cost drivers” (15 Nos.), that includes subjective assessment of product, Hardware, personnel and project attributes.

- * Product characteristics, Reliability, database size and complexity.
- * *Hardware characteristics* : Execution time, Main storage, virtual machine reliabilities, turn around time etc.
- * *Personal attributes* : Analyst capability, application experience, programmer capability, virtual machine experience programming language experience.
- * *Project attributes* : Modern programming environment, use of software tools, development schedule.

Each cost driver is rated for a given project environment. The rating uses a scale very low, low, nominal, high, extra high which describes to what extent the cost driver applies to the project being estimated.

The equation (for intermediate COCOMO) to find out.

$$\text{Effort}(E) = a_i (\text{KLOC})^{b_i} \times \text{EAF}$$

$$\text{Development}(D) = C_i (E)^{d_i}$$

where EAF = Effort Adjustment Factor and a_i , b_i , c_i and d_i are coefficients in intermediate

COCOMO (a_i) → Coefficient
→ intermediate

The multiplying factor for all 15 cost drivers are multiplied to get the effort adjustment factor (EAF).

The values for EAF range from 0.9 to 1.4

Table - 4.6 : Coefficient Table intermediate COCOMO

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

The steps to estimate the project using this model are :

- Identify the mode (organic, Semi detached, embeded) of development for the new product (Project).
- Estimate the size of the project in KLOCs
- Adjust the 15 cost drivers.
- Calculate the predicted project effort (E)
- Calculate the development effort (E) in intermediate mode by multiplying with effort adjustment factor (EAF)
- Calculate the project duration or development time (D).

Table-4.7 : Cost - driver rating for intermediate COCOMO

Cost Drivers		RATINGS					
Product attributes	Cost Drivers Code	Very	Low	Nominal	High	Very	Extra
a) Required software reliability	(RELY)	0.75	0.88	1.00	1.15	1.40	-
b) Database size	(DATA)	-	0.94	1.00	1.08	1.16	-
c) Product complexity	(CPLX)	0.70	0.85	1.00	1.15	1.30	1.65
Computer attributes							
a) Execution time constraint	RELY	-	-	1.00	1.11	1.30	1.66
b) Main storage constraint	DATA	-	-	1.00	1.06	1.21	1.56
c) Virtual machine volatility	CPLX	-	0.87	1.00	1.15	1.30	-
d) Computer turnaround time	TURN	-	0.87	1.00	1.07	1.15	-
Personnel Attributes							
a) Analyst capability	ACAP	1.46	1.19	1.00	0.86	0.71	-
b) Application experience	AEXP	1.29	1.13	1.00	0.91	0.82	-
c) Programmer capability	PCAP	1.42	1.17	1.00	0.86	0.70	-
d) Virtual machine experience	VEXP	1.21	1.10	1.00	0.90	-	-
e) Programming language experience	LEXP	1.14	1.07	1.00	0.95	-	-
Project Attributes							
a) Modern programming practices	MODP	1.24	1.10	1.00	0.91	0.82	-
b) Use of software tools	TOOL	1.24	1.10	1.00	0.91	0.83	-
c) Required development schedule	SCED	1.23	1.08	1.00	1.04	1.10	-

Example -4.16:

Estimate the effort and development time (D) using intermediate COCOMO with the mode i.e. organic, size = 200 KLOC cost driver and EAF = 1.086

Solution :

Here E_{effort} (E)

$$= a_1 (\text{KLOC})^{b_1} \times \text{EAF}$$

$$= 3.2 \times (200)^{1.05} \times 1.086$$

$$= 906 \text{ person-month.}$$

$$\text{Dev. time (D)} = 2.5 \times (906)^{0.38}$$

Detailed COCOMO

There are some drawbacks of using Basic and intermediate COCOMO model, as both the models use to calculate (estimate) the cost of a software product considering the product as a single entity. Whereas for a big software product, it is needed to modularize the big system into smaller systems (subsystems), some of them may be organic, semi-detached and even embedded also. The development complexity may be different, the reliability requirement may be high and the development team experience on the similar project may not be available. It results with errors on the final estimation.

To avoid it, the complete / detailed COCOMO model is introduced, where all the project characteristics (system / subsystems of any type i.e. organic semi detached or embedded) are taken into account to construct a software estimate.

The effort and development time is calculated as the sum of the estimates for the individual subsystems, and the cost of the subsystems are calculated separately.

COCOMO - II

Constructive cost Model-II is the revised version of original COCOMO, was developed at University of Southern California under the leadership of *Dr. Barry Boehm* in late 90s, because COCOMO was not enough to apply to new software development practices. It provides a quantitative analytic framework, and set of tools and techniques for evaluating the effects of software technology improvements on software life cycle (all different phases) costs and schedules.

The following categories of software applications / projects are identified by COCOMO-II for estimation.

- End User Programming

This category is applicable for small systems, developed by the end user using the software application generator e.g.- *MS-Excel, MS-Access, MS-Studio etc.*

Infrastructure Sector :

This category is applicable to the infrastructure development i.e the software that provides infrastructure like *operating systems, DBMS, user interface management system, Networking systems etc.* Some commercial examples are *microsoft products, Oracle, DB2, Maya, Linux, 3D-Studio.*

The developers under this category generally should have good knowledge of software development and software engineering practices and knowledge about the applications. Many re-usable components of the product lines may be used through new technology to build new components.

Intermediate Sector

This category is partitioned into three sub-categories. The software developers need to have good knowledge of software development and software engineering practices, experience and expertise of infrastructure software and domain knowledge of some applications.

Application generators

This sub-category creates largely prepackaged capabilities for user programming. Some firms operating in this sector are *Microsoft, Lotus, Novell, Borland, Alias Wavefront, Oracle, IBM etc.* The product lines of these firms have many re-usable components, but they also require a good deal of new capability development from the re-usable components.

Application composition sector

This sub-category deals with applications which are too diversified to be handled by pre-packaged solutions. These applications can be generated / composed using inter-operable components. Typical components are *GUI builders, database managers, domain specific components such as financial, medical and industrial process control packages etc.*

These applications are complex, large, versatile, diversified and require specialisation and good knowledge of development and software engineering practices. Some application generator environments like *CASE tools, DB2, Oracle, 4GL programming tools (Dev-2000, VB, ASP, JSP, PHP etc)* are also used to develop the applications.

System integration

This sub-category deals with large scale, highly embedded systems. These are basically developed using application composition capabilities. Upfront engineering and customised software development activities.

COCOMO-II has three different models at three different stages for cost estimation.

Stage - I : Application composite model.

Stage - II : Early design model.

Stage - III : Post architectural model.

Application composition model

This model is suitable for projects built with modern *GUI-builder* tools. It is based on object points.

Application composite model :

- This model is used to estimate effort and schedule on projects that use integrated CASE tools for rapid application development.
- It is suitable for the projects need to be diversified into simple form, so that the project can be rapidly composed from inter-operable components.
- This model is suitable for the projects built with modern *GUI-builder* tools and based on object points.

- The size of the project is first estimated using object points, as the object points are easy to identify and count.
- Object points are the count of the screens, reports and 3GL modules developed in the application.
- Each count is weighted by a three level : simple, medium and difficult complexity factor.
- The estimating approach is commensurate with the level of information available during the planning stages of Application composite projects.

Early design model

- It involves the exploration of alternative system architectures and concepts of operation.
- This model is based on function points (or lines of codes when available) and set of five scale factors and 7 effort multipliers.

Post architecture model

- The post - architecture model is used under top-level design is complete and detailed information about the project is available.
- It estimates for the entire development life cycle and is a detailed extension of the Early-design model. This model is the closest in structure and formulation to the intermediate COCOMO.
- It uses source lines of code and / or Function points for the sizing parameter, adjusted for re-use and breakage; a set of 17 effort multipliers and a set of 5 scale factors, that determine the economic scale of the software product under development.
- The post-architecture model has been calibrated to a database of 161 projects from different sectors.

Halstead's software Science Metrics : An Analytical Estimation Techniques

The software science metric was developed by M.H. Halstead in the year 1977. His attempt was to measure / estimate the programming effort. It includes the program size, development effort and cost of a software product.

In this technique, some basic program parameters are used to develop mathematical expressions for program length, program volume, program level, programming effort, development time etc.

A set of standard measurable and countable properties of a program are taken into account. such as

- n_1 = number of unique or distinct operators appearing in the program.
- n_2 = number of unique or distinct operands appearing in the program.
- N_1 = total usage of all of the operators appearing in the program.
- N_2 = total usage of all of the operands appearing in the program.

From these metrics, Halstead Defines that program vocabulary

$$n = n_1 + n_2 \quad (1)$$

and the program length = total usage of all operators and operands i.e.

$$N = n_1 + n_2 \quad (2)$$

It may be necessary to know the relationship between program length 'N' and vocabulary 'n'.

$$\text{Length Equation } (N^1) = n_1 \log_2 n_1 + n_2 \log_2 n_2 \quad (3)$$

It is experimentally observed that.

' $\bar{N}$ ' is more accurate as compare to 'N'

Some alternate equations for the program length are

$$1) \quad N_F = \log_2(n_1!) + \log_2(n_2!) \quad [\text{by Jensen \& Varian 1985}]$$

$$2) \quad \bar{N} = n_1 \log_2 n_2 + n_2 \log_2 n_1 \quad [\text{Mehndiratta and Grover 1986}]$$

$$3) \quad N_C = n_1 \sqrt{n_1} + n_2 \sqrt{n_2} \quad [\text{Card and Agresti 1987}]$$

Program volume

This metric is for the size of any implementation of any algorithm

$$V = N \log_2 n$$

Program level

It is the relationship between program volume and potential volume. Only the clearest algorithm can have a level of unity.

$$L = V^* / V$$

Program level equation

It is an approximation of the equation of the program level. It is used when the value of potential volume is not known because it is possible to measure it from an implementation directly.

$$L^1 = n_1^* n_2 / n_2 N_2$$

Potential volume

It is a metric for denoting the corresponding parameters in an algorithm's shortest possible form. Neither operators nor operands can require repetition.

$$V^* = (n_1^* + n_2^*) \cdot \log_2(n_1^* + n_2^*)$$

Effort equation

The effort required to develop a program can be obtained by dividing the program volume by the level of programming Language used to develop a code.

$$\text{So, Effort (E)} = V/L$$

$$\text{but } L = \text{Program Level} = V^*/L$$

$$\Rightarrow E = V / (V^* / V)$$

$$= \frac{V \times V}{V^*} = V^2 / V^*$$

Programmer's time

A concept concerning the processing rate of the human brain, developed by the psychologist John Stroud can be used. Stroud defined a moment as the time required by the human brain to perform the most elementary discrimination.

The Stroud number 'S'

= Stroud's momnts per second with $5 \leq S \leq 20$
(generally $S = 18$)

Time = $T' = E/S$

$$= \frac{(n_1 N_2 (n_1 \log_2 n_1 + n_2 \log_2 n_2) \log_2 n)}{2n_2 S}$$

4.9.6 Cyclomatic complexity

A measure of the complexity of a program was developed by *McCabe* in the year 1976. He developed a system that he called cyclomatic complexity of a program.

It is a software matrix that provides a quantitative measure of the logical complexity of a program. It measure the number of independent paths in a program and provides an upper bound for the number of tests that must be conducted to ensure that all the statements in a program have been executed at least once. Let's take an example of a flow chart of a program in the figure 4.6.

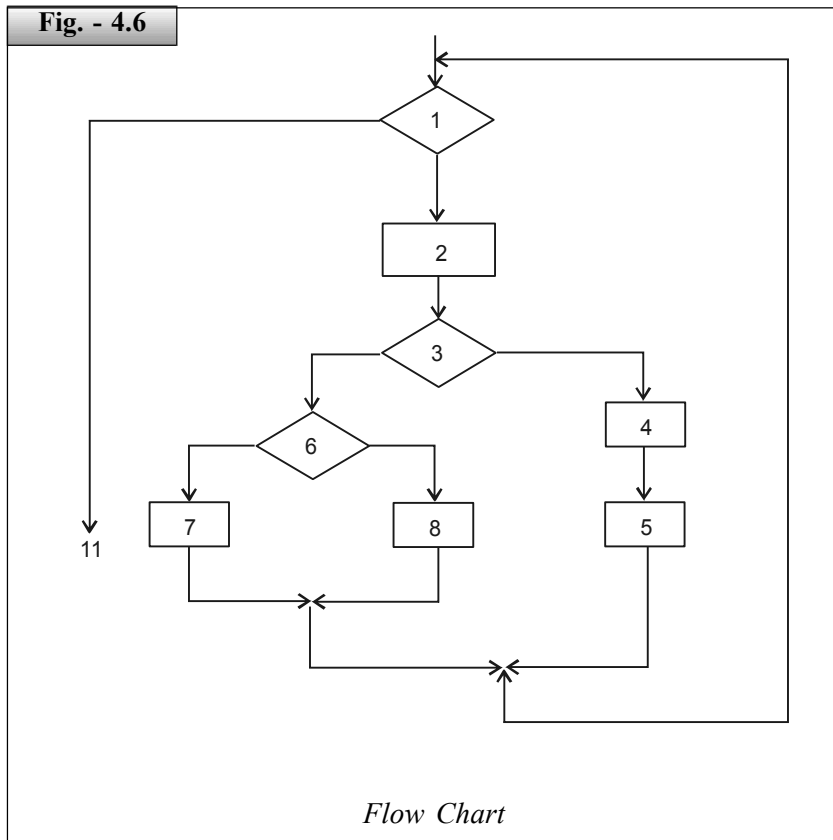
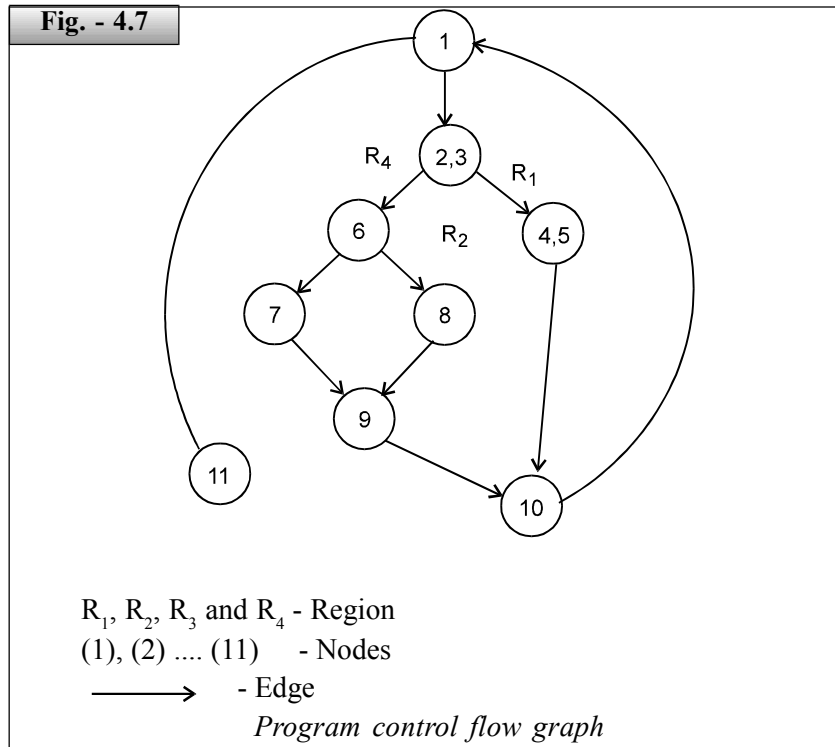


Fig. - 4.7



4.9.6.1 Program Control Flow Graph (CFG)

A program control flow graph consists of nodes, edges and regions.

- * Node represents a straight line section of the code i.e code without jump in or out.
- * Edge represents a possible sequence of execution.
- * Sequential statements should be put together.
- * The decision making statements should be branched into false and true parts.

An independent path must move along at least one edge that has not been traversed before the path is defined.

In case of the flow graph given.

- Path 1 : 1 – 11
 Path 2 : 1 – 2 – 4 – 5 – 10 – 1 – 11
 Path 3 : 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11
 Path 4 : 1 – 2 – 3 – 6 – 7 – 9 – 10 – 1 – 11

Each new path creates a new edge.

The path

Cyclomatic complexity is generally used in graph theory and is computed in one of the three ways.

1. No. of regions correspond to the cyclomatic complexity.
2. Cyclomatic complexity $V(G)$, for graph G , is defined as

$$V(G) = E - N + 2$$

E = No. of flow graph edges.

N = No. of flow graph nodes.

3. Cyclomatic complexity, $V(G)$ for graph G , is also defined as

$$V(G) = P + 1$$

where P = no. of predicate nodes contained in flow graph G .

Predicate node : The node which is a conditional statement and can also be called as decision node. This is represented by lower case alphabets.

The cyclomatic complexity w.r.t. the flow graph given is calculated as follows.

- The Graph has 4 regions.
- Edges = 10
- nodes = 9

$$\text{So, } V(G) = 11 \text{ edges} - 9 \text{ nodes} + 2 = 4$$

$$V(G) = 3 \text{ predicate node} + 1 = 4$$

4.9.6.2 Advanages of cyclomatic compexlity

- It can be used as an ease of maintenance metric.
- It is used as a quality metric, gives relative compexlity of various designs.
- It can be computed early in the life cycle.
- It measures the minimum effort and best areas of concentration for testing.
- It guides the testing process by limiting the program logic during development.
- It is easy to apply.

4.9.6.3 Disadvantages

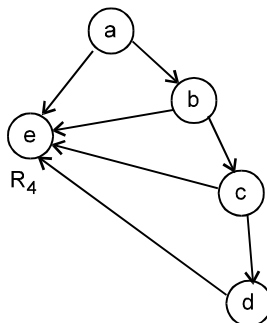
- It's a measure of program's control complexity and not the data complexity.
- It is applied both on nested and non-nested loops, but the deeply nested conditional structures are harder to understand than non-nested structures.
- In some cases, it fails to differentiate the complexity of simple cases involving single conditions.

Example - 4.17 :

Calculate the cyclomatic complexity of the following control flow graph.

(method-1)

Solution : Here the number of rignions = 4
(R_1 , R_2 , R_3 and R_4)



So, the cyclomatic complexity
= number of regions = 4 (Ans)

(Method-2)

In the control flow graph.

number of edges i.e. $E = 7$

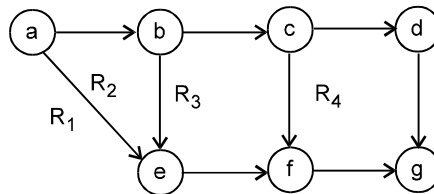
number of nodes i.e. $N = 5$

So, cyclomatic complexity

$$\begin{aligned}\text{i.e. } V(G) &= E - N + 2 \\ &= 7 - 5 + 2 \\ &= 2 + 2 \\ &= 4 \quad (\text{Ans})\end{aligned}$$

Example -4.18

Calculate the cyclomatic complexity of the given control flow graph.



Solution : (Method-1)

Here the number of regions = 4

so cyclomatic complexity = 4 (Ans)

(Method-2) In the control flow graph.

number of edges (E) = 9

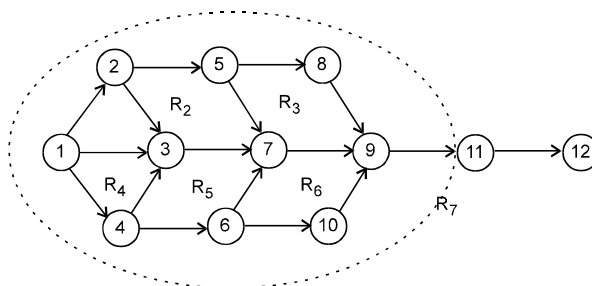
number of nodes (N) = 7

so, cyclomatic complexity.

$$\begin{aligned}V(G) &= E - N + 2 \\ &= 9 - 7 + 2 \\ &= 2 + 2 \\ &= 4 \quad (\text{Ans})\end{aligned}$$

Example - 4.19

Calculate the cyclomatic complexity of the given control flow graph.



Solution :

(Method-1) : Here no. of regions = 7
so cyclomatic complexity = 7 (Ans)

(Method-2) : In the graph
no. of edges (E) = 17
no. of nodes (N) = 12

So cyclomatic complexity i.e.

$$V(G) = E - N + 2$$

$$= 17 - 12 + 2$$

$$= 5 + 2$$

$$= 7 \quad \text{(Ans)}$$

Example - 4.20

For the given code, draw the control flow graph and calculate the cyclomatic complexity using method-1, method-2 and method-3 (using predicate node).

```

1.  Scarf ("% d % d % d", &a, &b, &c) ;
2.  if (a > 10)
3.  {
4.  printf("Hello") ;
5.  if (b < a)
6.      {
7.  printf("a");
8.  if (c > a)
9.      {
10.  Printf ("c") ;
11.      }
12.  }
13. else
14. {
15. printf("b");
16. }
17. }
18. printf("Hi");

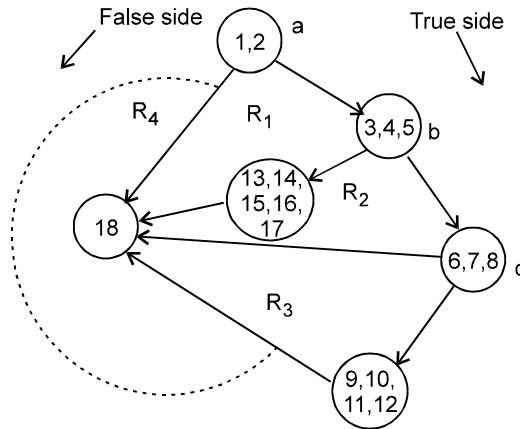
```

Soluton :

- (a) Statement no. 1 & 2 will be executed. So the node = (1,2)
- (b) if statement 2 is true, then statments 3, 4, 5 will be executed. i.e. the node right (Yes condition) to (1, 2) node willbe (3, 4, 5)
if statement 2 is false, statement 18 will be executed.
i.e. the node left (false condition) to (1, 2) is (18).
- (c) if statement 5 is true, then statments 6, 7, 8 will be executed i.e. the node right to (3, 4, 5) will be (6, 7, 8)

if the statement is false, then the statements (13, 14, 15, 16, 17) will be executed.
i.e. the node left to (3, 4, 5) will be (13, 14, 15, 16, 17) and terminated at (18).

- (d) if statement 8 is true, then statements 9, 10, 11, 12 will be executed.
i.e. the node right to (6, 7, 8) will be (9, 10, 11, 12) and terminated at (18).
if the statement 8 is false, the statement will be terminated at (18).
now, from above clarification, the control flow graph will be.



(a, b, c are predicate nodes)

Complexity calculation.

Method -1 : Here the number of regions = 4

so the cyclomatic complexity = 4 (Ans)

Method - 2 :

From the graph

No. of Edges (E) = 8

No. of nodes (N) = 6

so cyclomatic complexity

$$\begin{aligned} V(G) &= E - N + 2 \\ &= 8 - 6 + 2 = 2 + 2 = 4 \text{ (Ans)} \end{aligned}$$

Method - 3

From the graph

no. of predicate nodes P = 3 (a, b, c)

so, cyclomatic complexity

$$\begin{aligned} V(G) &= P + 1 \\ &= 3 + 1 \\ &= 4 \end{aligned} \quad \text{(Ans)}$$

4.10 ESTIMATION ON STAFFING

Every project has a defined number of people on the software i.e number of people on the software team. As time allocation occurs, the project manager must ensure about the numebr of people required as well as the proper staffing policy to carryout the project.

There is a common myth that is believed by many project managers even today. “If we fall behind the project schedule, we can add more people and catch up later in the project”. But it does not work, rather it create a disruptive effect on the project, resulting the slip of project schedule.

In this context, the *Putnam-Norden-Rayleigh (PNR)* curve provides a relationship between effort applied and delivery time for a software project. Let’s discuss it in a sequence.

4.10.1 Rayleigh’s model

This is more precisely to call as a mathematical curve which is modelled by a differential equation and used to plot the *distribution of effort* for a project.

$$\text{The equation } m(t) = \frac{dy}{dx} = 2 Kate^{-at^2} \quad \dots(1)$$

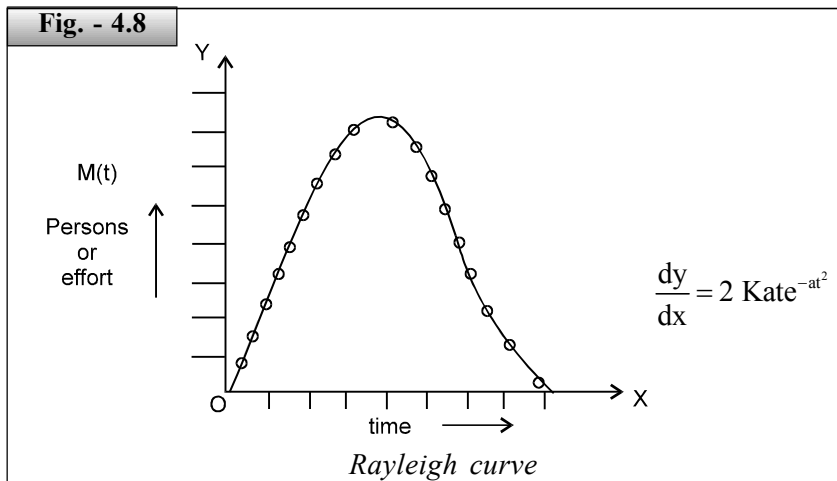
where $\frac{dy}{dx}$ = manpower utilization rate per unit time.

t = elapsed time.

a = parameter that affects the shape of the curve.

and K = the area under the curve in the interval $[0, \infty]$

The Rayleigh’s curve can be draw as shown Fig. 4.8.



Integrating equation (1), on interval $[0, t]$

$$\text{We obtain } Y(t) = K \left[1 - e^{-at^2} \right] \quad \dots(2)$$

Where $y(t)$ is the cumulative manpower used upto time t ,

$$y(0) = 0$$

$$y(\infty) = K$$

i.e. the cumulative manpower at the start of the project is zero (null) and grows monotonically towards the total effort K (area under curve).

I. Norden's work

Norden studied near about 150 different projects, not only at the project level but also for the software subsystem development.

Here the *Rayleigh's model* mathematics are taken into account. Let's consider the equation(2) of *Rayleigh's model*.

$$\text{i.e. } y(t) = K \left[1 - e^{-at^2} \right]$$

here the parameter 'a' plays an important role in determining the peak manpower.

The dimension of a i.e $1/\text{time}^2$.

The larger the value of 'a', earlier the peak time occurs and steeper is effort or person profile.

Deriving the manpower function relative to time and finding the zero value of this derivative, the relationship between the peak time " t_d " and "a" can be found to be

$$\frac{d^2y}{dt^2} = 2Ka e^{-at^2} [1 - 2at^2] = 0$$

$$\text{therefore } t_d^2 = \frac{1}{2a}$$

' t_d ' denotes the time where maximum effort rate occurs, so ' t_d ' on the time scale is close to the total project development time.

if we substitute ' t_d ' for t in the equation $y(t) = K \left[1 - e^{-at^2} \right]$,

an estimation for the development time is obtained as.

$$E = y(t) = K \left[1 - e^{-\frac{td^2}{2}} \right]$$

$$\text{or } E = y(t) = K \left[1 - e^{-0.5} \right]$$

$$\text{or } E = y(t) = 0.3935 K \text{ (Person month)}$$

$$\text{We know } t_d^2 = \frac{1}{2a}$$

$$\text{or, } a = \frac{1}{2t_d^2}$$

the peak manning time is related to 'a'.

The *number of people involved* in the project at the peak time then it becomes easy to determine by replacing 'a' with $\frac{1}{2td^2}$ in the equation,

We have the *Norden's formula*

$$E = M(t) = \frac{2K}{2td^2} t \cdot e^{-\frac{t^2}{2td^2}}$$

or,
$$E = \frac{K}{t_d^2} \cdot t \cdot e^{-\frac{t^2}{2td^2}}$$

at time $t = td$.

the peak effort (manning) $m(t_d)$ is obtained i.e.

$$m(td) = (t_d) = M_0 = \frac{K}{td\sqrt{e}} \quad \sqrt{e} = 1.648$$

where K = total project cost (or effort) in person-years.

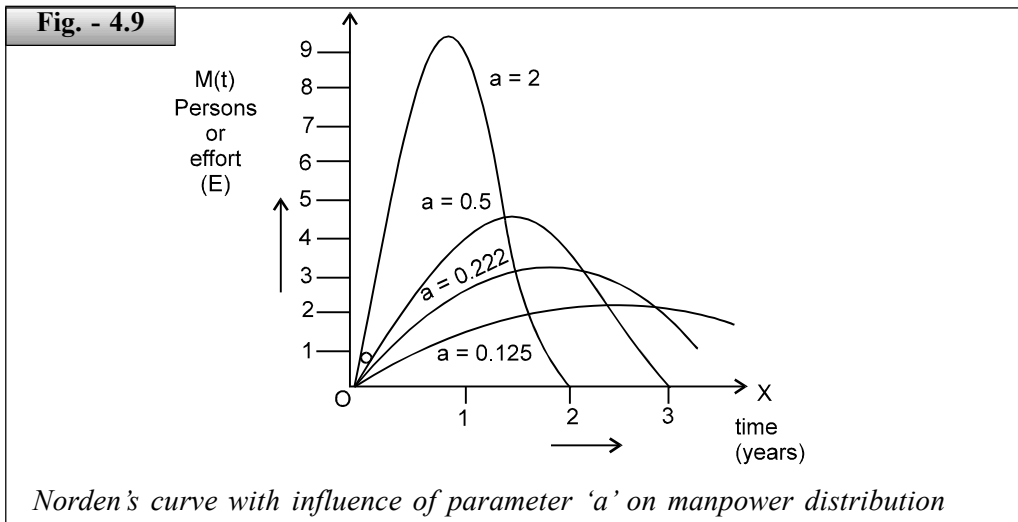
t_d = delivery time in years.

m_0 = no. of persons employed at the peak.

The average rate of the staff (team) build up.

$$= M_0/t_d.$$

Norden's curve can be shown as follows :



II. Putnam's model

Putnam observed the Ray leight's curve and apparently it was observed the same tendency as of Norden's curve for the man-power curve to rise, peak and then exponentially trail off as a function of time.

Putnam derived the manthematical expression.

$$\text{i.e. } L = C_k \cdot K^{1/3} \cdot t_d^{4/3}$$

where K = total effort in person-Month

L = Product size in KLOC

t_d = development time

C_k = the technological constant that influences the progress of the programmer during development.

(i.e. $C_k = 2 \Rightarrow$ poor environment, where as

$C_k = 8 \Rightarrow$ good and

$C_k = 11 \Rightarrow$ excellent environment.

from the above expressoin. the total effort (K) can be calculated

$$\text{i.e. } K = \frac{L^3}{(C_k)^3 \cdot (t_d)^4}$$

but, for a particular product size

$$C = \frac{L^3}{(C_k)^3} \text{ can be taken as constant}$$

$$\text{now, } K = \frac{C}{(t_d)^4}$$

Putnam's estimation model works efficiently for very large systems, but efficiently for very large systems, but not suitable for small / medium projects.

Jensen's model (1984)

This model is similar to Putnam's model, but it focuses on handling small / medium size projects.

The mathematical equation under this model is.

$$L = C_{te} \cdot t_d \cdot K^{1/2}$$

where C_{te} = effective technology constant

t_d = development time.

k = effort

$$\text{now } K = \frac{C}{(t_d)^2}$$

$$[\text{where the constant } C = \frac{L^2}{(C_{te})^2}]$$

4.11 TEAM STRUCTURE

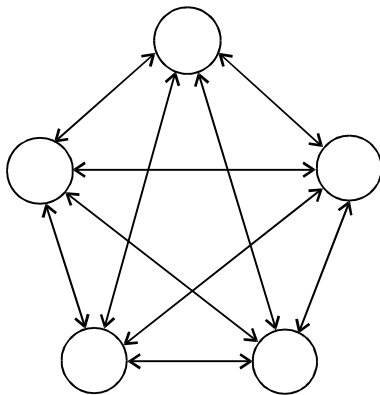
Often a team of people is assigned to a project. For the team to work as a cohesive group and contribute the most to the project, the people in the team have to be organized in some manner. The structure of the team has a direct impact on the product quality and project productivity.

Two basic philosophies have evolved for organizing a team : *egoless teams* and *chief programmer teams*. Egoless teams consist of ten or fewer programmers. The goals of the group are set by consensus, and input from every member is taken for major decisions. Group leadership rotates among the group members. Due to its nature, egoless teams are sometimes called *democratic teams*. The structure results in many communication paths between people, as shown in Figure 4.10.

The structure allows input from all members, which can lead to better decisions in difficult problems. This suggests that this structure is well suited for long-term research-type projects that do not have time constraints. On the other hand, it is not suitable for regular tasks that are not too complex and that have time constraints. For such tasks, the communication in democratic structure is unnecessary and results in inefficiency.

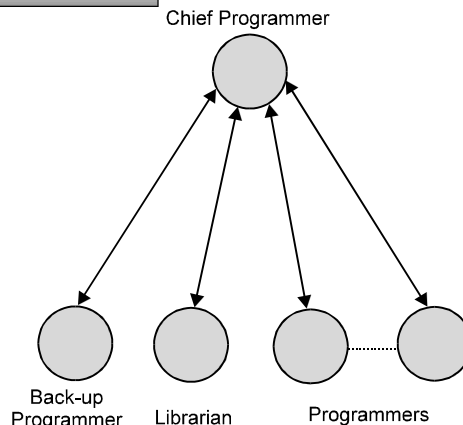
A chief programmer team, in contrast to egoless teams, has a hierarchy. It consists of a chief programmer, who has a backup programmer, a program librarian, and some programmers. The chief programmer is responsible for all major technical decisions of the project. He does most of the design and assigns coding of the different parts of the design to the programmers. The backup programmer helps the

Fig. - 4.10



Communication paths in a democratic team structure.

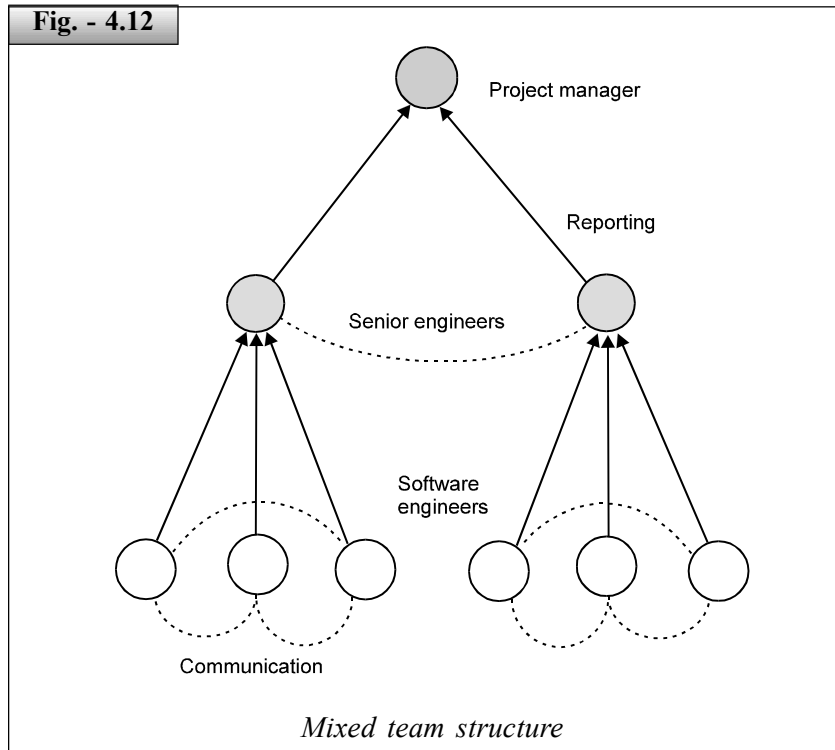
Fig. - 4.11



Chief programmer teams.

chief programmer makes technical decisions, and takes over as the chief programmer if the chief programmer falls sick or leaves. The program librarian is responsible for maintaining the documentation and other communication-related work. This structure considerably reduces interpersonal communication. The communication paths are shown in Figure 4.11.

This kind of structure is well suited for projects with simple solutions and strict deadlines. It is not suitable for difficult tasks where multiple inputs are useful.



A third team structure, called the *controlled decentralized team* of mixed team structure shown in figure 4.12 tries to combine the strengths of the democratic and chief programmer teams. It consists of a project leader who has a group of senior programmers under him, while under each senior programmer is a group of junior programmers. The group of a senior programmer and his junior programmers behaves like an egoless team, but communication among different groups occurs only through the senior programmers of the groups. The senior programmers also communicate with the project leader. Such a team has fewer communication paths than a democratic team but more paths compared to a chief programmer team. This structure works best for large projects that are reasonably straightforward. It is not well suited for very simple projects or research-type projects.

4.12 SOFTWARE RISK MANAGEMENT

Software risk can be understood as a problem that threatens the success of a software project.

More precisely, it's a measure of the inability to achieve objectives within the cost, schedule, and constraints.

Risk management is the process of identifying, addressing and eliminating these potential problems before they can damage the project.

Software Risks can be categorised as :

- Project risks - (Risk on project schedule)
- Product risks - (Risk on quality)
(also called technical risk)
- Business risks - (risk on client organisation)

Project risk are concerned with budgets, schedule, personnel, resources and customer related problems. As software is intangible, it is difficult to monitor and control a software project. The invisibility of software product result most projects suffer from the risk of out scheduled.

Product risk is based on the quality of a software product, and the quality purely depends as design, implementation, interfacing, testing and maintenance. This is also called as technical risks. Mainly, it is due to the inadequate knowledge of the developers about the product.

Business risks are generally associated with the commitments as made to the clients to have an excellent product within their budgets in time, which is the threats to the development organisation.

Software development projects are subject to the following risks.

- Poorly defined requirements.
- Changes in client requirements.
- Poor techniques for estimation.
- Rapid technological changes.
- Lack of development skills etc.

Definition :

Software risk management is the process of identifying software risks and planning to avoid those risks or to minimize their effects if they can't be avoided.

Risk has two characteristics.

- Uncertainty : an event may or may not happen.
- Loss : an event has unwanted consequences or results.

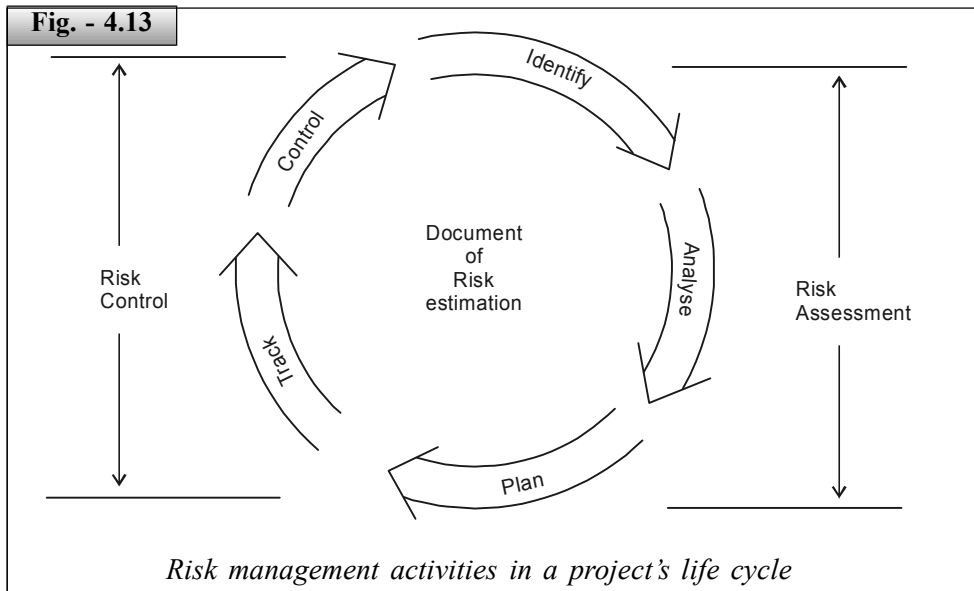
Risk management can be implemented in many ways. Some of the management approaches are listed below.

- Planned and documented risk management process for the project or product.
- Prospective assessment of a process by the management team to find and manage the problems.
- Periodical assessment of problems in project activities.
- Setting up definite set of evaluation criteria that covers all facets of the program.
- Documentation of all on-going results of risk management process.

4.12.1 Risk management activities

The risk management process must be a part of regularly scheduled periodic product management. It requires identifying and managing risks (in a routine format) throughout all phases of project's life.

The continuous risk management functions in all phases of project's life can be show as follows.



Risk assessment :

This activity is associated with “ranking of risk w.r.t. it's effect”. risk assessment activity include the following line.

- Risk identification
- Risk analysis
- Risk prioritifaction.

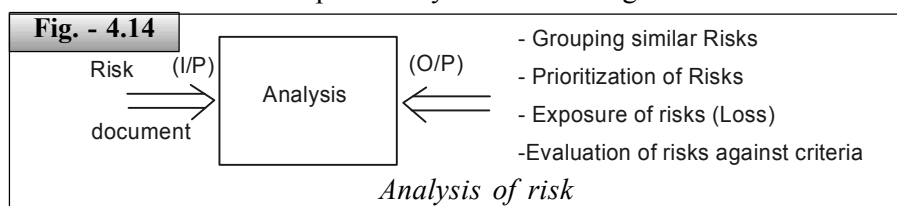
Risk identification: It's a systematic attempt to develop a list of risk items called risk statement through check-list. (i.e some predefined common areas of the project where the possibility of risk is there), interviews with the customers, brain storming reviews, surveys etc.

Once the risks are identified, they are evaluated with the criteria i.e. probability of risks, consequence and time frame of action.

All the risks are documented w.r.t. name of risks, risk statement and context of the risk and the document is communicated to all the members of the project.

Risk analysis :

Once the risks are identified and documented, they need to be analysed using different criteria and is to asses the probability of loss and magnitude of risk items.



Risk prioritization

It helps the project team to focus on the most severe risks by assessing the risk exposure.

risk exposure (P) = Probability of risk (r)

or $\times$ relative loss

priority due to risk (s)

or, $P = r * s$

where r = (0.1 to 1.0) on a scale of 1 – 10.

if the exposure is high, the risk needs to be tackled first.

4.12.2 Risk control

It is the process of managing the risks to achieve the desired results, what the software project means for. Risk control process involves the following activities.

- Risk planning
- Risk mitigation
- Risk resolution
- Risk monitoring

Risk planning :

It produces a plan to deal with each and every significant risks using the strategies like avoiding risk by undertaking projects on similar proven technologies and minimizing the risks through risk awareness and control training program.

Risk mitigation :

It is a plan to reduce or eliminate the highest risks by assigning a primary handler for the action.

Risk resolution : It is the executive of an implemented plan to resolve the risk by the project manager and correction of the deviation (if any) during the process of resolution.

The project manager also deals with the cost of handling the risk and the corresponding reduction of risk.

The “risk leverage” i.e. the difference in risk exposure divided by the cost of reducing risk.

or,

$$\text{Risk leverage} = \frac{(\text{Risk exposure before reduction} - \text{Risk exposure after reduction})}{\text{Cost of reduction}}$$

Risk monitoring :

It is a continuous assessment of risks as any project proceeds and conditions change, so that the risks can be easily handled (minimized or avoided) before they (risks) make the project affected.

SUMMARY

- A feasibility study is an investigation to check that a development is worthwhile. It is carried out at the start of a project. It assesses technical feasibility and costs.
- Cost-benefit analysis compares the cost of developing the system with the money saved by using it. The costs include development, additional hardware, maintenance and training.
- Good software project management is essential if software engineering projects are to be developed on schedule and within budget.
- Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative so there is no body of experience to guide their management. Software processes are not well understood.
- Software managers have diverse roles. Their most significant activities are project planning, estimating and scheduling. Planning and estimating are iterative processes. They continue throughout a project. As more information becomes available, plans and schedules must be revised.
- A project milestone is a predictable outcome of an activity where some formal report of progress should be presented to management. Milestones should occur regularly throughout a software project. A deliverable is a milestone that is delivered to the project customer.
- Project scheduling involves the creation of various graphical plan representations of part of the project plan. These include activity charts showing the interrelationships of project activities and bar charts showing activity durations.
- Major project risks should be identified and assessed to establish their probability and the consequences for the project. You should make plans to avoid, manage or deal with likely risks if or when they arise. Risks should be explicitly discussed at each project progress meeting.

EXERCISE

1. List the major responsibilities of a software project manager.
2. What do you understand by sliding window planning? Explain with the help of a few examples the types of projects for which this form of planning is especially suitable. What are its advantages over conventional planning?
3. When does the project planning activity start and end in a software life cycle? List the important activities software project managers perform during project planning.
4. What do you understand by product visibility in the context of software development? Why is it important to improve product visibility during software development?
5. Using a schematic diagram show the order in which the following are estimated in the COCOMO estimation techniques: cost, effort, duration, size.

6. What are the various reasons for poor/inacurate estimation.
7. Discuss various types of COCOMO model. Explain the phase wise distribution of effort.
8. Describe the basic COCOMO model in detail?
9. Why cost estimation plays an important role in software development process? Discuss COCOMO model in detail?
10. Explain how Putnam's model can be used to compute the change in project cost with the change in project duration. What are the main disadvantages of using the Putnam's model to compute the additional costs incurred due to schedule compression? How can you overcome these disadvantages?
11. Suppose you are developing a software product in the organic mode. You have estimated the size of the product to be about 100,000 lines of code. Compute the nominal effort and the development time.
12. What does Halstead's volume metric represent conceptually? How according to Halstead is the effort dependent on program volume?
13. What are the relative advantages of using either the LOC or the function point metric to measure the size of a software product?
14. List the important shortcomings of LOC for use as a software size metric.
15. The following table indicates the various tasks involved in completing a software project, the corresponding activities, and the estimated effort for each task in person-months.

<i>Notation</i>	<i>Activity</i>	<i>Effort in person-months</i>
T_1	Requirements specification	1
T_2	Design	2
T_3	Code actuator interface module	2
T_4	Code sensor interface module	5
T_5	Code user interface part	3
T_6	Code control processing part	1
T_7	Integrate and test	6
T_8	Write user manual	3

The precedence relation $T_i \leq \{T_j, T_k\}$ implies that the task T_i must complete before either task T_j or T_k can start. The following precedence relation is known to hold among different tasks: $T_1 \leq T_2 \leq \{T_3, T_4, T_5, T_6\} \leq T_7$. Draw the **Activity network** and the **Gantt chart** representations for the project.

16. Suppose you are the project manager of a software project that consists of the following activities.

<i>Activity no.</i>	<i>Activity name</i>	<i>Duration (weeks)</i>	<i>Immediate predecessor</i>
1.	Obtain requirements	4	—
2.	Analyze operations	4	—
3.	Define subsystems	2	1

4.	Develop database	4	1
5.	Make decision analysis	3	2
6.	Identify constraints	2	5
7.	Build module 1	8	3, 4, 6
8.	Build module 2	12	3, 4, 6
9.	Build module 3	18	3, 4, 6
10.	Write report	10	6
11.	Integration and test	8	7,8,9
12.	Implementation	2	10, 11

- (a) Draw the Activity network representation of the project.
- (b) Draw the Gantt chart representation of the project.
17. Explain when to use PERT charts and when to use Gantt charts, if you were to perform the duties of a project manager.
 18. Suppose that an on-the-shelf software product for business application costs Rs. 50,000 and that its size is 40 KLOC. Assuming that in-house engineers cost Rs. 6000 per programmer-month (including overheads), would it be more cost-effective to buy the product or build it? Which elements of the cost are not included in the COCOMO estimation model? What additional factors should be considered in making the buy/build decision?
 19. List the important items that a Software Project Management Plan (SPMP) document should discuss.
 20. What is risk? Is it economical to do risk management? What is the effect of this activity on the overall cost of the project?
 21. What are the two major sources of risk? Explain the three categories of risks that a software project might suffer from.



Requirement Engineering

5.1 INTRODUCTION

According to Webster's Ninth New Collegiate Dictionary, a requirement is defined as "something required; something wanted or needed". The IEEE Standard 729 defines a requirement as.

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system... to satisfy a contract, standard, specification, or other formally imposed document.

From these above definitions, we came to know that "what must be included in requirements" while omitting "what should be excluded".

The requirements phase begins in the pure software sense when.

- a new software idea arises (the equivalent of an invention in hardware).
- there is recognition that a problem exists and requires a solution.

The recognised problem might be purely application-oriented (for e.g. we need a new logistics management system); business-oriented (for e.g. we must develop a new product to challenge a competitor's offering, or we must develop a new product to meet our ten year revenue goals) or product improvement oriented (for e.g., we must rebuild a product to reduce its failure rate or poor maintenance record).

The end of the software requirements phase occurs when we have the complete description of the external behaviour of the software to be built. This complete description includes documenting all interfaces between the software and its environment (that is host hardware, or other software, humans). This complete description is recorded in a form of document (software requirement specification).

* A software requirements specification (SRS) is a document containing a complete description of "*what the software would do without describing how it would do it.*"

Requirements are statements of ***what and not how.***

An SRS is a document containing a complete description of a product's external behaviour. However, depending on how the software is being developed and who is writing the SRS, it may be detailed or general.

5.2 PROBLEM ANALYSIS AND PRODUCT DESCRIPTION

There are 2 different types of activities that occur during the requirements phase, i.e *problem analysis and product description*. The things done during each phase are quite distinct.

During problem analysis, analysts spend their time in brainstorming, interviewing people who have the most knowledge about the problem at hand, and identifying all possible constraints on the problem's solution. At this time, there is a considerable expansion of information and knowledge about the problem. The greatest problems that occur during this time are finding ways of trading off constraints and organising the excessive amount of information. Problem analysis must be performed until a complete understanding of the problem is reached.

During product description, we need to prepare a document that describes the expected external behaviour of the product to be built to solve the now-understood problem. This is a time to organise ideas, resolve conflicting views, and eliminate inconsistencies and ambiguities.

It must be pointed out, that the 2 phases do not represent two temporally sequential, mutually exclusive phases. There are 2 reasons for this.

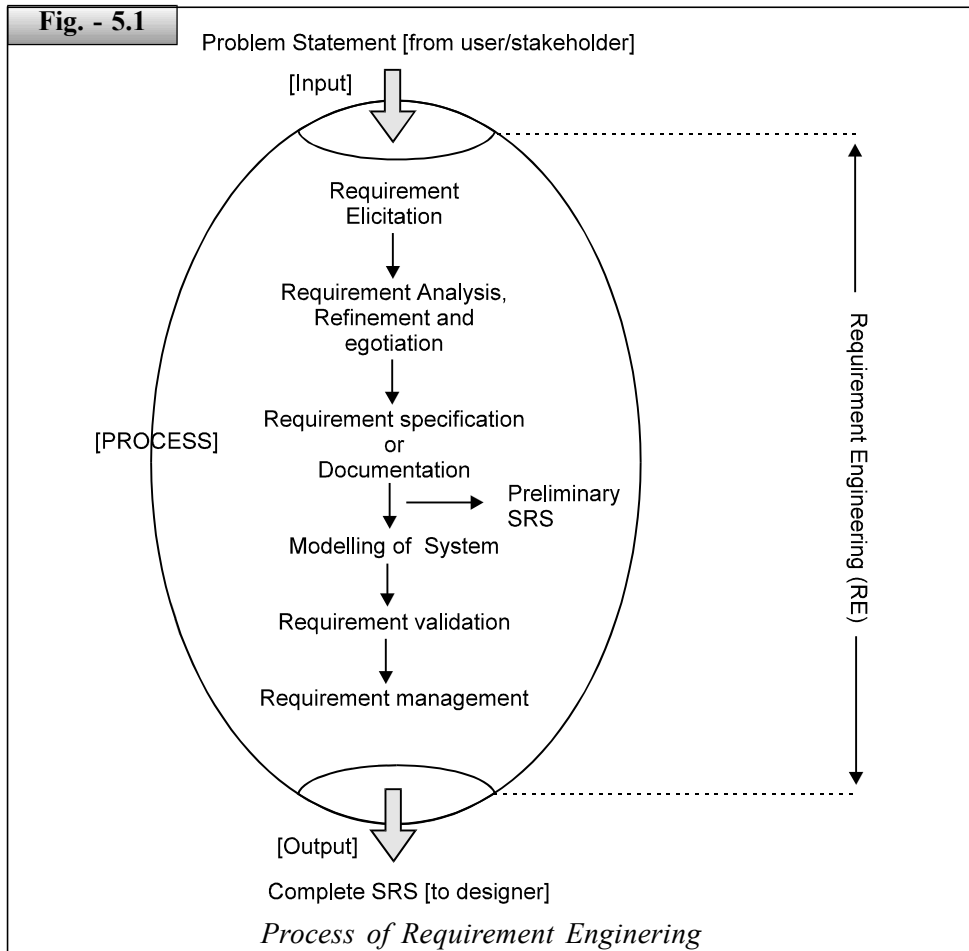
- 1) First, some product development require little or no problem analysis. In particular, problem analysis is applicable to only new, difficult, or yet-unsolved problems. Why waste time analyzing a problem that is already well understood ?
- 2) Second, given a new, difficult, or yet-unsolved problem, we do not perform problem analysis and then writing an SRS. In reality what is done is something like this problem analysis is initiated. As parts of the problem become well understood, corresponding sections of the SRS are written, finally the last aspect of the problem is analyzed and the SRS is finished.

5.3 REQUIREMENTS ENGINEERING (RE)

One of the main challenges of any systems engineering exercise is to ensure that we have specified a system that properly meets the customer's requirement and ensures that his/her expectations are satisfied. But how do we ensure this ? While this is not an easy thing, decades of research, development, learning and experience have shown that following good requirements engineering methodology helps. What the requirements engineering does is to make available a mechanism for the following.

- ✎ understanding what the customer wants
- ✎ analyzing needs
- ✎ feasibility assessment
- ✎ working out (negotiating) a solution that works and is reasonable.
- ✎ providing an unambiguous specification of the solution.
- ✎ validation of the specification.
- ✎ managing the requirements as things undergo changes during the other phases of the life cycle.

Requirement Engineering (RE) can be viewed as a system (approach), who responds a problem statement as input, processes (using different steps) and produces a complete document (SRS) as output. It is shown in the fig. 5.1



The process of requirements engineering has the following steps :

- Requirements elicitation.
- Requirements analysis, refinement and egotiation.
- Requirements specification / documentation
- Modeling of the system
- Requirements validation
- Requirements management.

We shall briefly deal with each one of the above steps in requirements engineering.

5.3.1 Requirements elicitation

Elicitation is the process of getting the set of requirements from all the stakeholders (the customer, the user, the management). while it may appear that it is relatively simple to ask the various stakeholders, the following (it is actually very difficult in practice).

- What are the objectives of the system or product ?
- What is to be achieved by the product or system ?
- How does the system or product help in the business needs ?
- How would you use the system or product on a daily basis ?

The process of requirements elicitation is considered very difficult in practice because of the following problems.

- *Scope problem* : This happens when the system's application is not well defined. In this situation the objectives of the overall system tend to be submerged in the face of the too many unnecessary details.
- *Understanding problem* : This happens when the stakeholders (customers, users, etc) are not at all sure about their needs. Usually this arises when people have poor domain of knowledge, inadequate appreciation of their work environment, poor communication skills etc.
- *Volatility problem* : The requirements tend to change over time. In other words, the requirements themselves are volatile.

While the process is difficult, the software engineer can follow the following guidelines for the requirements elicitation.

- Considers the business feasibility of the proposed system.
- Considers and evaluates the technical feasibility of the proposed system.
- Identifies persons in the organisation who would help to specify the requirements.
- Identifies persons in the organisation who could help to identify the strength in the requirements, if any
- State the technology environment of the system that includes hardware, software and communication platforms.
- List-down certain constraints that arise from the business domain that would reduce the functionality or performance of the system or product.
- States which requirements elicitation methods that would be used. Some of them are structured interviews, group meetings, filling in questionnaires, team brainstorming, etc.
- Ensures participation from all segments of organisations to have multiple perspectives on requirements.
- Writedown the rationale behind each requirement elicited.
- Makes a list of requirements that are unambiguous.
- Selects a such list of short requirements for prototyping.
- Help stakeholders to understand requirements better through usage *scenarios or use case analysis*.

After the requirements elicitation process, we would get the following.

- Need and feasibility document.
- Scope of system or product.
- List of all stakeholders who participated in the requirements elicitation.
- Technology environment document.

- List of all requirements, organized in a particular manner, say by function.
 - States all domain constraints that apply to each requirement.
 - Specification of the use cases or usage scenarios.
 - Prototypes to help understand requirements better.
- The stakeholders who participated in the requirements elicitation to ensure correctness of the documentation may review each of these.

5.3.2 Requirements analysis

Now we have a bulk of requirements. These requirements need to be analyzed to help, categorize them and reorganize them. Further the requirements are analyzed with respect to one another and the attributes such as consistency, incompleteness, ambiguity and priority are explored. As a guideline, consider asking the following questions.

- Is each requirement provided consistent with overall objectives of the system or product ?
- Are all requirements stated clearly in a detail appropriate manner to that requirements ?
- Is the requirement a necessary one or one of those “would be nice to have” one ?
- Is the requirement bounded in scope ?
- Is the requirement complete ?
- Is the requirement unambiguous ?
- Can you connect each requirement to a source in the organization ?
- Are you able to identify conflicting requirements ?
- Is the requirement consistent with the technology and environment ?
- Can we test each requirement, once implemented ?

We should note that stakeholders often ask for more things to be achieved than is possible, so it is necessary to ask the customers to prioritize their requirements. There is a need to negotiate with the customers when there are conflicting requirements. At this stage, risk features are attached with each requirement. Also a rough estimate of the requirements on the project cost, timelines etc. are also stated. It is a sort of an iterative process by which some inconsistent, conflicting requirements are weeded-out and the parties agree on a common ground of requirements.

5.3.3 Requirements specification

A requirement specification can be one of the following.

- i) A written document
 - ii) A graphical model
 - iii) A formal model (using mathematical basis)
 - iv) A collection of use cases
- A set of prototypes

It can even be any combination of the above.

5.3.4 Modeling the system

Some requirements studies do need to have some modeling of the system, to either create a better understanding of the system and / or to be the basis for further analysis. For newly encountered systems, it is useful to perform some systems modeling.

5.3.5 Requirements validation

In the requirements validation step, the requirements specification is subjected to some quality assurance. Assessment is done to ensure that the requirements specification has some or all of the following qualities.

- Non-ambiguity
- Consistency
- Completeness
- Conformance to certain standards

One of the mechanisms need for validation is the *formal technical review*. In this process, the stakeholders, the users, the customer, the engineers all go through the requirements specification attempting to weed out errors. Such things as conflicts, inconsistencies, omissions, ambiguities, etc. are tracked. As a guideline, we can ask the following questions.

- Are the requirements unambiguous ?
- Is the requirement connected with a source and verified ?
- Is the scope of the requirement bounded ?
- Are the ways in which the requirements related clear ?
- Are all the requirements within the domain constraints specified ?
- Are the requirements testable ?
- Are the requirements traceable ?
- Are the requirements readable and understandable ?
- Are there indexes, contents and cross-references ?
- Are requirements concerning performance, etc. clearly stated ?

These are just some guides. We need to actually work on a more exhaustive checklist of questions for validation.

5.3.6 Requirements management

The set of activities that assist in identifying, controlling and tracking requirements and changes to requirements during the process of implementation are called as *requirements management*. This part is often studied under the topic of *configuration management*.

5.4 CONDUCTING A REQUIREMENTS STUDY

A requirements study begins with communication between the customer and the requirements engineer. The task of the customer is to communicate his / her requirements (needs, aspirations and wish lists) to the requirements engineer. The task of the requirements engineer is to unearth all the requirements, structure them and write the requirements specification. In other words the requirements engineer has to understand the stated requirements of the customer. This communication is not quite as easy as it sounds and is often filled with troubles of various kinds.

Problems in communication can occur if the domain being explored, requires expertise to understand, if the language of communication has gaps, if the effectiveness of communication is poor and so on.

One simple method is for the analyst to start asking some context-free questions. Some of the questions could be

- Who is behind the request for this work ?
- Who will use the solution ?
- What will be the economic benefit of a successful solution ?
- Is there another source for the solution that you need ?

The idea is to have a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first meeting itself. The first set of context free questions focus on the customer, overall goals, and benefits.

Next we can ask some more focused questions that would help the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution.

- How would you characterize “good” output that would be generated by a successful solution ?
- What problem(s) will this solution address ?
- Can you show (or describe to) me the environment in which the solution will be used ?
- Are there special performance issues or constraints that will affect the way the solution is approached ?

Next, we can proceed to ask some additional questions that help in evaluating the effectiveness of the meeting itself.

- Are you the right person to answer these questions ? Are your answers “official” ?
- Are my questions relevant to the problem that you have ?
- Am I asking too many questions ?
- Is there anyone else who can provide additional information ?
- Is there anything else that I should be asking you ?

We should note that interviews of this kind have limited success. It should be used only as a preliminary approach to ease communication, and get over the initial awareness of the meeting. A better approach is to follow this up with meetings in a group so as to enable problem solving, negotiation, and specification.

5.5 FACILITATED APPLICATION SPECIFICATION TECHNIQUES (FAST)

To get the requirements study done well, the customer and the analyst have to work as a team. Alas, in reality, the customers and the analysts communicate through a series of memos, formal position papers, documents, and question and answer sessions. This method doesn't work very well. There will be many mis-understandings and omissions that continuously foster an ill-maintained relationship. The *Facilitated Application Specification Techniques* (FAST) is an approach that encourages the creation of a joint team of customers and developers who work together to achieve the following.

- Identify the problem
- Propose elements of the solution

- Negotiate different approaches
- Specify a preliminary set of solution requirements.
Many different approaches to *FAST* have been proposed over the years. But we can provide a common set of guidelines that may be useful.
- A meeting is conducted, preferably at a neutral site and attended by both developers and customers.
Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A “facilitator” (*who can be a customer, a developer, or an outsider*) controls the meeting.
- A “definition mechanism” is used. This can be in the form of work sheets, flip charts, wall stickers, or a white board.
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment to the goal.

5.6 IMPACT OF PROTOTYPING ON REQUIREMENTS

A *throwaway prototype* can be rebuilt during any synthesis phase in the software life cycle. While still in the requirements phase a throwaway prototype can be constructed and given to a potential user or customer in order to

- Determine the feasibility of a requirement.
- Validate that a particular function is really necessary.
- Uncover missing requirements.
- Determine the viability of a user interface.

Armed with the experience of having used the prototype, the team completes the *SRS* with increased assurance that the right system is being specified. If all requirements can be uncovered, well understood and agreed on without the prototype, we need not build one. Similarly, during the preliminary design phase, a throwaway prototype can be constructed to validate that a particular architecture actually exhibits the required set of properties, indicating that it has the potential to meet the requirements. If there is general agreement about design alternatives, we do not have to create a prototype. Finally, during detailed design, a throwaway prototype can be constructed to validate that a particular algorithm will meet its requirements. In all three cases, there are two important goals that should be borne in mind.

- The prototype should be quick and dirty. Its development should be quick because its advantage exists only if results from its use are available in a timely fashion.
- It should be dirty because there is no justification for building quality into a product that will be discarded. Among the dirty characteristics to be considered are no design, no comments, and no test plans, implementation in some relatively un-maintainable language that facilitates fast synthesis of programs. Examples of such languages are *SNOBOL*, *LISP* and *APL*.

5.7 USES OF THE SRS

The problem is repeatedly decomposed during problem analysis until it is understood thoroughly. However, the software requirements phase is not complete until the SRS has been written. This unit introduces the subject of the *SRS*, prescribes its appropriate content, explores the attributes of a well-written *SRS* and offers suggestions on how to organize an *SRS*.

Before the software requirements stage is complete, an SRS must be written. The SRS contains a complete description of the external behaviour of the software system. It is possible to complete the entire problem analysis before starting to write the *SRS*. The *SRS* serves a number of purposes depending on the person who wrote it.

The SRS could be written by a potential user (or customer) of a system or could be written by a developer of the system. The two possibilities create entirely different situation and establish entirely different purposes for the document. In the first case, the primary purpose of the SRS is to define user's needs.

On the other hand, an SRS written by a development organization as the first step of the software development process must be far more specific. This type of SRS is the subject of this unit and it performs an entirely different set of functions from the SRS that was described above. Its purpose is to provide a means of

- Communication among customers, users, analysts and
- Supporting system-testing activities designers.
- Controlling the evolution of the system.

A well-written SRS reduces the probability that the customer is disappointed with the final product. Disagreements between the customer and developer are worked out during the requirements stage and not during the testing stage, which would be more costly to correct. The SRS should be very specific about how the system will look externally to the system's environment.

The second purpose of the SRS is to serve as the basis for system testing and verification activities. The purpose of system testing is to stimulate the system with representative tests in order to show that the constructed system meets requirements. If the SRS is ambiguous or inconsistent, then testing is impossible.

The third purpose of the SRS is to help control the evolution of the software system. Let us assume that a software is either under development or has been deployed and a customer wants a certain function from the software.

How does anyone know if that is a new requirement or an old one ? The answer is to read the SRS and find out. If it is determined to be a new requirement, then the appropriate process to incorporate the customer's request is to update the SRS, update the design, update the code and so forth. The SRS serves as the definition and the only definition of what the software is supposed to do.

5.8 WHAT OUGHT TO BE INCLUDED IN THE SRS ?

A SRS must include a complete yet concise description of the entire external interface of the system with its environment, including other software, communication ports, hardware and human users. This includes two types of requirements namely behavioural or non-behavioural.

5.8.1 Behavioral Requirements

These requirements describe all the inputs of the system and outputs from the system as well as information concerning how the inputs and outputs interrelate.

- This description of how inputs map into outputs are called behavioral descriptions or operational specifications.

5.8.2 Non-behavioral Requirements

They define the attributes of the system as it perform its job.

- They include a complete description of the system's required levels of efficiency, reliability, security, maintainability, portability, visibility, capacity and standards compliance.
- Given the purpose of SRS, the following do not belong in a SRS.
- Project requirements (for example staffing, costs, schedules, milestones, activities, phases, reporting procedures)
- Designs
- Product assurance plan (for example, configuration management plans, verification and validation plans, test plans, quality assurance plans)

5.9 EXCLUSION OF PROJECT REQUIREMENTS FROM SRS

- The SRS and the project requirements have entirely different lifespans.
- The software product must be considered to be the SRS, design documentation, the code, test plans and so forth as the lifespan of SRS is same as the lifespan of the product.
- The life span of items like milestones, development cost and staffing is as long as the development of the project as it is only concerned with project development.

5.10 EXCLUSION OF DESIGN FROM SRS

- The advantage of packing requirements, designs, code, test and user documentation separately is the ability to define baselines in the development process that
- Signal completion of a major project milestone and thus denote process.
- Control change of the system.
- Requirement writers are choosen for their ability to analyse and specify, not for the ability to synthesize efficient designs.

5.11 EXCLUSION OF PRODUCT ASSURANCE PLANS FROM SRS

- The two reasons are same as that for excluding design from SRS.
- In particular, there is no reason to combine relatively unrelated subject in the same document, and the audience for product assurance plans is altogether different than the one for the SRS.

5.12 ATTRIBUTES OF HIGH QUALITY SRS

- To be a “perfect” SRS, it must have the following attributes.
- **Correctness** : A SRS is correct if and only if the requirements stated represents something of the system to be built.
- **Non-ambiguity** : A SRS is non-ambiguous if and only if every requirement stated has one and only one interpretation.
- **Verifiability** : A SRS is verifiable if and only if every requirement stated is verifiable.
- A requirement is verifiable if there exists some cost-effective, finite process by which a person or an automated process checks whether the software product meets the stated requirement.
- **Consistency** : A SRS is consistent if and only if no subset of stated individual requirements conflict.

Conflicts can be in terms of the following

- a) **Conflicting terms** : Two terms in the SRS that mean the same thing, but are used in different contexts.
- b) **Conflicting characteristics** : Such conflict occurs when the SRS seeks the product to have two different non-compatible characteristics.
- c) **Conflicting temporal characteristics** : Such a conflict occurs when in a SRS, two different requirements need the software to behave with different (conflicting) timing characteristics.
- **Understandability** : A trend that is possible in our vision of achieving correctness, verifiability, etc. would be to adopt a very formal approach in writing the SRS. That is, the formal methods include highly mathematical notations and other formalisms. While this would achieve the stated purposes, it would not be understandable to the common stakeholder, the client. Clients usually are experts in their own domains, not necessarily in computer science. Using formal methods do contribute to making the SRS unambiguous, correct, verifiable, etc. However, to make it understandable, we have to resort to less formal methods.
- **Modifiability** : A SRS is modifiable if its structure and content makes it easy to make changes in the requirements while preserving consistency and completeness. The minimal things that must be present for modifiability are a table of contents, index and cross-references. That enables suitable modifications in the relevant positions of the SRS should we make a change in the requirements.
- **Annotated** : A SRS that is annotated provides some significant guidance to the developers, the team and the organisation itself.
- **Traceability** : A SRS is traceable if the origin of each of its requirement is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

There are 4 types of traceability

- a) **Backward-from-requirements traceability** implies that we know why every requirement in the SRS exists. It implies that each requirement explicitly references its source in previous documents.

- b) **Forward-from-requirements traceability** implies that we understand which components of the software satisfy each requirement. It demands that each requirement in the SRS explicitly references a design component.
- c) **Backward-to-requirements traceability** implies that every software component explicitly references those requirements that it helps to satisfy. It implies that each requirement in the SRS has a unique name or reference number.
- d) **Forward-to-requirements traceability** implies that all documents that preceded the SRS can reference the SRs. Like backward-to-requirements traceability, this implies that each requirement in the SRS has a unique name or reference number.

Completeness :

A SRS is complete if it possesses the following 4 qualities.

- 1) Everything that the software is supposed to do is included in the SRS. This is the most difficult of the attributes to define. It is also the most difficult thing to detect violations of this completeness. A violation is difficult to detect because it implies that something is not in the SRS; it is not simple to find something that is not present by examining what is present. The only ones able to detect such an oversight or omission are those who own the problem to be solved by the software.

It is tempting to include the following sentence in the definition of completeness :
The SRS must specifically state those things that the software is not supposed to do. However, this becomes impossible to achieve considering the size of the environment with which the software isn't supposed to interact. For example, suppose in a telephone system we state the requirement that

If party A calls party B and party B is idle,
Then party's phone shall ring.

This seems perfectly reasonable. Later during testing, testers verify that the requirement has been satisfied by the system. However, what is to prevent the developer from satisfying the requirement by building a system that rings all phones in the telephone system whenever party A calls an idle party B ? To prevent this the requirements writer would be forced to write it as follows. If party A calls party B and party B is idle, then party B's phone shall ring and no other phones shall ring.

The problem here is that there may be a valid reason for party C's phone to ring at the same time ! An alternative is to write at the beginning of the SRS that, the system shall do the things stated in the SRS, and nothing else.

This is also a dead end because software does do things not stated in its requirements.

- 2) Definitions of the responses of the software to all realizable classes of input data in all reliable classes of situations are included. It is important to specify the responses to both valid and invalid inputs. This implies that for every system input mentioned in the SRS, the SRS specifies what the appropriate output will be.
- 3) All pages are numbered; all figures and tables are numbered, named, and referenced; all terms and units of measure are provided; and all referenced material and sections are present. This is completeness from a word processing perspective.

- 4) No sections are marked “To Be Determined (TBD).” Inserting the three letters “TBD” in a section of an SRS should be avoided whenever possible. When included, the TBD should be appended with a notation of who has responsibility for determining the contents and when the section will be completed.

Achieving all the attributes in a SRS is impossible :

For example, as we attempt to eliminate inconsistency and ambiguity (usually by reducing the natural language in the SRS language), the SRS becomes less understandable to the non-computer specialist. As we attempt to be absolutely complete, cost of the SRS increases and the document becomes extremely large and difficult to read. if we try to increase modifiability by eliminating all redundancy, the SRS becomes difficult to follow, and ambiguous. hence the only conclusion that we can reach is that a perfect SRS is not possible.

5.13 GENERAL FORMAT OF A SRS

A SRS must have some specific sections. In general a SRS can have the following format.

Section 1 : Product overview and summary.

Section 2 : Development, operating and maintenance environment.

Section 3 : External interfaces and dataflow

Section 4 : Functional requirements

Section 5 : Performance requirements

Section 6 : Exception handling

Section 7 : Early subsets and priorities in implementation.

Section 8 : Foreseeable modification and enhancement

Section 9 : Acceptance criteria

Section 10 : Design hints and guidelines

Section 11 : Cross reference index

Section 12 : Glossory of terms

Of course, individuals and organisations can and do have considerable variations in structuring SRS.

5.14 STANDARDS IN SRS

Over the years several standards have evolved which serve as guidelines for developing SRS. They also serve as indicators of best practices followed. Some of the Standards for SRS are given below.

- *IEEE 830 (by the IEEE society)*
- *DoD (by the Department of Defence, USA)*
- *NASA (by National Aeronautics and Space Administration)*

The IEEE 830 standard for SRS :

The following paragraphs provide the details of the IEEE 830 standard for writing SRS. It provides the skeleton of the document with some guidelines.

1. Introduction**i) Purpose of SRS**

- Delineate the purpose of the particular SRS
- Specify the intended audience for the SRS.

ii) Scope of product

- Identify the software product (s) to be produced by name ; eg. Host DBMS, Report Generator, etc.
- Explain what the software product (s) will, and, if necessary, will not do.
- Describe the application of the software being specified. As a portion of this, it should do the following.
- Describe all relevant benefits, objectives and goals as precisely as possible.
- Be consistent with similar statements in higher-level specifications if they exists.

iii) Definition, Acronyms, and Abbreviations

- Provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendixes in the SRS or by reference to other documents.

iv) Reference

- Provide a complete list of all documents referenced elsewhere in the SRS, or in a separate, specified document.
- Identify each document by title, report number (if applicable) date, and publishing organisation.
- Specify the sources from which the referenes can be obtained.
- This information can be provided by a reference to an appendix or another document.

v) Overview of the rest of SRS

- Describe what the rest of the SRS contains.
- Explain how the SRS is organized.

2. General description

This section should describe the general factors that affect the product and its requirements.

i) Product perspective

This subsection puts the product into perspective with other related products or projects.

- If the product is independent and totally self-contained, it should be stated here.
- If the SRS defines a product that is a component of a larger system or project then this subsection should do the following.
- Describe the functions of each compoent of the larger system or project, and identify interfaces.
- Identify the principal external intefaces of this software product.
- Describe the computer Hardware and peripheral equipment to be used.

A block diagram showing the major components of the larger system or project, interconnections, and external interfaces can be very helpful.

ii) Product functions

This subsection should provide a summary of the functions that the software will perform. For the sake of clarity, we can say that

- the functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.
- block diagrams showing the different functions and their relationships can be helpful. Remember, however, that such a diagram is not a requirement on the design of a product itself, it is merely an explanatory tool.

iii) User characteristics

This subsection should describe those general characteristics of the eventual users of the product that will affect the specific requirements. This subsection should not be used to state specific requirements or to impose specific design constraints on the solution.

iv) General constraints

This subsection of the SRS should provide a general description of any other items that will limit the developer's options for designing the system. These can include the following.

- Regulatory policies
- Hardware limitations; eg, signal timing requirements.
- Interfaces to other applications
- Parallel operation
- Audit functions
- Control functions
- Higher-order language requirements
- Signal handshake protocols; for example, XON - XOFF, ACK - NACK

This subsection should not be used to impose specific requirements or specific design constraints on the solution.

v) Assumptions and dependencies

This subsection of the SRS should list each of the factors that affect the requirements stated in the SRS. These factors aren't design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption might be that a specific operation system will be available on the hardware designated for the software product. If, in fact, the OS isn't available, the SRS would then have to change accordingly.

3. Specific requirements

i) Functional requirements

This section of the SRS should contain all the details the software developer needs to create a design.

The important points to be recognised here.

- Specific requirements should be organised in a logical and readable fashion.
- Each requirement should be stated such that its achievement can be objectively verified by a prescribed method.

i) Functional requirements 1

This subsection should specify how the inputs to the software product should be transformed into outputs. It describes the fundamental actions that must take place in the software.

For each class of function or sometimes for each individual function, it is necessary to specify requirements on inputs, processing, and outputs.

Introduction

This subparagraph should provide a description of the purpose of the function and the approaches and techniques employed. It should contain any introductory or background materials, which might clarify the intent of the function.

Inputs

This subparagraph should contain :

- A detailed description of all data input to this function to include :
 - * The sources of the inputs
 - * Quantities
 - * Timing
 - * The ranges of the valid inputs to include accuracies and tolerances.
- The details of the operator control requirements should include names and descriptions of operator actions, and console or operator positions.
- References to interface specifications or interface control documents where appropriate.

Processing

This subparagraph should define all of the operations to be performed on the input data and intermediate parameters to obtain the output. It includes specification of the following.

- Validity checks on the input data.
- The exact sequence of operations to include timing of events.
- Responses to abnormal situations, for example, overflow, communication failure and error handling.
- Parameters affected by the operations.
- Requirements for degraded operation.
- Any methods (eg. equations, mathematical algorithms, and logical operations) that must be used to transform the system inputs into corresponding outputs.

For example, this might specify :

- The formula for computing the income tax in a payroll package.
- A least squares curve fitting technique for a plotting package.

- A meteorological model to be used for a weather forecasting package.
- Validity checks on the output data.

Output

This subparagraph should contain.

- A detailed description of all data output from this function must include the following.
 - * Destinations of the outputs
 - * Quantities
 - * Units of measure
 - * Timing
 - * The range of the valid outputs is to include accuracies and tolerances.
 - * Disposition of illegal values
 - * Error messages
- References to interface specifications or interface control documents where appropriate.

In addition, for those systems whose requirements focus on input/output pairs and sequences of pairs.

Sequences will be needed when a system is required to remember its behaviour so that it can respond to an input based on that input and past behaviour; i.e., behave like a finite state machine.

ii) Functional requirements 2

- * Introduction
- * Inputs
- * Processing
- * Outputs

Functional requirements n

- * Introduction
- * Inputs
- * Processing
- * Outputs

b) External interface requirements

User interface

This should specify :

- The characteristics that the software must support for each human interface to the software product. For instance, if the user of the system operates through a display terminal, the following should be specified.
 - * required screen formats
 - * page layout and content of any reports or menus
 - * relative timing of inputs and outputs
 - * availability of some form of programmable function keys

- All the aspects of optimizing the interface with the person who must use the system. This may simply comprise a list of do's and don'ts on how the system will appear to the user. For instance, requirement for the option of long and short error messages. Like all others, these requirements should be verifiable and for example, a clerk typist grade 4 can do function X in Z minutes after 1 hour of training rather than a typist can do function X.

Hardware interfaces

This should specify the logical characteristics of each interface between the software product and the Hardware components of the system. It also covers such matters as what devices are to be supported, how they are to be supported and protocols. For instance, a terminal support might specify full screen support as opposed to line by line.

Software interfaces

This should specify the use of other required products (DBMS or OS) and interfaces with other application systems (linkage between an accounts receivable system and a general ledger system). For each required software product, the following should be provided.

- * Name
- * Mnemonic
- * Specification number
- * Version number
- * Source

For each interface, this part should

- * discuss the purpose of the interacing software as related to this software product.
- * define the interface in terms of message content and format (a reference to the document defining the interface is required)

Communication interfaces

This should specify the various interfaces to communiations such as local network protocols.

c) Performance requirements

This subsection should specify both the static and the dynamic numerical requirements placed on the software or on human interaction with the software, as a whole.

- Static numerical requiriements may include
 - * the number of terminals to be supported.
 - * the number of simultaneous users to be supported.
 - * no. of files and records to be handled.
 - * size of tables and files.

Static numerical requirements are sometimes identified under a separate section entitled capacity.

- Dynamic numerical requirements may include, for example, the numbers of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

All these requirements should be stated in measurable terms. For instance, “95% of the transactions shall be processed in less than 1 second”, rather than, “operator shall not have to wait for the transaction to complete”.

Note : Numerical limits applied to one specific function are normally specified as part of the processing subparagraph description of that function.

d) Design constraints :

Design constraints can be imposed by other standards, Hardware limitations, etc.

Standards compliance

This subsection should specify the requirements derived from existing standards or regulations. They might include the following.

- * Report format
- * Data naming
- * Accounting procedures
- * Audit tracing, for instance, specify the requirement for software to trace processing activity. Some applications need to meet minimum government or financial standards.

Hardware limitation

This subsection could include requirements for the software to operate requirements for the software to operate inside various Hardware constraints. For example, this could include the following.

- * Hardware configuration characteristics
(number of ports, instruction sets, etc.)
- * Limits on primary and secondary memory.

e) Attributes

There are a number of attributes that can place specific requirements on the software.

Availability

This could specify the factors required to guarantee a defined availability level for the entire system such as check-point, recovery and restart.

Security

This could specify the factors that would protect the software from accidental or malicious access, use, modification, destruction, or disclosure specific requirements in this area include the need to

- * utilize certain crypto graphical techniques.
- * keep specific log or history data sets.
- * assign certain functions to different modules.
- * restrict communications between some areas of a program.
- * compute checksums for critical quantities.

Maintainability

This could specify the requirements to ensure that the software could be maintained. For example, as a part of this

- * Specific coupling metrics for the software modules could be required.
- * Specific data/program partitioning requirements could be specified for micro devices.

Transferability / Conversion (Optional)

This could specify the user procedures, user interface compatibility constraints (if any) etc., required to transport software from one environment to another.

Caution (Optional)

It is important that required attributes be specified so that their achievement can be objectively verified by a prescribed method.

f) Other requirements

Certain requirements may, due to the nature of the software, the user, organization etc, be placed in separate categories such as below.

Database

This could specify the requirements for any database that is to be developed as part of the product.

This might include

- the types of information identified in the functional requirements
- frequency of use
- accessing capabilities
- data element and file descriptors
- relationship of data elements, records and files
- static and dynamic organization
- retention requirements for data.

Operations

This could specify the normal and special operations required by the user such as

- the various models of operations in the user organization such as user-initiated operations.
- periods of interactive operations and periods of unattended operations
- data processing support functions.
- backup and recovery operations.

Site adoption

- This could define the requirements for any data or initialization sequences that are specific to a given site, mission, or operational mode, for example, grid values, safety limits, etc.
- Specify the site or mission-related features that should be modified to adapt the software to a particular installation.

5.15 AN APPROVED FORMAT FOR SOFTWARE REQUIREMENTS SPECIFICATIONS(S.R.S)

Software Requirements Specifications(S.R.S)

for

<Project>

Prepared by <Author Name>

<Name of the Organization>

<Date of Creation>

Table of Contents

Table of Contents	ii
Revision History	ii
1. Introduction.....	1
1.1 Purpose.....	1
1.2 Document Conventions.....	1
1.3 Intended Audience and Reading Suggestions	1
1.4 Project Scope	1
1.5 References.....	1
2. Overall Description	2
2.1 Product Perspective.....	2
2.2 Product Features.....	2
2.3 User Classes and Characteristics.....	2
2.4 Operating Environment.....	2
2.5 Design and Implementation Constraints	2
2.6 User Documentation	2
2.7 Assumptions and Dependencies.....	3
3. System Features	3
3.1 System Feature 1	3
3.2 System Feature 2 (and so on).....	4
4. External Interface Requirements	4
4.1 User Interfaces	4
4.2 Hardware Interfaces	4
4.3 Software Interfaces	4
4.4 Communications Interfaces.....	4
5. Other Nonfunctional Requirements	5
5.1 Performance Requirements	5
5.2 Safety Requirements	5
5.3 Security Requirements	5
5.4 Software Quality Attributes	5
6. Other Requirements	5

Revision History

Name	Date	Reason For Changes (for an existing project)	Version

1. INTRODUCTION

1.1 Purpose

<Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.>

1.2 Document Conventions

<Describe any standards or typographical conventions that were followed when writing this SRS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

1.3 Intended Audience and Reading Suggestions

<Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SRS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.>

1.4 Project Scope

<Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies. If a separate vision and scope document is available, refer to it rather than duplicating its contents here. An SRS that specifies the next release of an evolving product should contain its own scope statement as a subset of the long-term strategic product vision.>

1.5 References

<List any other documents or Web addresses to which this SRS refers. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.>

2. OVERALL DESCRIPTION

2.1 Product Perspective

<Describe the context and origin of the product being specified in this SRS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SRS defines a component

of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.>

2.2 Product Features

<Summarize the major features the product contains or the significant functions that it performs or lets the user perform. Details will be provided in Section 3, so only a high level summary is needed here. Organize the functions to make them understandable to any reader of the SRS. A picture of the major groups of related requirements and how they relate, such as a top level data flow diagram or a class diagram, is often effective.>

2.3 User Classes and Characteristics

<Identify the various user classes that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the favored user classes from those who are less important to satisfy.>

2.4 Operating Environment

<Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and any other software components or applications with which it must peacefully coexist.>

2.5 Design and Implementation Constraints

<Describe any items or issues that will limit the options available to the developers. These might include: corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; parallel operations; language requirements; communications protocols; security considerations; design conventions or programming standards (for example, if the customer's organization will be responsible for maintaining the delivered software).>

2.6 User Documentation

<List the user documentation components (such as user manuals, on-line help, and tutorials) that will be delivered along with the software. Identify any known user documentation delivery formats or standards.>

2.7 Assumptions and Dependencies

<List any assumed factors (as opposed to known facts) that could affect the requirements stated in the SRS. These could include third-party or commercial components that you plan to use, issues around the development or operating environment, or constraints. The project could be affected if these assumptions are incorrect, are not shared, or change. Also identify any dependencies the project has on external factors, such as software components that you intend to reuse from another project, unless they are already documented elsewhere (for example, in the vision and scope document or the project plan).>

3. SYSTEM FEATURES

<This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product. You may prefer to organize this section by use case, mode of operation, user class, object class, functional hierarchy, or combinations of these, whatever makes the most logical sense for your product.>

3.1 System Feature 1

<Don't really say "System Feature 1." State the feature name in just a few words.>

3.1.1 Description and Priority

<Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>

3.1.2 Stimulus/Response Sequences

<List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>

3.1.3 Functional Requirements

<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate when necessary information is not yet available.>

<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>

REQ-1:

REQ-2:

3.2 System Feature 2 (and so on)

4. EXTERNAL INTERFACE REQUIREMENTS

4.1 User Interfaces

<Describe the logical characteristics of each interface between the software product and the users. This may include sample screen images, any GUI standards or product family style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. Define the software components for which a user interface is needed. Details of the user interface design should be documented in a separate user interface specification.>

4.2 Hardware Interfaces

<Describe the logical and physical characteristics of each interface between the software product and the hardware components of the system. This may include the supported

device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.>

4.3 Software Interfaces

<Describe the connections between this product and other specific software components (name and version), including databases, operating systems, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the system and going out and describe the purpose of each. Describe the services needed and the nature of communications. Refer to documents that describe detailed application programming interface protocols. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating system), specify this as an implementation constraint.>

4.4 Communications Interfaces

<Describe the requirements associated with any communications functions required by this product, including e-mail, web browser, network server communications protocols, electronic forms, and so on. Define any pertinent message formatting. Identify any communication standards that will be used, such as FTP or HTTP. Specify any communication security or encryption issues, data transfer rates, and synchronization mechanisms.>

5. OTHER NONFUNCTIONAL REQUIREMENTS

5.1 Performance Requirements

<If there are performance requirements for the product under various circumstances, state them here and explain their rationale, to help the developers understand the intent and make suitable design choices. Specify the timing relationships for real time systems. Make such requirements as specific as possible. You may need to state performance requirements for individual functional requirements or features.>

5.2 Safety Requirements

<Specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product. Define any safeguards or actions that must be taken, as well as actions that must be prevented. Refer to any external policies or regulations that state safety issues that affect the product's design or use. Define any safety certifications that must be satisfied.>

5.3 Security Requirements

<Specify any requirements regarding security or privacy issues surrounding use of the product or protection of the data used or created by the product. Define any user identity authentication requirements. Refer to any external policies or regulations containing security issues that affect the product. Define any security or privacy certifications that must be satisfied.>

5.4 Software Quality Attributes

<Specify any additional quality characteristics for the product that will be important to either the customers or the developers. Some to consider are: adaptability, availability,

correctness, flexibility, interoperability, maintainability, portability, reliability, reusability, robustness, testability, and usability. Write these to be specific, quantitative, and verifiable when possible. At the least, clarify the relative preferences for various attributes, such as ease of use over ease of learning.>

6. OTHER REQUIREMENTS

<Define any other requirements not covered elsewhere in the SRS. This might include database requirements, internationalization requirements, legal requirements, reuse objectives for the project, and so on. Add any new sections that are pertinent to the project.>

This is followed by a live example on “Library Management System”.

5.16. SRS : A LIVE EXAMPLE

Requirements Document For Library Management System

1. Introduction

1.1 Purpose of this document

The purpose of this document is to provide an overview of the recommended procedures and the available resources for developer; create the total number of requirements of the Library management system.

1.2 Scope of this document

The requirements document should communicate a complete, consistent, and feasible statement of what is needed in the system. This document represents the following efforts to define requirements: functional requirements, interface requirements, performance requirements, design constraints, Preliminary Object-Oriented Domain Analysis, operational scenarios and others.

In the process of requirements elicitation there are different group of participants:

- Project customer: Mahavir Institute of Engineering and Technology (MIET), Bhubaneswar (present also user requirements)

1. Prof. Sabyasachi Rout
 2. Prof. Sanjay Kumar Mohapatra
 3. Dr. Birendra Kumar Bindhani
 4. Dr. Satyajit Sahu
 5. Mr. Dillip Kumar Parida
- Requirements Engineer
 - Prof.(Dr.)J.P Tripathy
 - Designers
- Prof. Tanmaya Kumar Das
Prof. Dillip Kumar Mahapatra

Schedule for the requirements elicitation process. (Table 2)

Table 1. Schedule

Deliverables	Activities	Starting data	Deadline
Requirements document	Familiarization with the description of a desired system	March 25	April 4
	Creation of a draft requirements document	April 4	April 12
	Discussing draft requirements with customer	April 12	April 25
	Changing requirements Document	April 12	May 3
	Final version of requirements document		May3

1.3 Overview

The product of the requirements elicitation process is requirements document, which include functional requirements, interface requirements, performance requirements, design constraints, Preliminary Object-Oriented Domain Analysis, operational scenarios and others requirements.

1.4 Business Context

The organisation, Mahavir Institute of Engineering and Technology (MIET), Bhubaneswar finances the development of this project. Budget of the financing is Rs. 100000 (Rupees one lakh only)

2. GENERAL DESCRIPTION

Product Functions

System should be able to handle all major library activities: check out and check in library's items (book, magazines, newspaper), search for books in library's catalogue and provide information about books availability and positions, maintain users database, send overdue notes and print bills for overdue and loss of a book. Additionally automatic book prolongation through Internet should be possible. Program has to possess friendly user interface and to be easy to use.

2.1 User Characteristics

The Library Management System will be used by different persons of Organisation MIET, Bhubaneswar. Expected expertise:

- Librarian: Basic experience of work with different applications under Windows 2007 operating system; knowledge of usage web-browsers and work in the Internet.
- User (student): Knowledge of usage web-browsers and work at the Internet, experience with different applications under Windows 2007.

2.2 User Problem Statement

This section describes the features of the user community, including their expected expertise with software systems and the application domain, which can occur in really live. Such a way, expected community problems should be analysed and allowed.

During the usage of Library Management System some problem can occur. The main problem *can be* interactions between user and Library Management System through the user interface. User interaction is random. It is random in that it may happen at any time. Also there exist various type of user interaction including “Add items”, “Remove items”, “Search”, and others. Surveys forms can be are not simple for usage and inappropriate. Survey result very big – it is not suitable for user perception. Access to the Library Management System is difficult: for user to the search system, for Librarian to the management system.

2.3 User Objectives

The main objectives of the user are the following:

1. Librarian:
 - Library book catalogue
 - Friendly user interface of the program
 - Easy organisation of students information base (add/delete/update)
 - The search system for book and students.
 - Automated checking availability of books and theirs status.
 - Automatic book prolongation through Internet should be possible
 - Automated notice to student about deadline of book by e-mail.
 - Bill creation in case if book is lost or overdue by a student
- 1.1 Other*:
 - Electronic part of the library: e-books database, video and media files.
 - Update through Internet e-books, media-files and info providing of new books
2. Student:
 - Possibility to renew book through the Internet
 - Easy-to-use GUI
- 1.1 Other*:
 - Possibility to use this system from remote host for access to the electronic part of the library.

Note: * - These requirements will not be done in original project, due to deficiency of time and resources.

2.4 General Constraints

1. Operation system for the program is Windows 2000.
2. Screen resolution is 1024*768
3. For remote access two type of browsers can be used: IExplorer V4.X and higher and Netscape V4.7 and higher.

3. FUNCTIONAL REQUIREMENTS

This section represents the functional requirements. Functional requirements describe the possible effects of a software system, in other words, what the system must accomplish.

Traditionally, Requirements Engineering methods have been based on a monolithic and centralist vision of the requirements analysis and development process. Their underlying philosophy is: *one* method and *one* modelling notation should be used to address *one* problem. So, the fact that there are many actors in the RE process, each with a particular view of the problem that is not always consistent with the particular views of other actors, is overlooked. So is the fact that each modelling notation used enables the expression of some particular features (control, function, etc.) but disables, or prevents others. Surprisingly, no attention is paid to the fact that conflict is unavoidable in interactions between people (or agents) that, inevitably, have different views on a given situation. [1]

Viewpoints may be used during the early stages of requirements engineering process as a structuring mechanism for requirements elicitation and analysis.

General viewpoints of different people represent functional requirements and are lead as follows:

Name1	Director
Focus	Focus on to get the software product in time and easy to use.
Concerns	Program should be simple for use
Source	Customer's requirements document
Requirements	1. Program must be easy to educate staff to use it (help, friendly user interface) 2. Deadline for requirements documentation is 12.04.2010.

Name2	Librarian
Focus	Focus on the simplicity of usage
Concerns	Program should simple in use and has friendly interface.
Source	Customer's requirements document
Requirements	1. The system should have library book catalogue. (Possibility to add, remove, update information) 2. Friendly user interface of a program 3. Easy organization of student information base. 4. The search system for books and students 5. Automated checking availability of books and deadlines for returning 6. Automatic book prolongation through Internet 7. Automatic notice to student about deadline of book by 8. e-mail 9. Bill creation in case if book is lost or overdue by a student.

Name3	Financial manager
Focus	Focuses on saving money and full financial documentation
Concerns	Fines creation and documentation.
Source	Customer's requirements document, Project plan
Requirements	1. Program must record fines if book is lost or overdue by a student 2. Total cost of the program must fit Rs. 80,000/- 3. Easy and cheap to maintenance 4. Automated creation of financial reports (bills)

Name4	Student
Focus	Focuses on easy and suitable to usage
Concerns	Renew books and the easiest program
Source	Customer's requirements document
Requirements	1. Possibility to renew books through the Internet 2. Easy-to-use Graphical User Interface

Name5	System administration
Focus	Focus on Technical support
Concerns	Support the system
Source	Customer's requirements document, Project Plan
Requirements	1. Operating system for the program is Windows 2000 2. Automatic installation program must be included in distributive 3. Software must fit in a 1024 * 768 pixels screen the usage is possible without extensive scrolling all the time.

Name6	Requirements Engineer
Focus	Focus on suitable of usage
Concerns	Input of information
Source	Customer's requirements document, project plan
Requirements	1. Input the data items should be through the optical barcode scan system. (each book has to barcode, which provides information about the book) 2. Software should divide into different parts for the librarian management and user search part. 3. Student should has login that is student ID to renew the book through the Internet 4. Librarian should have login and password to login

4. INTERFACE REQUIREMENTS

4.1 User Interfaces

In this project User Interface plays the most important role for communication between user and software product. The interface should present and acquire information in a consistent fashion. This implies that:

1. All visual information is organized according to a design standard that is maintained throughout all screen displays;
2. Input mechanisms are constrained to a limited set that are used consistency throughout the application.
3. Mechanisms for navigating from task to task are consistency defined and implemented.

Following by these considerations can be given GUI requirements.

First, GUI requirements for librarian management part:

1. Login User Interface has to contain:
 - o Login dialog to input login and password
2. Library management system application has to contain help, educating the stuff (librarians). Help system contains:
 - o List of content should be alphabetically ordered
 - o Help information immediately.
3. Information about a book and add new book. Interfaces contains:
 - o Title of the book
 - o Author's name
 - o Status1 of the book (Charged, Not charged)
 - o Status2 of the book (Shows availability to lend or to use it only in reading-room)
 - o Data of publishing
 - o Number of items
 - o Number of available items
 - o Data of the end of the first deadline (the first data of returning)
 - o Library call number of the book
 - o These information can be correct and update
 - o These information can be removed from the system
 - o Allows to get help information
4. Information about the student. It should has the following information:
 - o Name of the student
 - o Student ID
 - o List of given items
 - o Author's name of the items

- o Data of begin time (when the book was taken)
 - o Deadline data
 - o If Deadline data was occurs, the note of data should be in red colour
 - o Given books can be removed from the list
 - o The list can be added
 - o Renew operation can be marked
 - o Information can be update
 - o Allows to get help information
5. General information. Interface should contain:
- o Request for the search a student information (by the student ID)
 - o Request for the search a book information
 - o Adding new items
 - o Adding the students
 - o Allows to get help information

GUI requirements for student's part:

Book searcher's User Interface should be implemented as a web-based interface. It contains the following requirements:

1. General. Interface consist of:
 - o Field for the input user's information
 - o Should be choosing of the search criteria (key words, title, author's name, Call number)
 - o List of searched books (10 items) and link to next
 - o Possibility of login to renew books (login = student ID)
 - o Possibility to clear field of request
 - o Search initiating
2. Help menu. Help system contains:
 - o Should be list of content alphabetically of content
 - o Help information immediately.
3. Users personal information. It contains:
 - o List of given books
 - o Title of the books
 - o Author's name of the given books
 - o Items, which should be renewed can be marked
 - o Deadline data
 - o Text of passed deadline's data should be in red colour

4.2 Hardware Interfaces

Interface for optical scanner of barcode should be IBM AT/PS2 compatible with DIN 5 pin.

5. PERFORMANCE REQUIREMENTS

Library management system software Provides an efficient indicators by the conditions: Windows platform. Minimal requirements to the system: Pentium-IV, 2.6 Ghz. MHz, RAM 512Mb.

6. DESIGN CONSTRAINTS

6.1 Standards Compliance

1. Library management system has to contain two parts: Library management part and user's search part.
2. User's search part should be implemented as web-based software.
3. Software must fit in a 1024 * 768 pixels screen the usage is possible without extensive scrolling all the time.
4. Installation of the program must be included in distributive
5. For remote access two type of browsers can be used: IExplorer V4.X and higher and Netscape V4.7 and higher.

6.2 Hardware Limitations

INTEL P IV, 2.6 Ghz. MHz, RAM 512Mb RAM.

7. OTHER NON-FUNCTIONAL ATTRIBUTES

7.1 Security

Priority: High

- Confidentiality – extent to which data and processes are protected from unauthorized disclosure
- Integrity – extent to which data and processes are protected from unauthorized modification
- Availability – extent to which data and processes are protected from denial of service to authorized users

7.2 Reliability

Priority: High

- Non-deficiency – degree to which SW does not contain undetected defects
- Fault tolerance – degree to which software will continue to work without system failure that would cause damage to users or property; also, the degree to which SW includes degraded operation and recovery functions

7.3 Maintainability

Priority: High

–corrective - finding and fixing faults

–adaptive - modifying software for a changed environment

- hardware is changed
 - new functionality is added
- rewrite documentation or comments
- renaming a variable or routine

7.4 Portability

Priority: Medium

- Hardware Independence – degree to which software does not depend on specific hardware requirements
- Software Independence – degree to which software does not depend on specific software environments
- Install ability – effort to adjust software to new environment

7.5 Reusability

Priority: Medium

- Generality – extent to which software can be reused across applications
- Self Descriptiveness – ease with which a software component can be understood and used

8. PRELIMINARY OBJECT-ORIENTED DOMAIN ANALYSIS

This section provides the preliminary class candidates. Following classes should be represented:

1. Name: Student

Attributes:

- o Name
- o Student ID
- o List of given items

Operations:

- o Take book
- o Renew book
- o Search for book
- o Return book

2. Librarian

Attributes:

- o Name
- o Login
- o Password

Operations:

- o Search for book
- o Search for student

- o Add book
- o Remove book
- o Update book
- o Generate bill

3. Book

Attributes:

- o Title
- o Author's name
- o Status of the book
- o Number of items
- o Number of available items
- o Data of the first deadline
- o Library call number

4. Bill

Attributes:

- o Name of the user
- o Sum of all fines
- o List of items

9. OPERATIONAL SCENARIOS

This section should describe a set of scenarios that illustrate, from the user's perspective, what will be experienced when utilizing the system under various situations. In the broad sense, a scenario is simply a proposed specific use of the system. A set of scenarios is represented below.

Name	Run Library Management system
Actors	Librarian and Software
Preconditions	Librarian initiated invoke process of application
Description	Library Management system is running
End Result	Show login dialog

Name	Login to Library management system
Actors	Librarian and Software
Preconditions	Librarian input all asked information
Description	Librarian login to the system
End Result	Show general menu

Name	Search of the user
Actors	Librarian and Management system
Preconditions	Librarian input student's information (student ID)
Description	Student is found at the system
End Result	Show information about student

Name	Search of the book
Actors	Librarian and Management system
Preconditions	Librarian input book's information (scan barcode)
Description	Book info is found at the system
End Result	Show information about the book

Name	Add new books to the database
Actors	Librarian and Management system
Preconditions	Librarian initiated add new book
Description	Invoke the form to add new book
End Result	Show dialog add new book

Name	Add new student to the database
Actors	Librarian and Management system
Preconditions	Librarian initiated add student
Description	Invoke the form to add student
End Result	Show dialog add student

Name	Updating information about book
Actors	Librarian and Management system
Preconditions	Librarian changes corresponding fields in form
Description	Information about the book will be updated
End Result	Show main dialog

Name	Remove the book from database
Actors	Librarian and Management system
Preconditions	Librarian initiated remove book
Description	Book will be removed from database
End Result	Show main dialog

Name	Add book for student
Actors	Librarian and Management system
Preconditions	Librarian initiated add new book (at form about student)
Description	Librarian adds new book for student
End Result	Show Information about student (updated list of items)

Name	Remove book for student
Actors	Librarian and Management system
Preconditions	Librarian chosen necessary for deleting books and initiated remove book (at form about student)
Description	Show Information about student (updated list of items)
End Result	Show Information about student (updated list of items)

Name	Renew book
Actors	Librarian and Management system
Preconditions	Librarian chosen necessary for renewing books and initiated renew book (at form about student)
Description	Attempt to renew selected books will be realized
End Result	Show Information about student (updated list of items)

Name	Using help
Actors	Librarian and Management system
Preconditions	Librarian initiated help option
Description	Librarian uses help to solve some problem concerning work with the program
End Result	Show help

Name	Exiting the help
Actors	Librarian and system
Preconditions	Librarian initiated the exit process
Description	Librarian return back to the search interface
End Result	Show general search interface

Name	Exiting the Library management system
Actors	Librarian and system
Preconditions	Initiate exit process
Description	Exit the application process
End Result	Exit the program

Name	Using help
Actors	Student (search book) and system
Preconditions	Student initiated help option
Description	Student uses help to solve some problem concerning work with the program
End Result	Show help

Name	Exiting the help
Actors	Student and system
Preconditions	Student initiated the exit process
Description	Student return back to the search interface
End Result	Show general search interface

Name	Book search
Actors	Student and system
Preconditions	Student set all parameters for search (criterion, data[title, author, ...])
Description	Search books and look the result
End Result	Show General search interface

Name	Clear
Actors	Student and system
Preconditions	Student initiated clear process
Description	The requested field is cleared
End Result	Show General search interface

Name	Show current state
Actors	Student and system
Preconditions	Student input login (Student ID) and login
Description	Student will see your own current state
End Result	Show Information about the student (updated)

Name	Renew book
Actors	Student and system
Preconditions	Student chosen necessary for renew books and initiate renew process
Description	Some book renew, another can be not renew
End Result	Show Information about the student (updated)

Name	Exiting the student Information
Actors	Student and system
Preconditions	Student initiated the exit process
Description	Student return back to the search interface
End Result	Show general search interface

11. PRELIMINARY BUDGET

Initial budget for the project is Rs. 80,000/-

12. APPENDICES

12.1 Definitions, Acronyms, Abbreviations

SUMMARY

- The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification, requirements validation and requirements management.
- Requirements elicitation and analysis is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organisation, requirements negotiation and requirements documentation.
- Different stakeholders in the system have different requirements. All complex systems should therefore be analysed from a number of viewpoints. Viewpoints can be people or other systems that interact with the system being specified, stakeholders who are affected by the system, or domain viewpoints that constrain the requirements.
- Social and organisational factors have a strong influence on system requirements and may determine whether the software is actually used.
- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability. Requirements reviews and prototyping are the principal techniques used for requirements validation.
- Business, organisational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.
- The requirements management process includes management planning, where policies and procedures for requirements management are designed, and change management, where you analyse proposed requirements changes and assess their impact.
- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out. Domain requirements are functional requirements that are derived from characteristics of the application domain.
- Non-functional requirements constrain the system being developed and the development process that should be used. They may be product requirements, organisational requirements or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- User requirements are intended for use by people involved in using and procuring the system. They should be written using in natural language, with tables and diagrams that are easily understood.
- System requirements are intended to communicate, in a precise way, the functions that the system must provide. To reduce ambiguity, they may be written in a structured form of natural language supplemented by tables and system models.

- The software requirements document is the agreed statement of the system requirements. It should be organised so that both system customers and software developers can use it.
- The IEEE standard for requirements documents is a useful starting point for more specific requirements specification standards.

EXERCISES

1. What do you mean by the term requirement? Explain the process of determining the requirements for a software based system.
2. Discuss the significance and use of requirement engineering. What are the problems in the formulation of requirements?
3. What are crucial process steps of requirement engineering? Discuss with the help of a diagram.
4. Describe the various steps of requirements engineering. Is it essential to follow these steps?
5. Explain the importance of requirements. How many types of requirements are possible and why?
6. What do you understand with the term “requirements elicitation”? Discuss any two techniques in detail.
7. List out requirements elicitation techniques. Which one is most popular and why?
8. Explain the requirements elicitation and analysis process model with the help of a suitable diagram.
9. What do you understand by requirements management? What are the steps of planning in requirement management?
10. What do you understand by traceability of requirement? Why is traceability important?
11. What is SRS? What are the components of SRS?
12. Discuss the characteristics of a good SRS document.
13. What are the different Components of Software Requirement Specification (SRS)?
14. Write short notes on:
 - (i) SRS validation
 - (ii) Components of SRS
 - (iii) Characteristics of SRS
15. List the seven desirable characteristics of a good Software Requirement Specification (SRS) document.
16. Discuss the organization of a SRS. List out some important issues of this organization?
17. Explain the IEEE standards for SRS.
18. According to IEEE standards what are the several specific benefits which a good SRS should provide?

Software Design & Coding

6.1 INTRODUCTION TO SOFTWARE DESIGN

The term design in a broad sense, may be

- a final plan
- a proposal
- a blue print
- a drawing
- a model or
- a description

which is implemented in a form of final product of the design process.

For example, to make a house, the owner needs a '*blue print*' of the house by a designer as per his / her choice or requirement. We may call it as a document shows a detailed plan of the house i.e. *how to build* ?

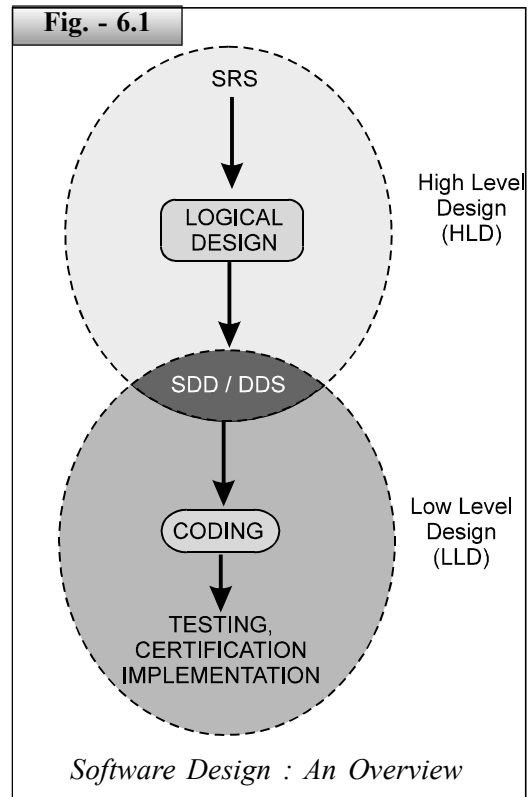
Software design can be broadly termed as a creative process, where customer requirements, business needs, and technical considerations, all come together in the formulation of a product or a software system.

6.2 DEFINITIONS

Software design can be defined as :

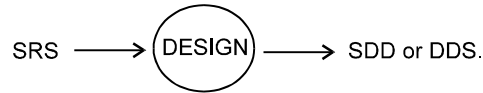
- an activity of specifying the nature and composition of a software system satisfying the client's needs and desire, subject to the design constraint.
- a significant phase in the field of software development (through *software engineering principles*), where the designer plans '*how*' to produce a software

Fig. - 6.1

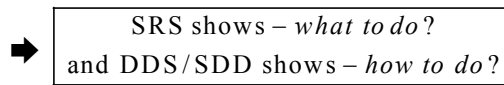


product / system in order to make it functional, reliable and reasonably easy to understand, modify and maintain.

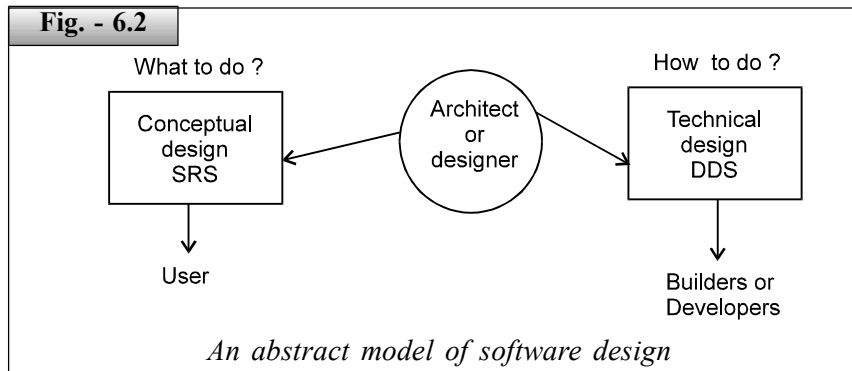
As the design phase follows the requirement and analysis phase of SDLC (*Software Development Life Cycle*), the SRS is treated as an input to the design phase and is transformed into SDD or DDS (Software Design Document or Design Document Specification) i.e.



So, the software design can also be defined as an interactive process that transforms the SRS into DDS.



More precisely, it can be represented by the following model.



6.3 DESIGN PROCESS

Software design is an interactive process through which requirements are translated into a “*blue print*” for constructing the software product. Initially, the blue print describes the entire view of software. That is, the design is represented at a high level of abstraction (i.e. a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioural requirements)

As design iteration occurs, subsequent refinement results with the design representations at much lower levels of abstraction. Through out the design process, the quality of the design is assessed with a series of *formal technical overviews*.

Generally, the software design process can be decomposed mainly into three levels/ phases of design.

- Interface Design
- Architectural Design
- Detailed Design
(also called *Data Design*)

6.3.1 Interface Design

The blueprint of a house as prepared by the architect (*during architectural design*) is not completed without the representation of doors, windows, connections for water supply, electricity, telephone etc. Likewise, the interface design for a software product / system describes :

- interfaces between the software system and external entities. e.g. other systems.
- interfaces between user and the software system.

Interface design proceeds at a high level of abstraction w.r.t the inner working of the system priorities are given on the dialogue between the software system and the users, devices and other systems with which it interacts. The uses, devices and external systems are called *agents*.

Interface design for a software product should usually includes the information details like,

- Precise description of events in the environment, or messages from agents, to which the system must respond.
- Description of events or message that the system must produce.
- Specification of data, data formats coming into and going out of the system.
- Specification of protocols for ordering and timing relationship among events.

6.3.2 Architectural Design

Architecture of a system can be viewed as “the overall structure of the system and the ways in which that structure provides conceptual integrity for the system.

In case of *software*, *architecture* tends to the structure or organisation of program components (*modules*), the manner in which these components interact, and the structure of data that are used by the components.

The architectural design for a software product is similar to the floor of a house, that shows the overall layout of the rooms, their size, shape, relationship and the doors and window that allow move into and out of the rooms. As the floor plan gives an overall view of a house, likewise the architectural design shows the overall view of the software product.

The highlighted activities under this architectural design are :

- Overall decomposition of the software into major components (modules)
- Identifying the functional responsibilities for the components.
- Component interfaces
- Identification of the component properties like, scale, performance, resource consumption, reliability etc.
- Communication and interaction between components.

Architectural design does not provide the details of the internal structure of the program components or modules. It can also be called as “*Black Box-View*” of a software system.

6.3.2.1 Assessment of Architectural Design

The software design process results with number of architectural alternatives, what need to be assessed to determine which one is most appropriate to used in.

Some of the assessment policies are discussed here.

a) *Trade-off (Compromise) analysis method :*

This method is being developed by the Software Engineering Institute (SEI), and establishes an iterative evaluation process for software architectural design. The activities under this method are :

- Representation of the system from user's point of view through use-cases.
- The stake-holder concerns must be also addressed.
- Description of the architectural styles that have been chosen to address the scenario and requirements.
- Evaluation of quality attributes i.e. reliability, performance, security, flexibility, maintainability, testability, reusability and inter-operability.
- Identification of the sensitivity of quality attributes of an architectural design by making small changes in the architecture.
- Finally, find out the trade-off points to which multiple attributes are sensitive. for example, the performance of a client-server architecture is highly sensitive to the number of servers. So the trade of point is *number of servers*.

b) *Architectural Complexity :*

Assessment of overall architectural complexity of a proposed software system is the measure of dependencies between the components within the architecture.

There are three types of dependencies :

- i) Sharing dependencies : for two components say u & v, if u and v refer to same global data, then there exists a shared dependencies between them i.e. between (u & v).
- ii) Flow dependencies : for two components u & v, if u must complete before the control flows into v, or if u communicates with v by parameters, then there exists a flow dependencies between u and v.
- iii) Constrained dependencies : it represents the relative flow of control among set of activities. e.g. for two components u and v, if u and v can't execute at the same time (mutual exclusion), then there exists a constrained dependencies between u and v.

c) *Architectural Description Language : (ADL)*

This language provides a semantic and syntax for describing a software architecture. An ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks and represents the interfaces between components.

It is the most effective assessment tool for architecture. Some of the ADLs are listed below.

Unicon - defines software architecture in terms of abstractions.

Aesop - address the re-use architecture.

Wright - static checks to determine the consistency and completeness of an architecture.
Acme - Second generation ADL etc.

6.3.3 Detailed Design

It is the specification of the internal elements of all major system components, component structure, properties, relationship, processing and their algorithms and data structures.

The detailed design entertains :

- Decomposition of major system components into program units.
- Fixing of functional responsibilities for program units.
- Unit interfaces.
- States and change of states of units.
- Exchange of data and control among program units.
- Scope and visibility of program units.
- Algorithms and data-structures.

6.4 DESIGN CHARACTERISTICS

There are three characteristics that guide towards the evaluation of a good design.

- The design must implement all of the explicit requirements contained in the analysis (SRS) model, and must entertain all the implicit requirements as desired by the customer.
- The design must be readable, easy to understand for those who use it for coding as well as testing.
- The design should provide a complete picture of the software, addressing the data, functional and behavioural domain from an implementation point of view.

6.5 CRITERIA FOR QUALITY DESIGN

In order to evaluate the quality of a design representation (*SDD* or *DDS*), some technical criterion are established for good quality design.

- i) A design should exhibit an architecture, which has been created using some standard predefined patterns, is composed of components that exhibit good design characteristics and can be implemented in a way that facilitates the implementation and testing process.
- ii) A design should be modular; i.e. the software should be logically partitioned into subsystems.
- iii) A design should contain distinct representation of data, architecture, interface and components.
- iv) A design should model with data structures that are appropriate for the classes to be implemented and are derived from recognized *data patterns*.
- v) A design should show the components that exhibit independent functional characteristics (*cohesiveness*)
- vi) A design should lead to interfaces that reduce the complexity of connections between components and with the external environment (*coupling*).

- vii) A design should be derived through an interactive method and referenced with the information (SRS) obtained during software requirement and analysis.
- viii) The notations used in the design should be very clear and effective.

All these above criteria lead to add the software product with quality attributes like functionality, usability, reliability, performance and maintainability etc.

6.6 PRINCIPLES OF DESIGN

The basic principles for software design are listed below :

- ☛ The design process should include the examination of alternative approaches. It is the responsibility of a good software designer to look for alternatives and secondary approaches, and do not limit oneself to a single approach. Each alternative should be determined on the basis of requirements, available resources and design concepts presented for the tasks.
- ☛ There should be a way of tracing the design concept back to the analytical model.
- ☛ Time being a precious commodity, a designer should invest his / her time in representing new ideas as well as integrating already existing patterns into the present design being created.
- ☛ The intellectual distance between the problem (as it exists) and the software needs to be reduced to a minimum. This means that the structure of the software should be as much like the structure of the problem domain as possible.
- ☛ The design should be uniform and have integrity. It means that rules pertaining to style and format required to be set before the commencement of the design process. A design can be said to be integrated when interfaces between various design components are defined properly.
- ☛ We must remember that *“design” is not “coding”* and *“coding” is not “design”* also. Keeping this in mind, the level of abstraction should always be maintained at a slightly higher plane than the source code.
- ☛ We should check for quality as the design is being generated, i.e. during the design process, not after the design has been completed.
- ☛ A review process is must for reducing the conceptual error in the design.

These above principles are revised and strictly implemented / followed-up during the software design phase (Practically) in the terms of :

● modularity and partitioning,	● coupling
● cohesion	● span of control
● size of module (module size)	● shared use.

6.6.1 Modularity and Partitioning

Each system should consists of a hierarchy of modules. Lower level modules are generally smaller in scope and size compared to higher-level modules and serve to partition processes into separate functions.

6.6.2 Coupling

This is a measure of the interconnection between different modules within a software structure. It signifies, modules should have little dependence on other modules in a system.

The coupling between two modules indicates the degree of interdependences between them. If two modules interchange large amount of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The *interface complexity* is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

- **Highly Coupled**

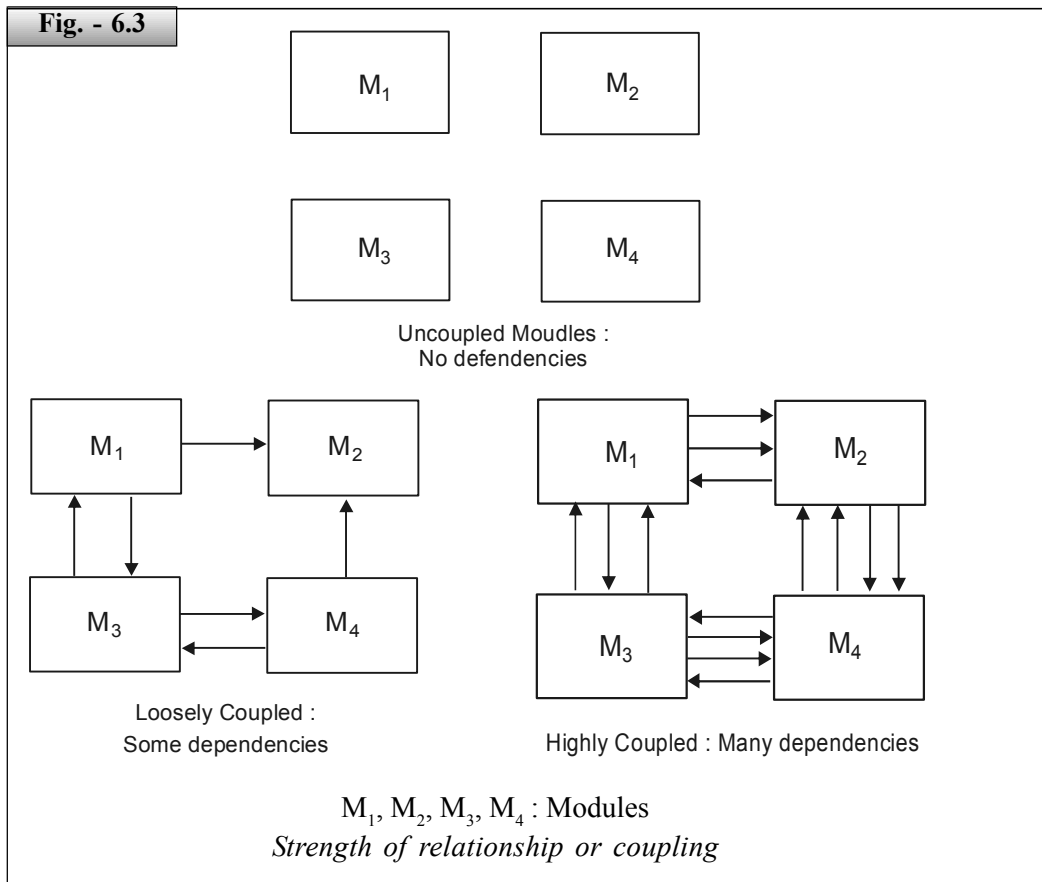
When the modules are highly dependent on each other then they are called *highly coupled*.

- **Loosely Coupled**

When the modules are dependent on each other but the interconnection among them is weak, then they are called *loosely coupled*.

- **Uncoupled**

When the different modules have no interconnection among them then it is called *uncoupled module*.



Objectives of Coupling

To maximize independence between modules by minimizing or reducing module coupling in the system to be designed or developed.

Coupling is dependent on three things.

- The complexity of the interface between two different modules.
- The point at which entry to a module or a reference to it is made.
- The nature of the data passing across the interface.

Types of Coupling

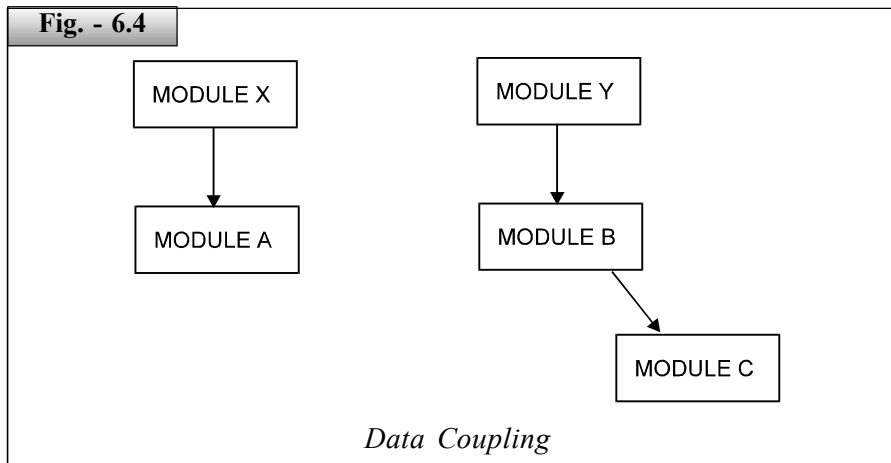
Different types of coupling are content, common, control, stamp and data. The strength of coupling from lowest coupling (best) to highest coupling (worst) is in Table : 6.1:

Table : 6.1 - Types of Coupling

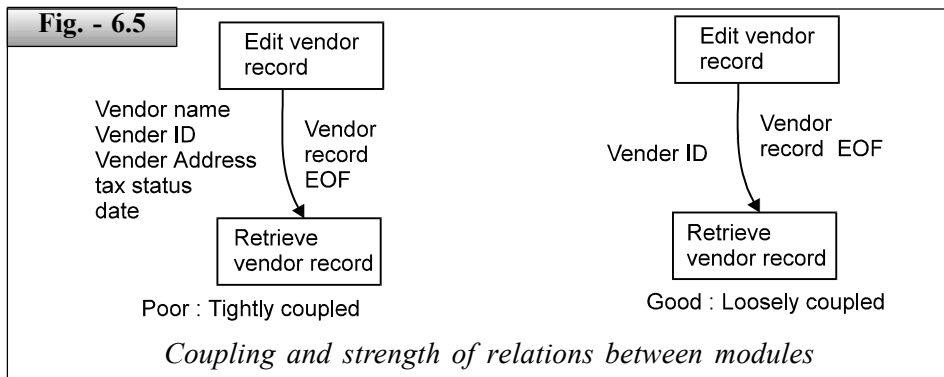
Data coupling	Best
Stamp coupling	↑
Control coupling	
External coupling	
Common coupling	
Content coupling	Worst

i) Data Coupling

In software, simpler levels of connectivity among different modules ensure simplicity and efficiency in the software. Therefore it is only natural that we should always look for low levels of coupling;



The above figure-6.4 shows different modules with low coupling existing within a software structure. Here module A and module B are subordinate to different other modules.



They are both unrelated and hence there is no direct coupling.

Module C, on the other hand, is related to module B and is accessed through a conventional argument list through which data is passed. If the argument list is simple, then low coupling, also *called data coupling*.

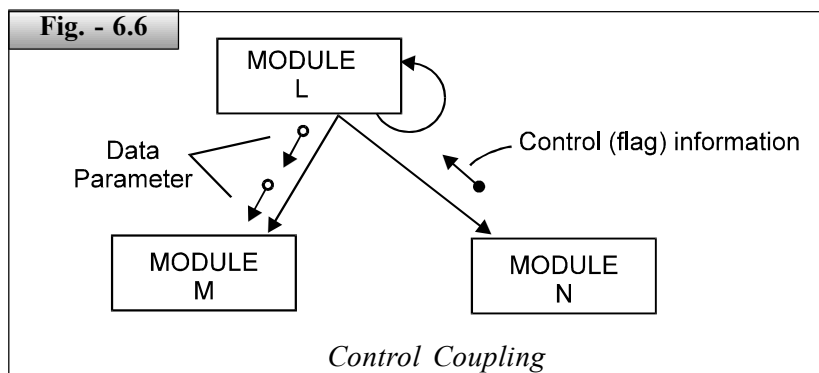
ii) Stamp Coupling

Two modules are said to have *stamp coupling*, if they communicate by using a composite data item such as a record, structure, and object etc.

When module passes non global data structure or entire data-structure to another module, they are said to be stamp coupled. *For example, passing a record in PASCAL or structure variable in C or object in C++ language to a module.*

iii) Control Coupling

Control coupling is seen in moderate levels. Control coupling exists between two modules; when data from one module is used to direct the order of instruction execution in another. For example, “Flag” set in one module and tested in another module.



Selection of subordinate module depends on decision points or iterative processing.

A variable (L) that controls decisions in subordinate module ‘N’ is set in superordinate module ‘L’ and then passed to N.

iv) External Coupling

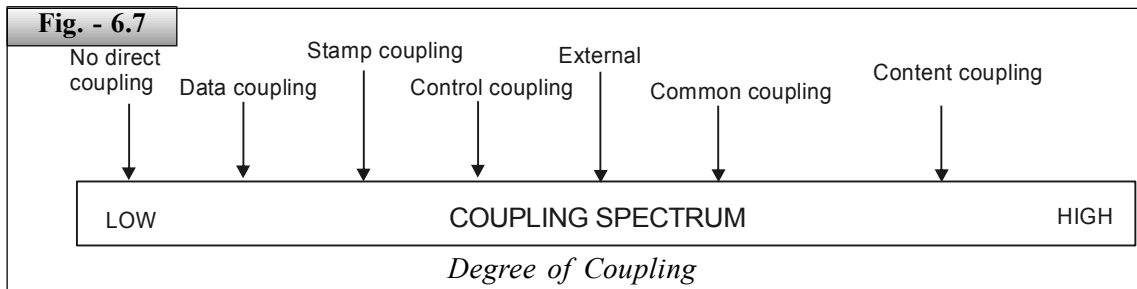
External coupling occurs when modules are fired to an environment external to the software. External coupling is essential but it should be limited to a smaller number of modules with structure.

v) Common Coupling

Common coupling exists in between two modules if they share some global data items e.g. Global variables. Diagnosing problems in structures with considerable common coupling is time-consuming and difficult. However, this does not mean that the use of global data is necessarily “bad”. It signifies that a software designed must be aware of potential consequence of common coupling and take special care to guard against them.

vi) Content Coupling

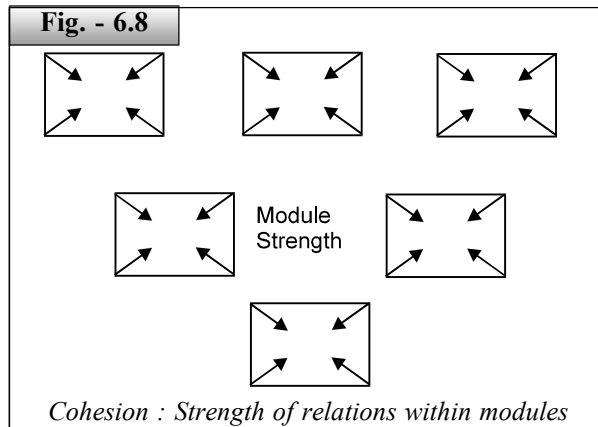
Content coupling is the highest degree of coupling. It occurs when one module makes use of control or data information maintained within the boundary of another module. Content coupling also occurs when branching takes place in to the middle of a module (and not at the beginning). Content coupling ought to be avoided.



NOTE : High coupling among the modules not only makes a design difficult to analyze and maintain, but it also increases development effort as the modules having high coupling cannot be developed independently by different team members. Modules having high coupling are difficult to implement and debug.

6.6.3 Cohesion

Cohesion is the property by which “like things” are placed or held together. Cohesion is the common property that holds the pieces in a structure like glue. It is an extension of the information-hiding attribute. A cohesive module would carry-out a single task in a software procedure. It would need little or no interaction with other procedures being performed in other parts of the program. In other words, a cohesive module would look at accomplishing just one specific task. We should always aim for a high degree of cohesion, though even a mid-level of cohesion would be readily acceptable. Cohesion follows a non-linear scale, meaning that low-end cohesion is worse than mid-range cohesion, which again is almost as good as high-end cohesion. Low-end cohesion is counter productive for design purposes and should be avoided as much as possible.



Cohesion is the measure of the relative functional strength of a module. This cohesion of a component is a measure of the closeness of the relationships between its components. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program.

The above figure represents the cohesiveness of the module of a software structure.

Thus it is concluded that, we need to maximize the interaction within a module. Therefore, an important design objective is to maximize the module cohesion and minimize the module coupling.

Types of Cohesion

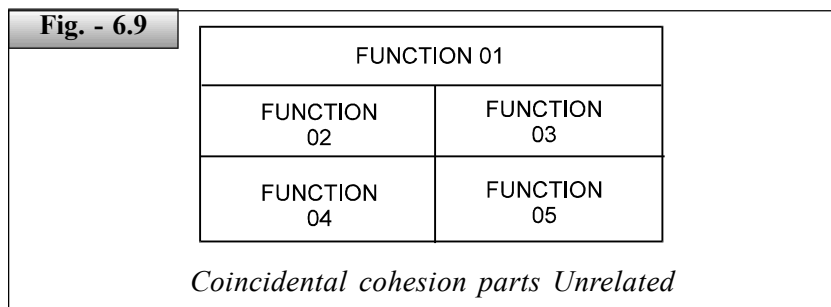
There are seven levels of cohesion in increasing order to desirability which are as follows :

Table : 6.2 - Types of Cohesion

Coincidental Cohesion	Worst (Low)
Logical Cohesion	
Temporal cohesion	
Procedural cohesion	
Communicational cohesion	
Sequential cohesion	
Functional cohesion	Best (High)

i) Coincidental Cohesion :

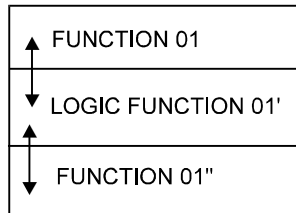
A module is said to be coincidentally cohesive when it carries out a set of tasks that are loosely related to each other. In this type of cohesion, the module contains a random collection of functions. It means that the functions have been put in the module out of pure coincidence without any thought or design. It is the worst type of cohesion.



ii) Logical Cohesion

A module is said to be logically cohesive when it performs tasks that are loosely related to each other by some logic. In this case all the elements of the module performs similar operations.

For example, error handling, data input, data output etc. An example of logical cohesion is the case where *a set of print functions generating different output reports are arranged in to a single module.*

Fig. - 6.10

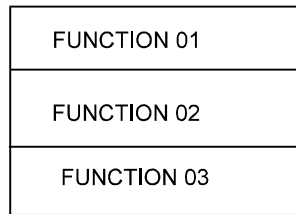
Logical Cohesion Similar Function

iii) Temporal Cohesion

A module is said to be temporally cohesive when it performs certain tasks that are related solely by the fact that they all have to be executed within a certain time span.

iv) Procedural Cohesion

procedural cohesion can be said to exist when the processing elements of a module are related and have to be executed in a certain preset order. High cohesion is achieved when a module performs a single procedural task (*algorithm*). For example the algorithm for decoding a message.

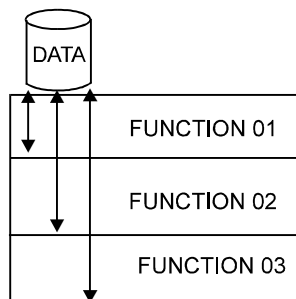
Fig. - 6.11

Procedural Cohesion Related by ordered function

v) Communicational Cohesion :

Communicational cohesion is said to exist when all the processing elements in a module concentrate on a single specific area of data structure.

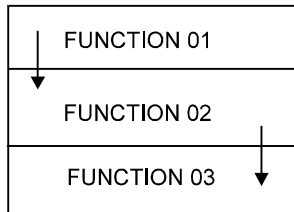
For example, the set of functions defined on an array or a stack. All the modules in communicational cohesion are bound tightly because they operate on same input or output data. For Example, set of functions defined on *an array or a stack*.

Fig. - 6.12

Communicational Cohesion : Access same Data

vi) Sequential Cohesion

Sequential cohesion exists when the processing elements in a module (*statements*) are related to one another in that they have to be executed sequentially i.e. a module is said to possess sequential cohesion, if elements of a module form the part of a sequence, where the output from one element of the sequence is input to the next.

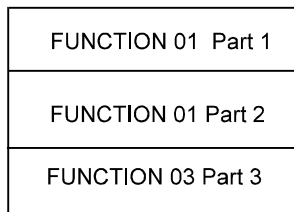
Fig. - 6.13

Sequential cohesion : out of one part is input to next

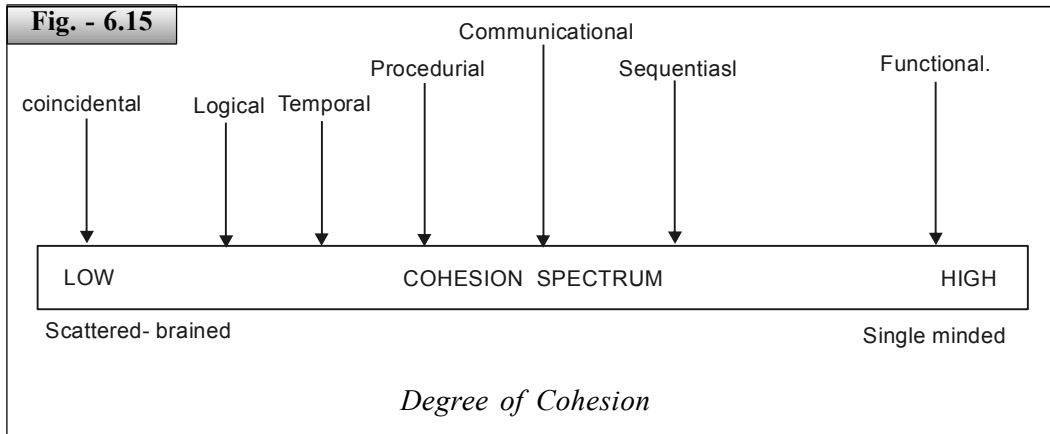
vii) Functional Cohesion

It is same as procedural cohesion. The processing elements in a module are related to one another in the need to perform as specific, well - defined function.

Functional cohesion is said to exist if different elements of a module cooperate to achieve a single function. For example, managing an employee's pay roll. When a module displays functional cohesion, and if we are asked to describe what the module does we can describe it using a single sentences.

Fig. - 6.14

Functional Cohesion : Sequential with complete, Related Functions

Fig. - 6.15

6.6.4 Span of Control

Span of control refers to the number of subordinate modules controlled by a *calling module*. In general, we would seek to have no more than five to seven subordinate module.

Fig. - 6.16

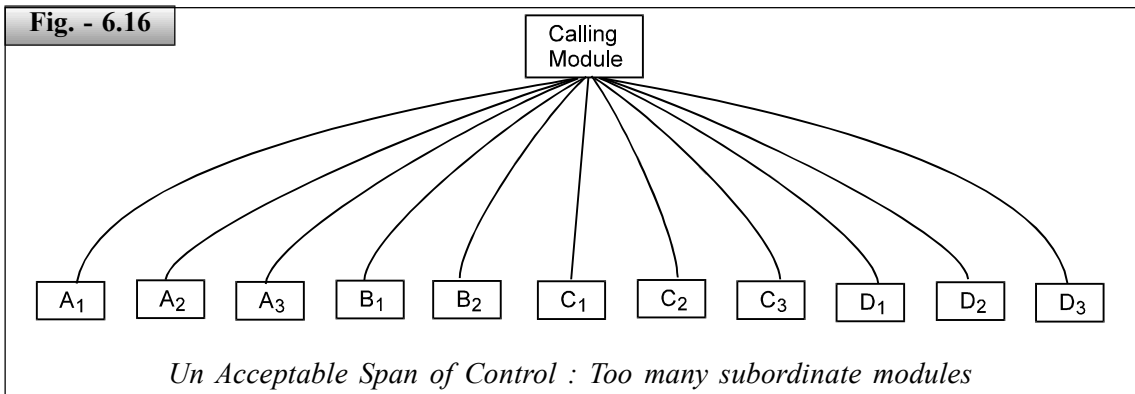
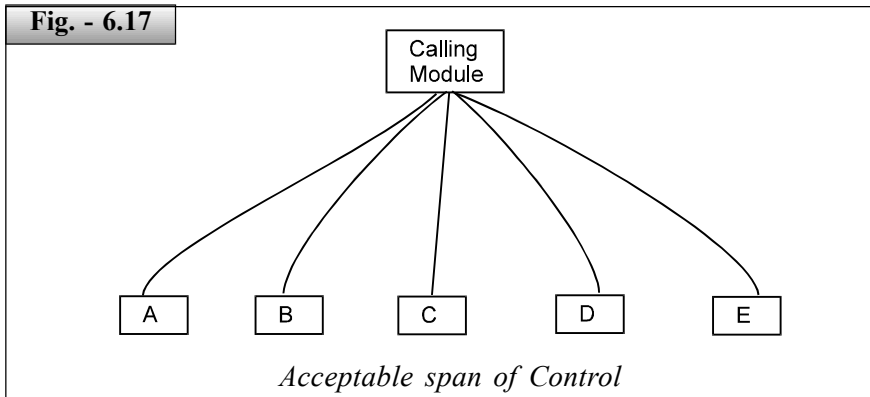


Fig. - 6.17



On the one hand, excessive span of control, meaning a high number of subordinate modules creates considerable overhead in determining which module will invoke certain conditions and in establishing calling sequences to pass data and receive results.

6.6.5 Module Size

It describes the number of instructions composing a module.

During the design phase the designers must look after the module size so as to limiting the size, so that full module function focuses a single purpose there by maximizing the cohesiveness of the module.

Therefore the software engineers consider about the followings :

- How large should a program module be ? While it is impossible to fix a specific number of instructions, there are useful guidelines to manage module size.

Many organisations have established rules to manage module size. A common one is that no module should contain more than 50 instructions. Another is that *one listing of source code for a module should fit on a single printed page*.

In some situations, these rules are appropriate, but in others they results in arbitrary decisions (For example “*If the module cannot be coded in 50 instructions, create a second*”).

module that is called by the first) that miss the point of managing module size.

In general, we should seek, designs in which the module focus on a single purpose, are highly cohesive, and are loosely coupled.

The size of the module depends on the language used for the software to be designed. Approximately 50 statements in COBOL may be an acceptable upper limit (*programmers should not have to redesign if, say 51 or 55 statements are needed, providing other design objectives are met*). On the other hand, 50 instructions is a powerful fourth generation language (4GL) (*where one instruction replaces from 10 to 50 equivalent COBOL statements*) will probably be unacceptable.

6.6.6 Shared Use

Share use results from the desire to have a certain function, calculation, or type of processing performed in one place in a system. We want to design the system so that a certain calculation (*such as determination of sales tax in an order processing system*) is performed once. Then the module can be shared throughout the system by invoking it from various calling modules.

Why share modules ? There are several reasons to answer this question.

First : Sharing modules minimizes the amount of software that must be designed and written.

Second : It minimizes the number of changes that must be made during the system maintenance.

For instance, if tax calculation procedures change, only one module must be modified under the shared module principle.

And third : having a single shared module reduces the chance of error : there is a greater likelihood that redundant modules will follow different calculation procedures, if not initially, after the introduction of maintenance changes (*one module may not be changed because it is overlooked*). Many systems establishes library modules - Predefined - Procedures - that are included in the system's program Library. The routine is quickly invoked by single command or call.

6.7 IEEE RECOMMENDED DDS [DESIGN DOCUMENT SPECIFICATION] OR SDD [SOFTWARE DESIGN DOCUMENT]

It is recommended by IEEE (*Institute of Electrical and Electronics Engineers*) for the descriptive structure of a DDS/SDD i.e. the major output of a software design process, on which the coding activity is to be carried out.

DDS/SDD shows how the software system (product) will be structured to satisfy the requirements as shown in a SRS.

It is basically the transformation of the requirements into a description of the structure, components, interfaces, data structure etc. of a software product.

Definitions

Few important definitions are given below:

- (i) **Design entity.** An element (Component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

- (ii) **Design View.** A subset of design entity attribute' information that is specifically suited to the needs of a software project activity.
- (iii) **Entity attribute.** A named property or characteristics of a design entity. It provides a statement of fact about the entity.
- (iv) **Software design description (SDD).** A representation of a software system created to facilitate analysis, planning, implementation and decision making. A blueprint or model of the software system. The SDD is used as the primary medium for communicating software design information.

6.7.1 Content description of SDD / DDS

- (i) **Introduction.** The SDD is a representation or model of the system to be created. The model should provide the precise design information needed for planning and implementation of the software system. It should represent partitioning of the system into design entities and describe the important properties and relationship among those entities.
- (ii) **Design entities.** A design entity is an element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

Design entities result from a decomposition of the software system requirements. The objective is to divide the system into separate components that can be considered, implemented, changed, other components.

The entities may have different nature, but may have common characteristics. Each design entity will have a name, purpose, and function. There are common relationship among entities such as interfaces or shared data. The common characteristics of entities are described by design entity attributes.

- (iii) **Design entity attributes.** A design entity attribute is a named characteristic or property of a design entity. It provides a statement of fact about the entity.

Design attributes can be thought of as questions about design entities. The answer to those questions are the values of the attributes. All the questions can be answered, but the content of the answer will depend upon the nature of the entity. The collection of answers provides a complete description of an entity.

All attributes shall be specified for each entity. Attribute descriptions should include references and design considerations such as trade-offs and assumptions when appropriate. In some cases, attribute descriptions may have the value none. When additional attributes are identified for:: specific software project, they should be included in the design description. The attributes are associated information items are defined in the following subsections (from 'a' to 'j').

- a) **Identification.** The name of the entity: Two entities shall not have the same name. The names for the entities may be selected to characterise their nature. This will simplify referencing & tracking in addition to providing identification.
- b) **Type.** A description of the kind of entity. The type attribute shall describe the nature of entity. It may simply name the kind of entity, such as subprogram, module, procedure, process, or data store. Alternatively, design entities may be

grouped into major classes to assist in locating an entity dealing with a particular type of information. For a given design description, the chosen entity types shall be applied consistently.

- (c) **Purpose.** A description of why the entity exists. The purpose attribute shall provide the rationale for the creation of the entity. Therefore, it shall designate the specific functional and performance requirements for which the entity is created.
- (d) **Function.** A statement of what the entity does. The function attribute shall state the transformation applied by the entity to inputs to produce the desired output. In the case of a data entity, this attribute shall state the type of information stored or transmitted by the entity.
- (e) **Subordinates.** The identification of all entities composing this entity. The subordinates attribute shall identify the “composed of relationship” for an entity. This information is used to trace requirements to design entities and to identify parent/child structural relationships through a software system decomposition.
- (f) **Dependencies.** A descriptions of the relationships of this entity with other entities. The dependencies attribute shall identify the uses or requires the presence of relationship for an entity. These relationships are often graphically depicted by structure charts, data flow diagrams etc.
- (g) **Interface.** A description of how other entities interact with this entity. The interface attribute shall describe the methods of interaction and the rules governing those interactions. The methods of interaction include the mechanisms for invoking or interrupting the entity, for communicating through parameters, common data areas or messages, and for direct access to internal data. The rules governing the interaction include the communications protocol, data format, acceptable values, and the meaning of each value.
This attribute shall provide a description of the input ranges, the meaning of inputs & outputs, the type and format of each input or output error codes. For information systems, it should include inputs, screen formats, and a complete description of the interactive language.
- (h) **Resources.** A description of the elements used by the entity that are external to the design. The resources attribute shall identify and describe all of the resources external to the design that are needed by this entity to perform its function. The interaction rules and methods for using the resource shall be specified by this attribute.
This attribute provides information about items such as physical devices (like printers), software services (like math libraries, OS services), and processing resources (like CPU cycles, memory allocation, buffers).
- (i) **Processing.** A description of the rules used by the entity to achieve its function. The processing attribute shall describe the algorithm Used by the entity to perform a specific task and shall include contingencies. This description is a refinement of the function attribute.

- (j) **Data.** A description of data elements internal to the entity. The data attribute shall describe the method of representation, initial values, use, semantics, format, and acceptable values of internal data.

The description of data may be in the form of a data dictionary that describes the content, structure, and use of all data elements. Data information shall describe everything pertaining to the use of data or internal data structures by this entity. It shall include data specifications such as formats, number of elements, and initial values. It shall also include the structures to be used for representing data such as file structures, arrays, stacks, queues, and memory partitions.

6.7.2 Organisation of SDD

Each design description writer may have a different view of what are considered the essential aspects of a software design. The organisation of SDD is given in Table 6.3. This is one of the possible ways to organise and format the SDD.

Table 6.3: Organisation of SDD

1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions and acronyms
2. References
3. Decomposition description
 - 3.1 Module decomposition
 - 3.1.1 Module 1 description
 - 3.1.2 Module 2 description
 - 3.2 Concurrent Process decomposition
 - 3.2.1 Process 1 description
 - 3.2.2 Process 2 description
 - 3.3 Data decomposition
 - 3.3.1 Data entity 1 description
 - 3.3.2 Data entity 2 description
4. Dependency description
 - 4.1 Inter-module dependencies
 - 4.2 Inter-process dependencies
 - 4.3 Data dependencies
5. Interface description
 - 5.1 Module Interface
 - 5.1.1 Module 1 description
 - 5.1.2 Module 2 description
 - 5.2 Process interface
 - 5.2.1 Process 1 description
 - 5.2.2 Process 2 description
6. Detailed design
 - 6.1 Module detailed design

- 6.1.1 Module 1 detail
- 6.1.2 Module 2 detail
- 6.2 Data detailed design
 - 6.2.1 Data entry 1 detail
 - 6.2.2 Data entry 2 detail

Entity attribute information can be organised in several ways to reveal all of the essential aspects of a design. There may be number of ways to view the design. Hence, each design-view represents a separate concern about a software system. Together, these views provide a comprehensive description-of the design in a concise and usable form that simplifies information access-and assimilation. A recommended organisation of the SDD into separate design views to facilitate information access and assimilation is given in Table 6.4.

Table 6.4: *Design views*

<i>Design View</i>	<i>Scope</i>	<i>Entity attribute</i>	<i>Example representation</i>
Decomposition description	Partition of the system into design entities.	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, re-sources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

6.8 USER INTERFACE DESIGN

Interface design is one of the characteristics of software design process, that focuses on areas like :

- interface between software components.
- interface between software and the external entities.
- interface between the user and the computer system using software products.

[Interface : *A bridge between the world of products or systems and the world of users.*]

Here we concentrate on user interface design :

User interface design creates an effective communication medium between an user and the computer system. A software engineer designs the user interface by applying an interactive process that entertains widely accepted design principle.

User interface design begins with the identification of user task and environmental requirements. Once the user tasks have been identified, user scenarios are created and analyzed to define a set of interface object and actions. These are the basis for the creation of screen layouts that represents graphical design and placement of icons, definition of screen text, specification and titles for windows, and specification of major and minor menu items. Some tools are used to prototype and implement the design model; and finally the result is evaluated for quality.

User interfaces in the field of communication (interaction) increases the human processing speed, reduce the human errors and speed up the computer processing time.

6.8.1 Graphical User Interface (GUI) vs. Character- based User Interface (CUI)

There are two major factors that distinguish GUI from CUI (*also called as text-based user interface*).

- the use of graphics to communicate information to the users visually in addition to textually.
- presenting finite number of options, to the users rather than requiring the users to memorize and manually enter commands from a virtually unlimited set of options.
- accessing / interacting different information using multiple windows displayed on the screen.
- command selection for different operation through attractive and user-friendly menu selection system [*Menu driver*]
- use of pointing devices like, light pen, mouse to increase the efficacy of the command issue procedure.
- representation of commands, directories, files, through icons, fielders etc. that makes the user only to select the icons and other related options for certain manipulation without any difficulties.
- facilitates any novice user to work on computer in a friendly manner.

Golden Rules for User Interface design

The Mandel [MAN97] coins three golden rules, that guide the important software design action for user interface design.

a) Place the user in control

The software product which is generally meant for the users, must be designed in such a way that user gets confidence to have control over the computer system and system activities.

These are some design principles to maintain control by the user.

- **Define interaction mode** : in a way that the user is free to move on from one application to other.
- **Provide flexible interaction** : Interaction must be flexible to user to handle complex task / operation.
- **User interaction to be interruptible and undoable** : when an user wants to quit an application or learning the current task, task can be interrupted or use 'undo' operation to seize an application / operation whenever required.

- ☛ **Customise the interaction on skill level :** Advanced user should interact with the system as per their skill in a friendly manner.
- ☛ **Hide technical internals from the casual user :** Normal user should not interact with the insight of any application or system, rather the normal operation at the user level.

Example : should not aware about operating system, file management functions and other technology used in the system etc.

- ☛ **Direct interaction with the objects on the screen :** The user should feel a sense of direct control to manipulate the objects. e.g. an application say Coreldraw, where user can stretch or squeeze an object (re-sizing) as per his / her need.

b) Reduce the User's Memory Load :

User should be free from remembering commands, any other operations those are difficult to memorize during interacting with the computer system. Some design principles (given below) that enable to reduce the user's memory load.

- ☛ **Reduce demand on short-term memory :** When users are involved in complex task, it is required to memorize / recall the past action and results. The interface should be design to reduce the requirements to remember. This can be accomplished by providing visual pool (where the recent past operations are available) that enables the user to recognize the past action, rather than recall them.
- ☛ **Establish meaningful pre-selected options :** It helps the user to redefine the original values as per the user's individual preference. e.g. using "reset" option the original default values will be redefined.
- ☛ **Define shortcuts that are immediate when mnemonics are used to :** Accomplish a system function (e.g. **alt + P** to invoke print function), the mnemonic should tied up with the option in such a way that it is easy to remember.
e.g : 'P' for print.
- ☛ **Visual layout of interface should be based on a real world metaphor (phrase) :** This enables the user to rely on well understood visual pool, rather than memorizing the interaction sequence.
- ☛ **Disclose information in a progressive fashion :** The interface should be organised hierarchically, i.e. the information about a task along with it's details should be presented sequence.
e.g. : *Pop-up menus, sub-menus etc.*

c) Make the interface consistent.

The interface should present and acquire information in a consistent fashion. i.e.

- all visual information is organised according to a design standard and is maintained throughout all screen displays.
- input mechanisms are constrained to a limited set that is consistently throughout the application.
- mechanism for navigating from task to task are consistently defined and implemented.

Some design principles that help to make the interface consistent.

- ☛ ***Allow the user to put the current task into a meaningful context :*** Many interfaces implement complex layers of interactions with dozen of screen images. It is important to provide indicators (e.g. *window titles, graphical icons, consistent color codes*) so that the user can know the context of the work at hand. And also the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.
- ☛ ***Maintain consistency across a family of applications :*** A set of applications (products) should implement the same design rules so that the consistency is maintained for all interaction.
- ☛ ***If past interactive models have created user expectations, do not change the model :*** i.e. if some previous model in which the user is familiar, he/she always expect the same standards for other similar models.

6.8.2 Classification of User Interface

Generally User Interface can be classified into three categories.

a) Command Language-Based Interface

This interface is based on the design of command language, where user interacts with the computer systems through predefined commands w.r.t. the language environment.

Advantages :

- ☛ Command language-based interface facilitates the user to type commands using key board which is cheaper as compare to other graphical input devices.
- ☛ A set of related commands can also be combined to form *Macro command*, which performs some typical operations and reduces the number of commands to be typed.
- ☛ Command language - based interface is much easier to develop as compare to graphical interface.

Disadvantages :

- ☛ User has to remember the commands, related to their operations.
- ☛ There is possibility of errors or type mismatch during typing the commands even, the user has to be much aware of the command syntax (*How to use it ?*)
- ☛ All user interactions will be made through the keyboard only, although more sophisticated devices (*mouse, light pen etc.*) are available.
- ☛ Novice users find more difficulties to interact with the computer/ application through command language based interface.

b) Menu-Based Interface :

This interface has great advantages over the command language based interface. The users need not have to remember the syntax of using commands, rather the command names only. This interface facilitates the user for nominal use of keyboards, and more efficiently the graphical supporting devices (*pointing device*) are used just to **click** on any *command menu* to get the work done. Some of the limitations of this interface are : *designing needs more care to carry out.*

- It is more complex to structure a number of menu based commands or choices. However, the above difficulties can be solved using.
 - *Scrolling menus*
 - *Pop-up menus*
 - *Pull-down menus etc.*

c) **Direct manipulation Interfaces**

This interface is in the form of visual models i.e icons or objects, where the user issues commands by performing actions on the visual representation of icons.

Direct manipulation interface facilitates the user to recognize the icons, what are language independent.

It is also difficult to give complex commands using this interface.

6.8.3 Qualities of good User Interface Design (UID)

Some standard (mostly accepted) qualities of a good interface design are :

- **Simplicity** : It should be simple, to minimize the number of *mouse clicks* or *keystrokes* to perform a tasks and also easy to understand.
- **Consistency** : The interface should have a certain consistency, else it will be more difficult to use. The interface should provide permanent objects as unchanging reference points around which the user can navigate. If the users even get lost or disoriented, they should be able to quickly find the permanent objects and from there get to go where they need to be. It also prevents on-line designers information chaos, ambiguity and inability.
- **Intuitiveness** : It is the most fundamental quality, what enables the users for easy learning and use it in an efficient way. Icons and labels should be brief and clear, and must to the standards of the terminology that the application supports.
- **Prevention** : A good user interface should prevent users performing an illegal operations, and this is accomplished by disabling certain elements under specific conditions.
- **Forgiveness** : It encourages the users to use the software to a full extent, by providing users with a way out when they find themselves somewhere they should not be. It is much better to label the buttons more cleanly indicating what action will be taken.
- **Graphic Interface Design** :
A graphic interface Design provides screen displays that create an operating environment for the user and form an explicit visual and functional context for the user's actions. It consists of standard objects such as *buttons icons, text fields, windows, and pull-down or pop-up screen menus*.
- **Aesthetics** : it is crucial to present an attractive user interface. The effectiveness of an user interface is influenced by aesthetics, so it should be presented with a pleasing user interface.

6.8.4 User Interface Design Principle

User-Interface design is governed by a set of design principles, some of the highlighted principles are :

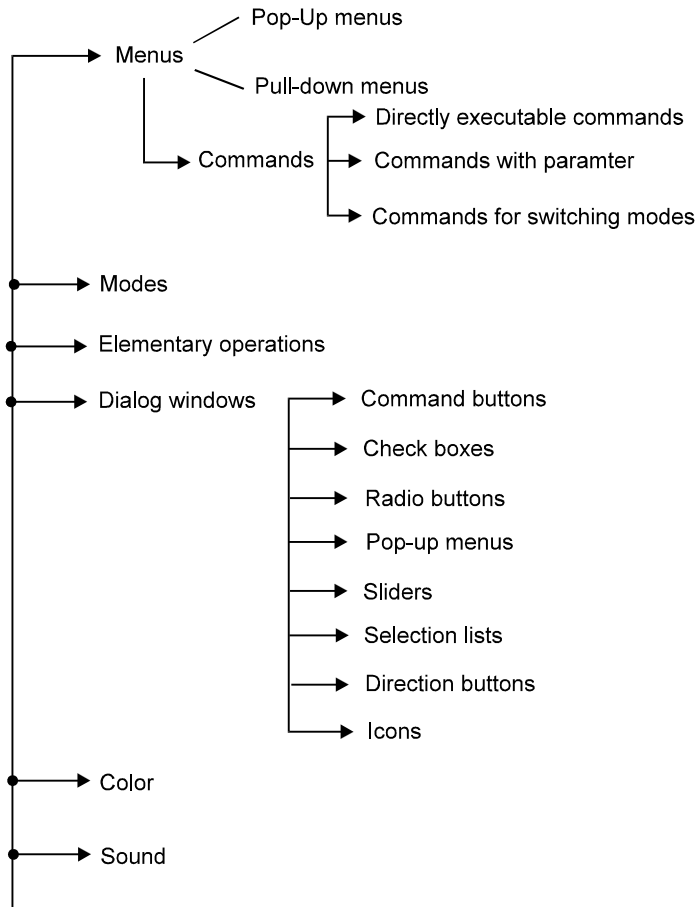
- ☛ **User Profiling** : To know in detail about the user (*technical / non-technical*)
- ☛ **Methaphor** : To know the behaviour of the systems (*previously used by the user*)
- ☛ **Feature Exposure** : To make the user to know about the detail functions available in the new system.
- ☛ **Coherence** : The behaviour of the program should be consistent both internally and externally i.e. for the parts of the program and with environment like operating system, applications etc.
- ☛ **State visualization** : Every change in the behaviour of the system (program) should be accompanied by a corresponding change in the appearance of the interface.
- ☛ **Shortcuts** : to provide both concrete and abstract way of getting a task done i.e. pre-memorized “*shortcuts*” should be available to allow rapid access to more powerful functions.
- ☛ **Focus** : Some aspects of user Interface attract attention more than others do.
e.g : *mouse, text cursor* etc.
- ☛ **Help** : To understand the different kinds of help a user needs.
- ☛ **Safety** : To make the user, developing confidence by providing safety.
- ☛ **Context** : Limiting user activities to a specific context and create a good relationship between contexts.
- ☛ **Aesthetic** : The program should be beautiful.
- ☛ **User testing** : Must have the provision for instant help in identifying defects in the design.
and more .

6.8.5 Elements for user interface Design

These are the elements generally required for designing user - Interface.

- Menus
- Modes
- Elementary operations
- Dialog
- Color
- Sound

All the elements are shown graphically as follows :



User Interface design may also be called as user interface engineering, which is the design of computer systems, appliances, machines, mobile communication devices, software applications, websites. As it is mostly associated with the user needs, it is often called user-centered design.

Interface design is involved in a wide range of project from computer-to-commercial applications. As a result, designers tend to specialize in certain type projects and have skills centered around their expertise.

6.8.6 Graphical user interface

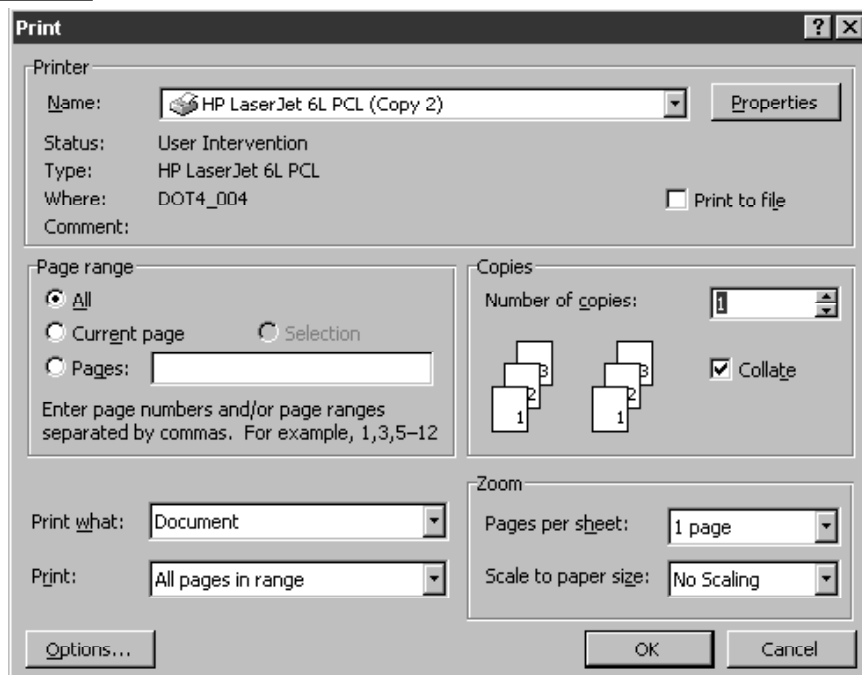
GUI (Pronounced as goo-ey) stands for *graphical user interface*. It is a popular user interface. One of the first computers to support the GUI-based applications was *Apple Macintosh computer*. There after number of other electronic devices like, *MP₃ players, portable media players, gaming devices, household appliances, office equipments etc. supported these (GUI) applications.*

In the earlier days, the programmers were writing programs to draw lines, circles, boxes etc. to display menu items under graphical user interface. But the trend, technologies have changed to a greater extent and the programmers are readily availed with a set of readymade (*pre-defined*) components like *menu*, *dialog boxes*, *forms* etc. to create good GUI based applications. This technique may also be called as *component based GUI developments*.

GUIs consists of **widgets** (*window objects*) that allow a user to interact with computer systems as well as software applications.

Some of the widgets are : *Icons*, *Menus*, *Buttons*, *selection boxes*, *scroll bars* and *other design elements*.

Fig. - 6.18



Dialog box is a typical GUI element

6.8.6.1 Elements of GUI design

A set of standard elements for this design are :

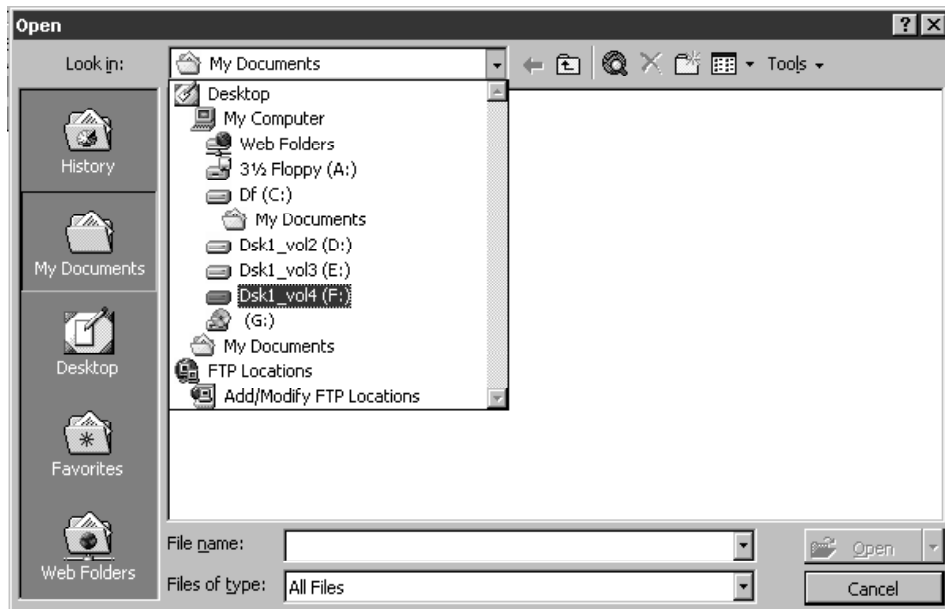
- Windows
- Pull down / Menus
- Icons
- Radio Buttons
- Sliders
- Popup list
- Events
- Push Buttons
- Check boxes.
- Scrolling lists
- Text field
- Spin Boxes.

Window : It is the most comprehensive element used in GUI.

A window can be thought as a rectangular area of the computer screen, even we can consider the video screen to be a window into the computer system. In character-based

interface, the video screen was considered as the only window. Where as in case of GUI, multiple windows can be created within the video screen area

Fig. - 6.19



Client part and non client parts.

A window can be divided into two parts : *client part* and *non client parts*.

Client part is the area. i.e. the white window except the borders and scroll bars, which is used to display the client applications.

Where as, the *non-client part* of the window decides the basic behaviour (*create, move, resize* etc.) of all the windows displayed on the screen.

A window manager (*Utility Program*) is used to manage and maintain the non-client part of the window.

Events

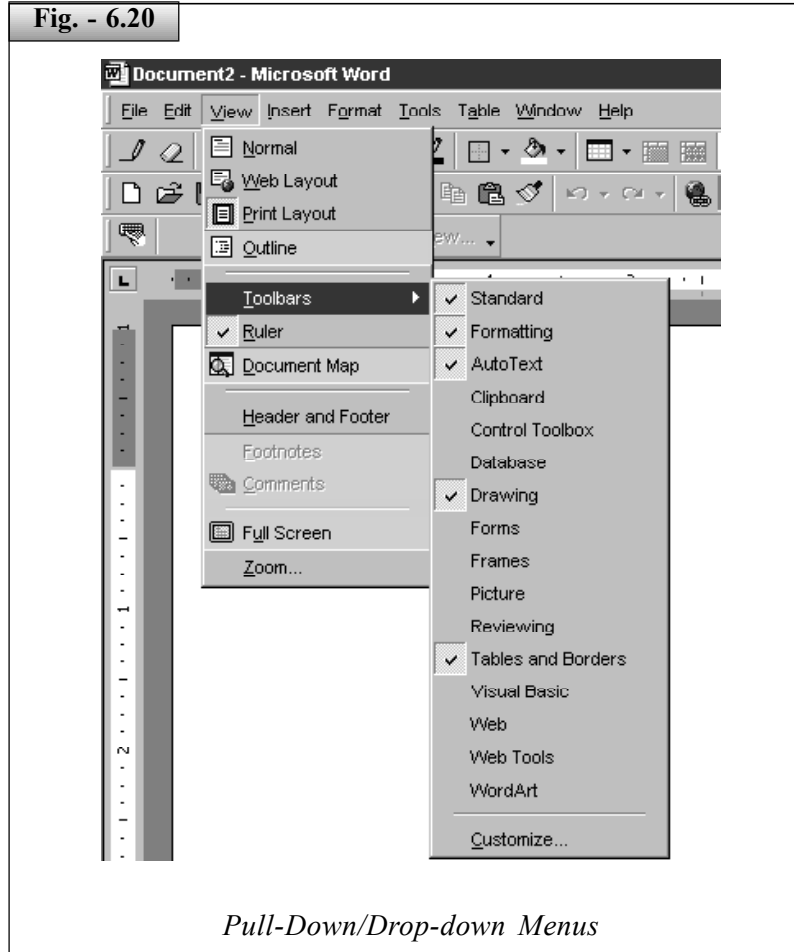
An event with respect to user interfaces is any function initiated by the user. Selecting something from a *pull-down menu*, clicking a *button* or a *checkbox*, and closing a window are all examples of events.

It is a common misconception that events were born of the *GUI* age. In fact character-based user interfaces (*GUIs*) accommodated myriad events, but they were generally performed by complex keystrokes such as **Control-Shift-F6**.

By visually presenting all of the possible events to the user, the need for a quick reference card vanished. Also, the advent of pointing devices such as the mouse allowed entirely new kinds of events to be implemented, such as **drag and drop**.

Pull-down Menus/Drop-down Menu

Pull-down menus are menus that the user can *pull down* from the menu bar that traverses the top of the screen. On some platforms these are called *drop down* menus because the user does not need to hold the *mouse button* down in order for the menu to remain visible. These are illustrated in **Fig. 6.20**.

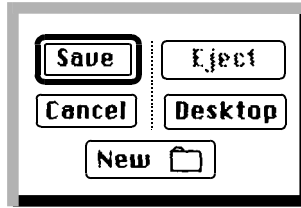


Push Buttons

A *push button* is simply a rectangle that appears on a panel with some sort of label or icon inside it. The metaphor is to any button you'd find in the physical world, such as on a calculator or telephone. *Clicking on* a *push button* will cause some sort of action will occur.

Sometimes a panel will have a *default* button, which appears with an enhanced border. This button will be activated when the Enter key is pressed. Push buttons are illustrated in **Fig. 6.21**.

Fig. - 6.21

*Push Buttons*

Icons

Icons are small pictures that are generally represent objects in the physical world or are used as metaphors for functions or actions. Icons can be *clickable* and used to initiate an event These are sometimes referred to as *buttons* as they become hybrids of buttons and icons These are illustrated in Fig. 6.22.

Fig. - 6.22

*Buttons and Icons*

Checkboxes

A checkbox is a small square with some sort of label inside it. Clicking on a checkbox will cause an 'X' to appear in the box. Clicking it again will cause the 'X' to disappear. These are illustrated in Fig. 6.23.

Fig. - 6.23

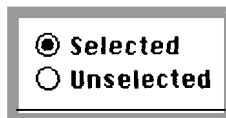
*Checkboxes*

Radio Buttons

A radio button is small circle with some sort of label beside it. A black dot inside the circle indicates that the button is selected. The absence of a dot indicates that the button is unselected. Only one button can be pushed in at a time. Similarly, with the radio button form object, clicking on one option will cause whatever other option that had been selected to become unselected.

- When applicable, radio buttons should be placed in a logical order.
 - Don't use radio buttons for binary choices (e.g. Yes/No). Use check boxes for this.
- These are illustrated in Fig. 6.24.

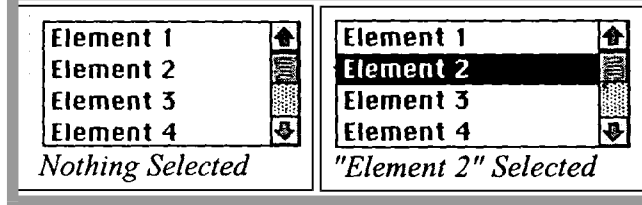
Fig. - 6.24

*Radio-Buttons*

Scrolling Lists

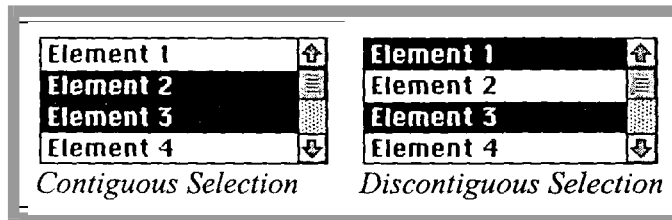
Scrolling lists are lists of elements that appear in a box with a scroll bar on the side, allowing the user to scroll through the elements in the list. In this way the list can contain more elements than can be displayed at any one time. Clicking on an element in the list will cause that element to become selected, as shown below in **Fig. 6.25**.

Fig. - 6.25

*Scrolling Lists*

Optionally, a list can allow multiple selections. These can either be contiguous or discontinuous, as shown below respectively in **Fig. 6.26**.

Fig. - 6.26

*Scrolling Lists with Multiple Selections*

Text Field

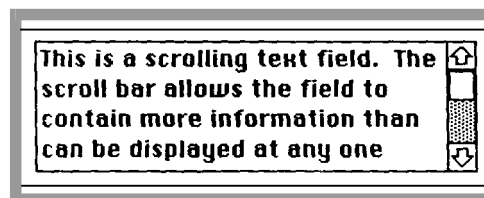
A text field is simply a space in which the user can type text. Text fields are usually contained within a rectangle, but it could just be space on a panel that can accept text.

Fig. - 6.27

*Text Field*

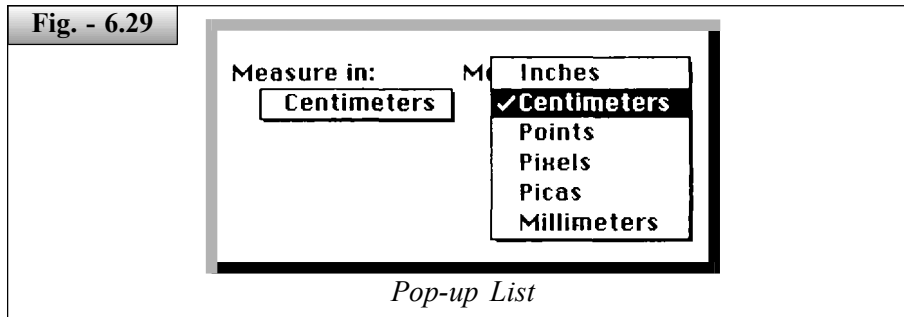
If a text field is contained within a rectangle, it can optionally have a scroll bar to allow it to contain more information than can be displayed at any one time.

Fig. - 6.28

*Text Box*

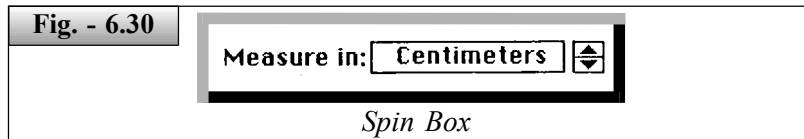
Pop-up List

A popup list appears initially as a box containing some sort of label. It is shown in Fig. 6.29. When the user clicks on the box, a larger box containing a variety of choices will **pop up**. By moving the mouse, the user can cause another choice to become selected. When the user releases the mouse, the popup menu will disappear leaving currently selected choice to appear in the box.



Spin Boxes

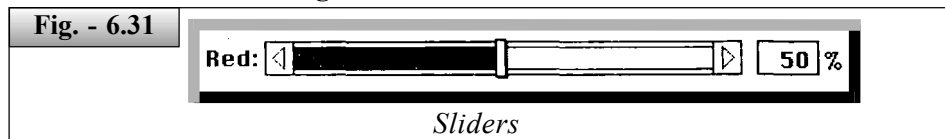
A spin box is a box containing some value, with up and down arrows on one side and generally a label on the other. It is shown in Fig. 6.30.



By clicking the up or down arrow, different values will appear in the box. The entire range of choices can be cycled through if the up or down arrows are clicked enough times. Whatever value appears in the box is the value that is selected at that time.

Sliders

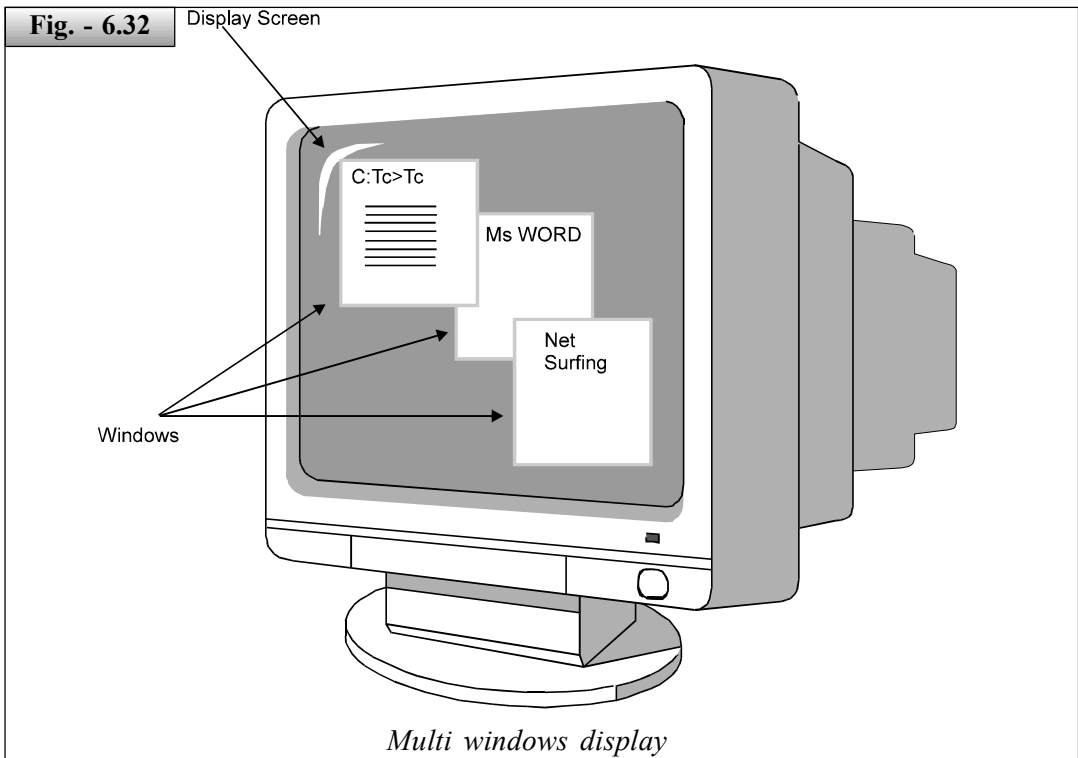
A slider is a long box with a control that the user is able to slide one way or the other. It is illustrated in Fig. 6.31.



Often a slider will have an arrow button at either end to allow the user to adjust the slider one unit at a time. Also there is often a text field to the side of the slider that will indicate the value that the slider is currently set on and allow the user to enter a specific value.

6.8.6.2 Window Management System (WMS)

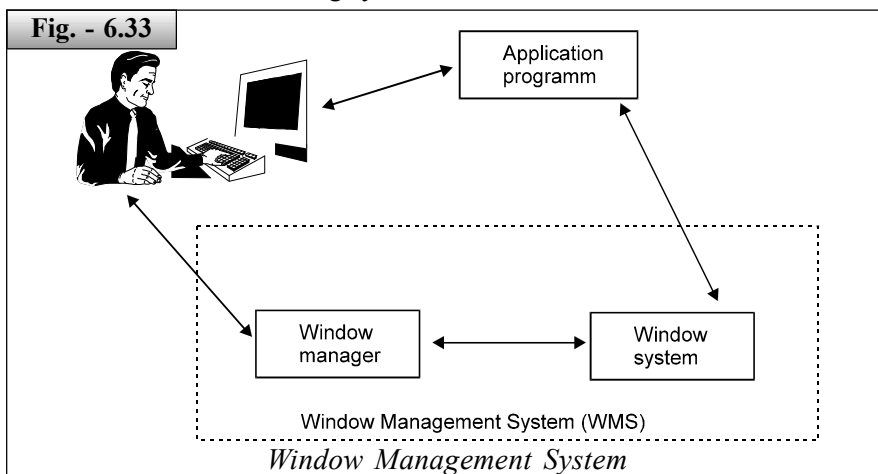
In a GUI environment, there are multiple windows displayed on the video screen. So it is necessary to have control over these windows and also need to be managed efficiently.



Window Management System (*WMS*), is a resource manager what controls, manages and maintains this multi windows environment. *WMS* can also be considered as a user interface management system. It assists the application programmer during the user interface development process by managing the resources, imparting the basic behaviour of the windows (more resize and also providing utility routines).

Window Management System (*WMS*) consists of two major parts.

- the window manager
- the windowing system.



Window Manager is associated with the management of window operations like *window positioning, resizing, iconifications etc.* It may also be considered as a special kind of client that makes use of the services (*function calls*) supported by the window system. Even the application programmer can also directly invoke the services of the window system to develop user interface.

Window System is an environment what enables the application programmer for user interface design by providing a set of routines and widgets (window objects) like icons, menus, buttons etc. Some of the windowing systems are :

- DEC Windows (for VMS)
- X-Window system (for GNU and Unix like systems)
- Microsoft Windows
- Sun's open windows.

6.8.6.2.1 X-Window system

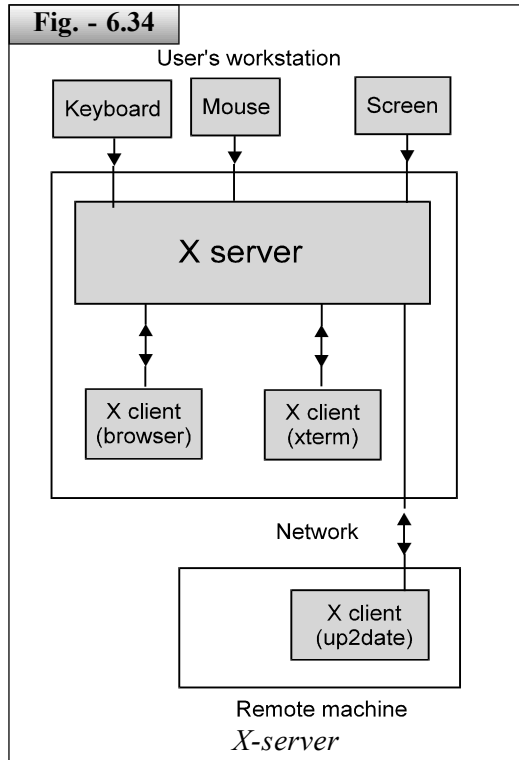
X-window system is a network transparent windowing system based on *client-server* model. The applications developed using X-window system are device independent, and also network independent i.e. the interface can work as a terminal connected at any place in the same network to which the compute running the application is connected.

X-window system was created as a part of *Project Athena at MIT (Massachusetts Institute of Technology)*

a) Client-server model (X-Architecture)

An *X-server* program runs on a computer with a graphical display and communicates with various *client programs*. The server acts as a go-between for the user and the client programs, accepting requests for graphical output (*windows*) from the client programs and showing them to the users and receiving user input (*keyboard, mouse*) and transmitting it to the client programs.

In *X-window systems*, the server runs on the user's computer, while the client may run on a different machine. This is the reverse of the common configuration of client-server systems, where the *client* runs on the user's computer and the server runs on a remote computer. This reversal often confuses new X users. The X Window terminology takes the perspective of the program, rather than the end-user or the hardware : the remote programs connect to



the X server display running on the local machine and thus act as clients; the local X display accepts incoming traffic, and thus acts as a server.

X-server runs on the hardware to which display and keyboard are attached. X server performs low level graphics, manages windows and user input function as well controls access to a bit-mapped graphics display resource and manages it.

In this example, the X server takes input from a keyboard and mouse and displays to a screen. A *web broker* and a *terminal emulator* run on the user's workstation, and a system updater runs on a remote server but is controlled from the user's machine. Note that the remote application runs just as it would locally.

The *communication protocol* between server and client runs *network-transparently*: the client and server may run on the same machine or on different ones, possibly with different *architectures* and *operating systems*. A client and server can communicate *securely* over the *Internet* by *tunneling* the connection over an encrypted connection.

b) Core protocol

Communication between server and clients is done by exchanging packets over a network channel. The connection is established by the client, which sends the first packet. The server answers by sending back a packet stating the acceptance or refusal of the connection, or with a request for a further authentication. If the connection is accepted, the acceptance packet contains data for the client to use in the subsequent interaction with the server.

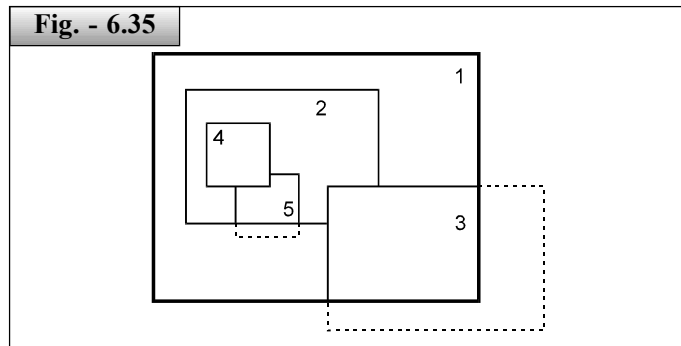
After connection is established, four types of packets are exchanged by the client and the server over the channel:

1. **Request:** The client requests information from the server or requests it to perform an action.
2. **Reply:** The server responds to a request. Not all requests generate replies.
3. **Event:** The server sends an event to the client, e.g., keyboard or mouse input, or a window being moved, resized or exposed.
4. **Error:** The server sends an error packet if a request is invalid. Since requests are queued, error packets generated by a request may not be sent immediately.

The X server provides a set of basic services. The client programs realise more complex functionalities by interacting with the server. X-protocol is designed to be independent of hardware, operating system, programming language as well as underlying network protocol.

c) Windows :

Fig. - 6.35



A possible placement of some windows: 1 is the root window, which covers the whole screen; 2 and 3 are top-level windows; 4 and 5 are subwindows of 2. The parts of window that are outside its parent are not visible.

What is usually called a window in other *graphical user interfaces* is a *top-level window* in the *X-Window System*. The term *window* is also used for windows that lie within another window, that is, the *subwindows* of a *parent window*. Graphical elements such as buttons, menus, icons, etc. are all realized using windows.

A window can only be created as a subwindow of a parent window. This causes the windows to be arranged *hierarchically* in a *tree*. The root of the tree is called the root window, which is automatically created by the server. The top-level windows are exactly the direct subwindows of the root window. Visibly, the root window is as large as the screen, and lies behind all other windows.

d) Identifiers

All data about windows, fonts, etc. is stored in the server. The client knows identifiers of these objects—integers it can use as names for them when interacting with the server. For example, if a client wishes a window to be created, it requests the server to create a window with a given identifier. The server creates a window and associates it with the identifier. The identifier can be later used by the client to request, for example, a string to be drawn in the window.

Identifiers are unique to the server, not only to the client; for example, no two windows have the same identifier, even if created by two different clients. A client can access any object given its identifier, even if the object has been created by another client.

e) Attributes and properties

Every window has a predefined set of attributes and a set of properties, all stored in the server and accessible to the clients via appropriate requests. Attributes are data about the window, such as its size, position, background color, etc. Properties are pieces of data that are attached to a window. Contrary to attributes, properties have no meaning at the level of the X Window core protocol. A client can store arbitrary data in a property of a window.

A property is characterized by a name, a type, and a value. Properties are similar to *variables in imperative programming languages*, in that the application can create a new property with a given name and of a given type and store a value in it. Properties are associated to windows: two properties with the same name can exist on two different windows while having different types and values.

Properties are mostly used for inter-client communication. For example, the property named `WM_NAME` is used for storing the name for the window; window managers typically read this property and display the name of the window at the top of it.

The properties of a window can be shown using the `xprop` program. In particular, `xprop -root` shows its properties of the root window, which include the *X-resources* (parameters of programs).

f) Events

Events are packets sent by the server to the client to communicate that something the client may be interested in has happened. A client can request the server to send an event to another client: this is used for communication between clients. For example, when a client requests the text that is currently selected, an event is sent to the client that is currently handling the window that holds the selection.

The content of a window may be destroyed in some conditions (for example, if the window is covered). Whenever an area of destroyed content is made visible, the server generates an **Expose** event to notify the client that a part of the window has to be drawn.

Other events are used to notify clients of keyboard or mouse input, of the creation of new windows, etc.

Some kinds of events are always sent to client, but most kinds of event are sent only if the client previously stated an interest in them. This is because clients may only be interested in some kind of events. For example, a client may be interested in keyboard-related event but not in mouse-related events.

g) Color modes

The way colors are handled in the X Window System sometimes confuses users, and historically several different modes have been supported. Most modern applications use *TrueColor* (24-bit color: 8 bits for each of red, green and blue), but old or specialist applications may require a different color mode. Many commercial specialist applications use *PseudoColor*.

The X11 protocol actually uses a single 32-bit unsigned integer for representing a single color in most graphic operations, called a *pixelvalue*. When transferring *primary colors* intensity, a 16 bits integer is used for each colour component. The following representations of colors exist; not all of them may be supported on a specific device.

- **DirectColor:** A pixel value is decomposed into separate red, green, and blue subfields. Each sub linked indexes a separate colormap. Entries in all colormaps can be changed.
 - **TrueColor:** Same as DirectColor, except that the colormap entries are predefined by the hardware and cannot be changed. Typically, each of the red, green, and blue colormaps provides a (near) linear ramp of intensity.
- **GrayScale:** A pixel value indexes a single colormap that contains monochrome intensities. Colormap entries can be changed.
 - **StaticGray:** Same as GrayScale, except that the colormap entries are predefined by the hardware and cannot be changed.
- **PseudoColor (Chunky):** A pixel value indexes a single colormap that contains color intensities. Colormap entries can be changed.
 - **StaticColor:** Same as PseudoColor, except that the colormap entries are predefined by the hardware and cannot be changed.

h) Xlib and other client libraries

Most client programs communicate with the server via the *Xlib* client library. In

particular, most clients use libraries such as *Xaw*, *Motif*, *GTK+*, or *Qt* which in turn use *Xlib* for interacting with the server. X-lib provides more than 300 utility routines for application to call. These routines convert procedure calls into requests that are transmitted to the server.

i) **Inter-client communication**

The X Window core protocol provides mechanisms for communication between clients: window properties and events, in particular the client-to-client message events. However, it does not specify any protocol for such interactions. These protocols are instead governed by a separate set of inter-client communication conventions.

The *Inter-Client Communication Conventions Manual* specifies the protocol for the exchange of data via selections and the interaction of applications with the window manager. This specification has been considered difficult and confusing; consistency of application *look and feel* and communication is typically addressed by programming to a given desktop environment.

The Inter-Client Exchange protocol (ICE) specifies a framework for building protocols for interaction between clients, so that a specific protocol can be built at the top of it. In particular, the X Session Management protocol (XSMP) is a protocol based on ICE that mandates over the interaction between applications with the *session manager*, which is the program that takes care of storing the status of the desktop at the end of an interactive session and recovering it when another session with the same user is started again.

Newer conventions are included in the *freedesktop* specifications, including the drag-and-drop convention Xdnd used for transferring data by selecting it and dragging in another window and the embedded application convention Xembedded which details how an application can be run in a subwindow of another application.

j) **Selections, cut buffers, and drag-and-drop**

Selections, cut buffers, and drag-and-drop are the mechanisms used in the X Window System to allow a user to transfer data from one window to another. Selections and cut buffer are used (typically) when a user selects text or some other data in a window and pastes in a different window. Drag-and-drop is used when a user selects something in a window, then clicks on the selection and drags it into another window.

Since the two windows may be handled by two different applications, data transfer requires two different clients connected with the same X server to interact. The X Window core protocol includes some types of requests and events that are specific to selection exchange, but the transfer is mainly done using the general client-to-client event sending and window properties, which are not specific to selection transfer.

Data to be transferred between clients can be of different types: it is usually text, but can also be a pixmap, a number, a list of objects, etc.

Selections and drag-and-drop are active mechanisms: after some text has been selected in a window, the client handling the window must actively support a protocol for transferring the data to the application requesting it. Cut buffers, by contrast, are a passive

mechanism: when the user selects some text, its content is transferred to a cut buffer, where it remains even if the application handling the window terminates and the window is destroyed.

k) Window manager

A window manager is a program that controls the general appearance of windows and other graphic elements of the *graphical user interface*. Differences in the look of X Window System in different installations are mainly due to the use of different window managers or different configurations of the window manager.

The window manager takes care of deciding the position of windows, placing the decorative border around them, handling icons, handling mouse clicks outside windows (on the “background”), handling certain keystrokes, etc.

From the point of view of the X server, the window manager is not different from the other clients. The initial position and the decorative borders around windows are handled by the window manager using the following requests:

1. an application can request the server not to satisfy requests of mapping (showing) subwindows of a given window, and to be sent an event instead;
2. an application can request changing the parent of a window.

The window manager uses the first request to intercept any request for mapping top-level unicons (children of the root window). Whenever another application requests the-mapping of a top-level window, the server does not do it but sends an event to the window manager instead. Most window managers *reparent* window: they create a larger top-level window (called the frame window) and reparent the original window as a child of it. Graphically, this corresponds to placing the original window inside the frame window. The space of the frame window that is not taken by the original window is used for the decorative frame around the window (the “border” and the “title bar”).

The window manager manages mouse clicks in the frame window. This allows for example to move or resize the window when the user clicks and drags on the border or on the title bar.

The window manager is also responsible for the handling of icons and related visual elements of the graphical user interface. Icons do not exist at the level of the X Window core protocol. They are implemented *by* the window manager, for example, whenever a window has to be “iconified”, the window manager FWM unmaps the window, and creates a window for the icon name and possibly another window for the icon image. The meaning and handling of icons is therefore completely decided by the window manager: some window managers such as *wm2* do not implement icons at all.

l) Session manager

Roughly, the state of a *session* is the “state of the desktop” at a given time: a set of windows with (their current content. More precisely, it is the set of applications managing these windows and the information that allow these applications to restore the condition of their managed windows if required. An X-session manager is a program that saves and restores the state of sessions.

The most recognizable effect of using a session manager is the possibility of logging out from an interactive session and then finding exactly the same windows in the same state when logging in again. For this to work, the session manager program stores the names of the running applications at logout and starts them again at login. In order for the state of the applications to be restored as well (which is needed to restore the content of windows), the applications must be able to save their state of execution upon request from the session manager and load it back when they start again.

The X-Window System includes a default session manager called `xsm`. Other session managers have been developed for specific desktop systems: for example, `kserver` is the default session manager of KDE.

m) X-display manager

The *X display manager* is the program that shows the graphical login prompt in the X-Window System. More generally, a display manager runs one or more X-servers on the local computer and accepts incoming connections from X-servers running on remote computers. The local servers are started by the display manager, which then connects to them to present the user the login screen. The remote servers are started independently from the display manager and connect to it. In this situation, the display manager works like a graphical *telnet* server: an X-server can connect to the display manager, which starts a session; the programs of this session run on the same computer of the display manager but have input and output on the computer where the X-server runs (which may be the computer in front of the user or a remote one).

XDM is the basic display manager supplied with the X-Window System. Other display managers include *GDM* (*GNOME*), *KDM* (*KDE*), *WDM* (using the *WINGs* widget set used in *Window Maker*) and *entrance* (using the architecture used in *Enlightenment v.1.7*).

n) User interface elements [X tool Kit] - It consists of two parts

i) intrinsic and ii) widgets.

Early **widget toolkits** for X included *Xaw* (the *Athena* Widget Set), *OLIT* (*OPEN LOOK* Intrinsics Toolkit) *XView*, *Motif* and *Tk*. *OLIT* and *XView* are the base toolkits for Sun's legacy *Open Windows* desktop environment.

Motif provides the base toolkit for the *Common Desktop Environment* (CDE), which is the desktop environment used on commercial *Unix* systems such as *Solaris*, *AIX* and *HP-UX*. (*Solaris 10* includes both CDE and *GNOME*, with the latter now the preferred desktop environment.)

More modern toolkits include *Qt* (used by *KDE*), *GTK +* (used by *GNOME*), *wxWidgets*, *FLTK* and *FOX*.

6.9 SOFTWARE DESIGN METHODS

The major software design methods are :

- Function-oriented Design
- Data structure Based Design
- Object oriented Design
- Reuse Based Design methods.

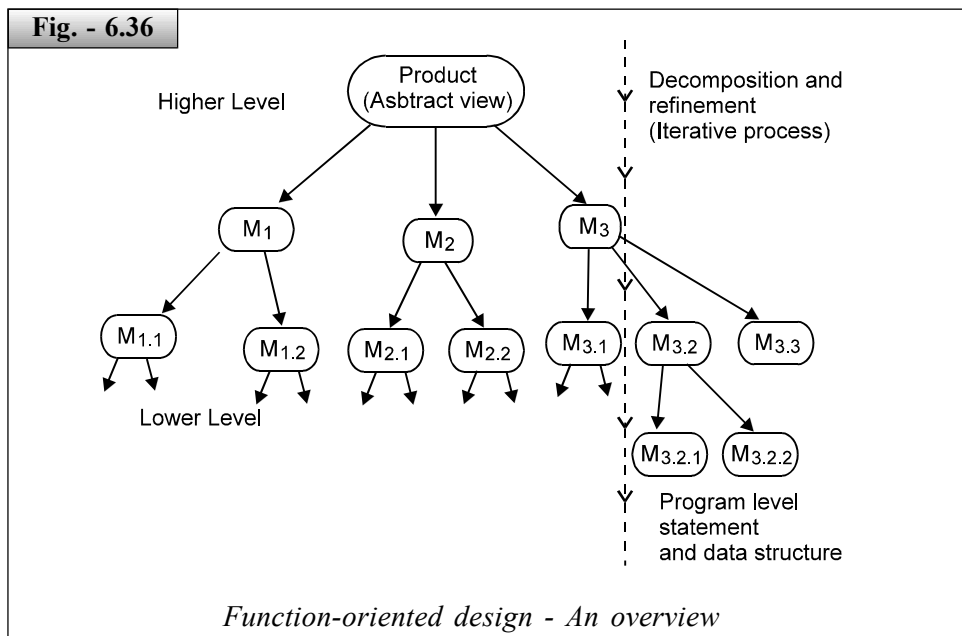
6.9.1 Function-Oriented Design

As the name suggests, this design method focuses on the functional aspects of the software product.

Function oriented design is an approach, where the system / product is decomposed into set of interactive modules and each module is clearly defined with its specific task (function)

This design method is described in a stepwise refinement policy as framed by Niklaus hirth, the crator of PASAL language and some more number languages, and it uses the top-down design strategy in a interactive way.

Initially, the high-level description of the main program is made. The program is then decomposed into number of meaningful program modules along with the related functionalities and the process continues in an interactive way for further modules with a refinement of function and related data structure till to reach the statement level of the programming language (coding phase) This is something, a process that proceeds from a highly conceptual model (abstraction) to lower level design details, and users procedures.



M_1, M_2, M_3 - decomposed modules

$M_{1.1}, M_{1.2}, \dots, M_{3.3}$ - decomposed and refined modules

$M_{3.2.1}, M_{3.2.2}, \dots$ - Lower level modules.

For effective function-oriented design, the best methodology mostly coming in use by the designers is structured Analysis and Design (SAD).

This methodology is intended to transform a textual problem description into a graphical model, what entertains the decomposition and refinement of program (abstract

model) into statement level of the programming language through top-down design strategy (no. doubt in an iterative way).

A set of graphical tools used for structure Analysis and Design of a program / product are :

- DFDs (Data Flow Diagram)
 - Data Dictionary
 - Structured Chart etc.
- (already discussed in Requirement)

6.9.2 Data Structure Based Design

The best known data structure based design methods were developed in 1970's.

These are :

- *Jackson System Development*
- *Warnier-Orr System Design.*

Michael Jackson created Jackson System Development (JSD) using the principles laid out in Jackson Structured Programming (JSP). JSD is a structured analysis and design method (Jackson 1975).

The technique introduced by Jackson is known as Jackson Structured Programming (JSP) and the technique proposed by Warnier-Orr is known as Logical Construction of Program LCP (Warnier 1977).

6.9.2.1 Jackson Systems Development

Essentials of JSP have been carried over to *Jackson Systems Development (JSD)*. The basic idea if JSP is that a good program reflects the structure of both the input and output in all its facets. Given a correct model of these data structures, the corresponding program from the model can be derived straightforward. Jackson distinguishes elementary and compound components. Elementary components are not further decomposed.

There are three types of compound components:

- *Sequence*
- *Iteration*
- *Selection*

Compound components are represented by diagrams also called Jackson Diagrams or Structure Diagrams.

The fundamental issues involved in the use of JSP are:

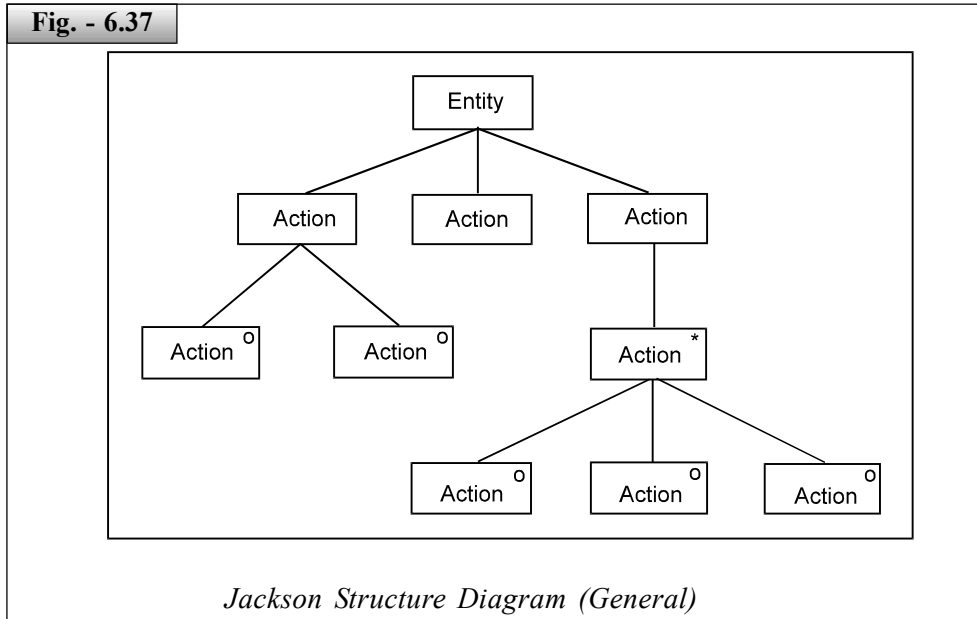
- *Modelling input and output using structure diagrams*
- *Merging these to create the program structure*
- *Resolving possible structure clashes*
- *Optimizing the result through program inversion*

JSD distinguishes three stages in the software development process:

- *Modelling Stage.* A modelling stage is that in which a description is made of real world problem through identification of entities and actions.
- *Network Stage.* In this stage, the system is modelled as a network of communicating concurrent processes.
- *Implementation Stage.* In this stage, the network of processes is transformed into a sequential design.

Jackson System Design (JSD) starts with analysis of a text description of the problem and acquisition of information about the application domain. Once again, objects and their interactions are identified.

Fig. - 6.37



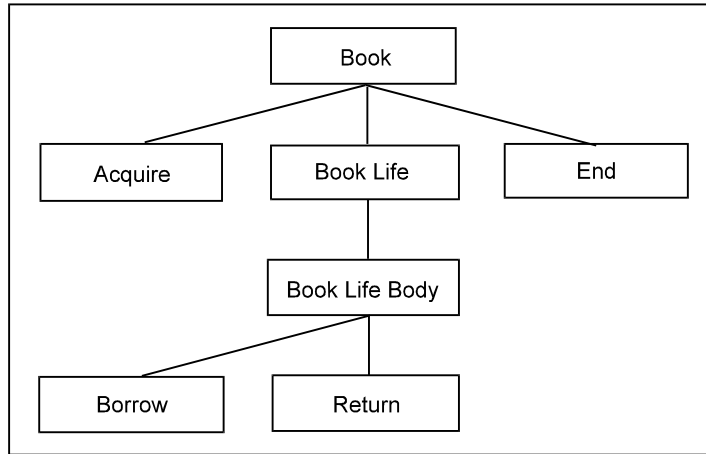
At this point, JSD models the lifetime of each object, represented both by a modified structure chart and by a kind of pseudocode called structure text, which includes control concepts. Jackson structure diagram is illustrated in above figure.

To represent a choice between two or more mutually exclusive actions, mark the actions with a small “o” (for option) on the upper right hand corner.

If an action is repeated, place a small asterisk (*) in its upper right hand corner. There is usually only one action under an iteration construct.

An example of Jackson structure diagram for Book entity is illustrated as in fig. given below.

Fig. - 6.38

*Jackson structure diagram (for book entity)*

The objects are then organized into a network representing interactions. Data streams in the network are classified, and additional functions are added as dictated by the structure text.

Timing and synchronization issues between objects are next dealt with. Finally, detailed design is carried out to simplify the design from an operational standpoint. The techniques of process backtracking, inversion, and state-vector inspection are provided to handle data ambiguity and operating system overload.

Jackson System Design (JSD) supporters claim that it is better than object oriented design, since it encompasses many of the same concepts and is more systematic. On the other hand, it isn't apparent that JSD would be as effective as object-oriented design in dealing with extremely complex problems. For simpler applications, however, JSD is certainly a competitor.

6.9.2.2 Warnier-Orr' System Design

LCP expanded into Logical Construction of Systems (LCS). Warnier diagrams are like hierarchical block diagrams: they depict information in a tree-like structure, but they allow indicating *repetitions* and *conditional occurrences*.

Here is an example for the structure of a newspaper: The 'text+indentation' notation might be adequate for the top layer description:

Front section

Headline news

National news

Local news

Editorial section

Editorials

Columns

Letters to the editor

Mocking cartoon

Second section

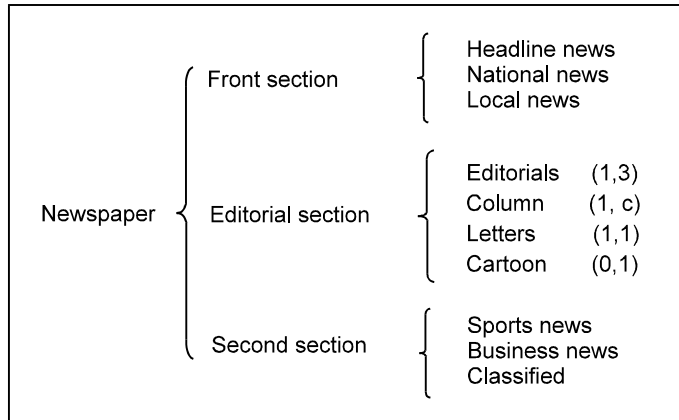
Sports news

Business news

Classified

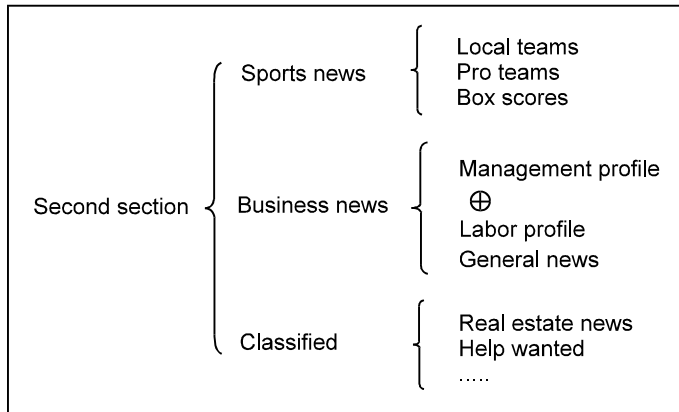
The Warnier notation allows being more precise:

Fig. - 6.39



Warnier Diagram of Main Structure of Newspaper

Fig. - 6.40



Warnier Diagram of Second Section

Later LCP expanded into the more comprehensive Warnier-Orr method Data Structured Systems Development (DSSD). The latter retains the data structure orientation

of LCP but also considers information flow and functional characteristics (Orr 1977; Orr 1981).

It has two main phases:

- *Logical definition and*
- *Physical definition.*

In the first phase the application context, functions and results of the system are defined. The second phase has five steps that are described as under:

- *Definition of the constraints imposed upon building the system. These may include aspects such as computer storage and processing capacity, reliability and security, and type of data base management system.*
- *Definition of alternative physical solutions, ranging from “do nothing”, buy an existing package, to develop the system in-house.*
- *Identification of the benefits and risks involved for each proposed solution.*
- *Selection of a solution.*
- *Preparation of the final requirements definition document.*

6.9.3 Object-Oriented Design Methods

Object-Oriented design (OOD) is the result of focusing attention not on the function performed by the program, but instead on the data that are to be manipulated by the program. Thus, object-oriented design is orthogonal to function-oriented design.

Object-Oriented technology is one of the latest approaches to software development, and it shows much promise in solving the problems associated with building modern software systems (Shlaer 1988; Meyers 1988; Rambaugh et. at 1991; Rubin 1992; Agha 1990).

Software engineers have become more aware of the importance of data structures. By using higher-level data structures beyond the basic integer, real, character, and boolean, algorithms can be significantly simpler and more elegant.

Data structures are a major force in promoting abstraction, at least comparable to procedures and functions. Complex, but familiar, objects can easily be conceptualised in designs and code. Complex behaviours can be referred to using natural sounding operation names. Such Abstract Data Types (ADTs) can function as software modules (Gutttag 1997). Previously, design methods have led people to assume that modules are procedural in nature. But ADTs **do** abstract concepts, simplify design, promote reusability, and define interfaces. ADTs **can** also be encapsulated and organized structurally into design architecture.

True object-oriented design occurs when ADTs are designed for flexibility and reuse, are encapsulated by excluding all implementation detail from the publicly visible interface, and are carefully organized in an architecture that is given shape by the relationship of visibility. Visibility exists between two objects when one can request services of another by invoking an operation that is part of the second object's interface.

Prototyping is an important aspect of design when systems employing object technology are built but, since it involves a whole new way of thinking about software development, most authorities in the software engineering community consider object technology as a paradigm itself. It is a programming paradigm that can best be applied with development tools that are compatible with it. O-O technology contains these three key aspects:

a) *Objects*

Software packages designed and developed to correspond with real-world entities and containing all the data and services required to function as their associated entities.

b) *Messages*

Communication mechanisms are established that provide the means by which objects work together.

c) *Methods*

Methods are services that objects perform to satisfy the functional requirements of the problem domain. Objects request services of other objects through messages.

d) *Classes*

Templates for defining families of objects and all the data and services that are common to them, and providing for the concept of inheritance that makes O-O software easier to modify and maintain than conventional software.

6.9.3.1 Benefits of OOD

The benefits of object-oriented development as claimed by its proponents are many: ‘

- ☛ *The concept of objects performing services is a more natural way of thinking.*
- ☛ *Objects are inherently reusable.*
- ☛ *Emphasis is on understanding the problem domain.*
- ☛ *Internal consistency of systems is improved because attributes and services can be viewed as an intrinsic whole.*
- ☛ *The characteristic of inheritance capitalizes on the commonalty of attributes and services.*
- ☛ *The characteristic of information hiding stabilizes systems by localizing changes to objects.*
- ☛ *The object-oriented development process is consistent from analysis, through design, to coding.*

6.9.3.2 Types of OOD Methods

The following are popular object-oriented design methods:

- *Booch's Object-Oriented Design*
- *Yourdon and Good's Object-Oriented Design*

The above methods are discussed more in detail in OOSE (object oriented software engineering) chapter.

6.9.4 Reuse-Based Design Methods

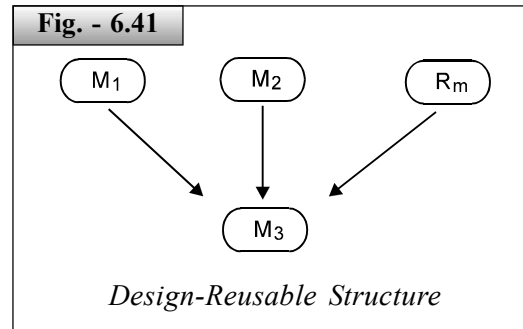
Reuse-based design accepts an existing partition of reusable modules, functions, or designs, but crafts an interface that ties them together in order to provide the specified software function.

For a module to be reusable, however, we must require that it should be used by *several* other modules as in design-reusable-structure, shown in Fig.

It is, of course, not necessary- to create a program top-down, even though its structure is function-oriented. However, if we want to delay the decision of what the system is supposed to do as long as possible, a better choice is to structure the program around the data rather than around the actions taken by the program.

M_1, M_2, M_3 - modules

R_M = Reusable module



The Unix filter provides a good example of reuse-based design by means of a toolkit. Unix filters represent highly encapsulated functions, or tools, which accept input and provide output in a standard format.

Each such function is designed to be both useful and primitive. The individual functions can then be assembled by gluing them together with pipes, using a shell language. If these tools are sufficiently varied and general, most of the tasks can be rapidly implemented. In case performance isn't as high, any individual tool can be rewritten for optimization, without modifying anything else.

Toolkit reuse generally depends on heavy encapsulation, a standard interface between tools, and a late-binding, interpretive language to tie the tools together. Encapsulation allows for re-implementation to achieve optimization, fault-tolerance, customisation, etc.

6.9.5 Criteria for selecting a software Design Method

The following are certain factors that may impact the choice of a particular software design method:

Experience

- Experience of the designers with a given method is always successful in applying that method because they are aware of the constraints and limitations of that method.

Problem Domain

- If the designers are well versed with the type of problem to be solved, a top-down technique or a technique based on data structures, may be very effective.
- If the design is experimental, one will go about it in a more cautious way, and a bottom-up design technique then seems more appropriate.

Philosophy

- Overall development philosophy is also another important factor.
- Many design methods are embedded in a wider philosophy that also addresses other aspects of “system development, ranging from ways to conduct interviews or reviews to full-scale models of the software life cycle.

Tools

- Availability of supporting tools is also an important factor in the selection and success of a design method.

6.10 INTRODUCTION TO SOFTWARE CODING

Coding is the activity where a design is converted to run-able code. Depending on the details of the design specifications, the coding activity may vary greatly.

Nowadays, there is little distinction between coding and module design. In fact, as the module design is being done, the result is usually directly typed in, in the form of code and then tested on-line. For that reason, coding sometimes may be thought of as part of the module design activity. Other issues involve simply the structure of the code itself.

Although one of the purposes of the coding activity is to convert a specification into a representation understandable by the machine, the more important purpose is to communicate ideas to other people.

Many companies have coding standards that all the programmers must follow. The purpose of a coding standard is to make it easier for programmers to read each other’s code, and to make it easier to maintain the code, i.e. it is important that the code should be written so as it could be modified later, if the need be.

When a programmer writes the initial code for a program, he or she has certain ideas on the overall structure of the program. These ideas determine how modifications should be done. Unfortunately, since the program is only *one example* of these ideas, it is usually not sufficient simply to read the program in order to understand these underlying ideas.

Therefore, the program must be accompanied by some kind of *documentation*, whether in the form of comments or in the form of a separate document. In both cases, the purpose of the documentation is *not* to repeat what is already known from the program itself, but to serve as a *complement* to the program. In short, the documentation contains what the program does *not* contain.

6.10.1 Coding Standards

Coding standards are a set of conventions that the programmer follows to standardize their computer code to some degree and to make the overall program easier to read (Mall 1999).

This will make your programs easier to read and understand when it comes time for grading, but it is also good programming practice (Dijkstra 1968; Jackson 1975; Aron 1974; Abbott 1983; Aron 1983; Dijkstra 1976). It will also allow you to quickly understand what you did in your program when you or others look at it in the future.

In other words, *coding conventions* as it is sometimes called is about how you code so that the code is easily readable and understandable by a human (Shooman 1983).

6.10.2 Coding Conventions

Although writing programs is an art, however, there should be some minimum conventions that should be followed while writing programs.

a) **Commenting**

- Program should begin with a preface stating:
 1. *Programmer's name.*
 2. *Date written.*
 3. *Course and assignment number*
 4. *A summary of description of functionality, i.e., what the program does and a brief description of how it does it.*
 5. *Any known assumptions built into the program.*
 6. *Any special instructions on how to run it.*
- All variables should have a brief comment explaining their purpose. Looping control variables such as: *x* or *i* need not be commented.
- Comments within code should explain tricky or complicated code.
- Comments should be clearly organized and easily distinguishable from the actual code.
- Comments should be comprehensible and useful. Comments such as: *This adds 1 to K*, are trivial and should be avoided.
- Each procedure or function should have a brief comment explaining the purpose of the procedure or function. The comments might also state the limitations of the procedure and/or function using possibly pre-conditions and post-conditions.
- Comments should be indented to at least the level of the statements they describe.

b) **Program Robustness**

- Do not check for equality among real numbers.
- Prevent the abnormal termination of a program due to input error, check for the validity of the data when appropriate.
- Programs should not be built around a specific data set. Programs, unless otherwise stated, should work correctly for any reasonable test data set.
- Do not use GOTO statements, or minimize the use of GOTO.

I/O Behaviour

- All I/O should use a proper format:
 1. All real number output should be formatted in decimal form unless scientific notation is appropriate.
 2. All input from the keyboard should be preceded by a prompt telling the user what format is expected. For example, *Please Input the Date in mm/dd/yy format.*

- Label all output values. Remember, the output of a very complicated and elegant program is useless unless one knows how to interpret the output.
- c) ***Modules and Modular Program Structure***
 - *Module coherence*: each module should correspond to one subtask in the overall algorithm to solve the problem.
 - *Module independence*: each module should be independent, i.e., it should be self-contained and it should perform its task successfully without needing to know the inner workings of the calling module.
 - Procedures or functions should, in general, occupy at most 50 lines.
 - Use appropriate parameters rather than global variables. Avoid side effects.
 - Hide any information from the caller that it does not need. The default should be to hide information unless the caller actually needs it.
- d) ***Program Testing***
 - The program should follow the program specifications delineated in the problem statement.
 - Testing should be sufficient to thoroughly check all parts of the program,
 - Make sure that the output of your program illustrates that it does in fact solve the problem.
- e) ***Cohesive Procedures***
 - Each procedure should be designed to complete a single task. If a procedure handles too many tasks then its logic can be hard to comprehend and bugs can easily occur.
Minimize Coupling
 - Parameter passing is good programming practice, but too many parameters and the code again becomes difficult to manage.
 - Procedures with many parameters are highly coupled; they have lots of links to other procedures.
 - Where possible try to minimize these links. However, global variables should not be used instead!
- f) ***Minimize data scope where possible***
 - Variables and arrays can be accessed by code in different parts of the program. Accessible anywhere is *Global*. This sounds ideal, but without care strange side effects can occur where code in other parts of a program ‘mess about with’ the contents of a variable by mistake.
 - Restricting the access range, or scope, of a variable or array can solve this problem,
 - An intermediate scope is access to a variable throughout a single module. This means any procedure in the module can access the variable, but procedures in other modules are denied access. And the most restricted scope *is* local.
 - These are variables and arrays that can only be accessed from within the subroutine in which they were defined; even other subroutines within the same module can no access these variables.
 - Where possible limit scope as much as possible.

6.10.3 Programming Style

Programmers have their own style of writing programs. Good programming style makes the code easy to understand, not only for other people but also the programmer himself (hoar 1969; Aron 1974; Dijkstra 1976; Jackson 1975; Aron 1983).

6.10.3.1 Importance of Programming Style

Programming style is important for the following main reasons :

To simplify software maintenance

- Readability implies clarify and understanding
- Programs have to be read to be maintained.

To avoid problems

- Good style can avoid common bugs and typos
- Good style can make mistakes clear

To achieve simplicity

- Consistency is simplicity

To achieve readability

- By yourself and others
- Now and a year from now

6.10.3.2 General Program Style

- If you have to use global variables, use long descriptive names.
- Never guess about data structure sizes.
- *Variable names* should be mnemonic, clear and simple, e.g., *largest* rather than *x*.
- *Expressions* should be clear and simple.
- 1. Indent bodies of structured statements at least 3 spaces.
- 2. Leave blank lines between logically related groups of statements, after the declarations, and between functions.
- 3. Clearly delineate subprograms (procedures, functions, modules).
- 4. Wrap long lines at no more than 80 characters and indent the continued statement further than the beginning of that statement
- Make use of the *features of the language*, i.e., if the language has a useful built-in feature, use it.
- *Procedure and function names* should be mnemonic.
- *Local variables* should be declared at the beginning of the function.
- Use Purify to plug memory leaks and detect memory access violations.
- If what you're doing seems clever, think of another way to do it.
- Make core dumps impossible.

6.10.3.3 Good Programming Style

There are several components of good style:

- *Naming Conventions (variables, controls, and subroutine names)*
- *Indented and properly spaced code*
- *Documented code*
- *Cohesive Modules*
- *Minimum Coupling*
- *Minimize data scope where possible ,*

6.10.3.4 Good Programming Style Aids

Good programming style aids the following:

Modifiability

- Code must be easily understood and modified.
- Almost each code is modified during its lifetime.
- A good style will help people to **easily** modify other's code.

Transportability

- To increase productivity and usability, a program should be easy to transport to different environments and computers.
- A good style of programming is to isolate all machine dependent areas of code and label them as such.
- Input/Output is a typical area where machine/operating system dependencies surfaces

Readability

- Use of good indentation, blank lines, and comments aid in reading sections of code.
- To improve readability, use identifiers that mean something or are relevant to the values they represent.
- Separate blocks of self contained code with spaces, comments, and blank lines.
- Indent segments of code in loops or in conditional statements.
- Make good use of procedures and functions to modularise program.
- If it has to be modified or transported it must be able to be understood easily by the modifier/transporter.

Usability

- The user should be presented with a consistent, easily understandable, and a friendly interface.
- Use error recovery techniques that help the user. Also make certain that your error messages are consistent and helpful.
- Make certain that your prompts are consistent and contain all needed information for the user.
- A frustrated user will blame the programmer rather than him/herself.
- *User Interface Management Systems* are becoming increasingly important in computing especially with the advent of graphical workstations with windowing systems.

Efficiency

- A well thought out program is nearly always an efficient program.
- Efficiency is not just speed but takes several aspects into consideration.
- Optimise your code without reducing readability to zero. This is efficiency with restrictions.

6.10.4 System Verification

The purpose of verification is to verify that the product works according to the specifications ped in the feasibility study and the design phase.

There are essentially two ways to accomplish this:

- *Testing the system, i.e. running the system on certain inputs and observing the behaviour of the system.*
- *Proving the correctness of the system*

The first method cannot ensure the correctness of the system. However, if a problem is found one can be certain that the system is not correct.

The second method is able to ensure the correctness of the system. Unfortunately it requires a formal specification of what the system is supposed to do. Such a specification is likely to be as large and as error prone as the system itself. The specification itself thus needs verification.

Testing, although unable to cover all possible cases, is able to make sure that a large subset of cases work, thus increasing our confidence that the system works as specified.

Proving the correctness of the program is transformed into increasing the confidence in the structure of the system. To accomplish such confidence, the design and the coding of the system needs to be done in a well-disciplined and structured way. Therefore, the verification activity must be prepared during the specification and coding activities. It is hard, if not impossible, to have any level of confidence in a product that is badly structured or badly coded.

Finally, verification is used in order to verify other aspects of the system, e.g. performance. This aspect is particularly important in real-time systems where performance may determine whether the system is correct or not.

As mentioned above, verification must be an integral part of the design and the coding activities. However, there are tasks that can be thought of as particular to the verification activity.

The following are two major tasks involved in verification:

- *Program (or Module) Testing*
- *Reviews of Design and Code*

Both above tasks are discussed below.

6.10.4.1 Program Testing

Testing the program is a pure verification activity. In order for tests to be useful, they must be put in system. It is not sufficient to try a few cases and conclude that it

works. On the other hand, it is impossible to cover all possible cases. A practical rule is the following:

Make sure Each statement of the system has been executed at least once by at least one test case.

Clearly, if this is not the case, one cannot pretend to have tested the system.

But even if you have executed every statement in the system, it is not clear that the system corresponds to the specification. For instance, if some cases simply are not treated at all, although specified in the requirements specification, then executing *all* statements will not help, since no statement treating the omitted case is present in the code. Therefore, test must always be a combination of testing the code present in the system, and testing the system with respect to the requirement specification.

Testing can be done on each separate module of the system. In that case, it is called *module testing*.

It can also be done on the entire system. In that case, it is called *system testing*.

System testing often involves putting together a few modules at a time and making sure that the subsystem is tested before a new module is added. For that reason, system testing is sometimes referred to as *integration*.

An error discovered during module test usually indicates an error during the coding phase. Errors of this sort are in general not too hard to correct. An error discovered during integration sometimes indicates a design problem, and may be extremely hard to fix.

Testing methods fall into two categories: *black box* and *white box* testing, which are discussed in detail in software testing chapter.

Black box testing is done without knowledge of the internal structure of the system and only external behaviour is tested.

White box testing, on the other hand, means that the structure of the product is taken into account, when the tests are constructed. That way the hope is to discover cases those are hard for the system to handle.

Black and white box testing may be applied both to module and system test.

6.10.4.2 Reviews of Design and Code

Another task that can be thought of as pure verification is to conduct *reviews*. These are extremely useful activities and they fall into the second category of verification, namely *proving the correctness of the system*.

Usually, the designer or the programmer presents his/her design or code to a committee. The committee tries to discover potential problems with the design or the code. Everybody must be convinced that the purpose of the review is to increase the quality of the product, and that this task is a group effort.

6.10.5 Code Inspections

Code Inspection technique is well known since 1976 and *is* a silver bullet that can drastically improve the code while reducing delivered bugs. Code inspections are one of the most powerful tools for eliminating bugs (Eagan 1976; Myers 1978).

Formal code inspections are probably the most important tool you can use to get your code out faster with fewer bugs. The inspection plays on the well-known fact that “*two heads are better than one*”.

The goal of Code Inspections is to identify and remove bugs before testing the code.

The best code inspections come about from properly organized teams. Experience indicates that when management is involved usually only the most superficial bugs are caught, since no one wishes to show the author to be the cause of major program defects. Four formal roles exist: the **Moderator, Reader, Recorder, and Author**.

- The **Moderator**, who must be very competent technically, leads the inspection process. He or She paces the meeting, coaches other team members, deals with schedule a meeting place and disseminating materials before the meeting, and follows up rework (if any).
- The **Reader** takes the team through the code by paraphrasing its operation. Nilet the Author take this role, since he may read what he meant instead of what implemented.
- A **Recorder** notes each error on a standard form. This frees the other team members to focus on thinking deeply about the code.
- The **Author’s** role is to understand the errors found and to illuminate unclear areas. As code inspections are never confrontational, the Author should never be in a position of defending the code.

6.10.5.1 Code Inspection Process

Code inspections are a *process* consisting of following several steps that are required optimal results.

Planning

- When the code compiles cleanly with no errors or warning messages, the At submits listings to the Moderator, who forms an inspection team.
- The moderator distributes listings to each team member, as well as other relevant documents such as design requirements and documentation.
- The Moderator does the bulk of the Planning process.
- An effective Moderator respects the time constraints of his colleagues and avoids interrupting them.

Overview

- This optional step is a meeting for cases where the inspection team members are familiar with the development project.
- The Author provides enough background to team members to facilitate ‘ understanding of the code.

Preparation

- Inspectors individually examine the code and related materials.
- They use a checklist to ensure they check all potential problem areas.
- Each inspector marks up his or her copy of the code listing with suspected pro areas.

Inspection Meeting

- The entire team meets to review the code.
- The Moderator runs the meeting tightly.

- The only subject for discussion is the code under review.

The person designated as Reader presents the code by paraphrasing the meaning of small sections of code in a context higher than that of the code itself.

All errors are recorded and classified as *Major* or *Minor*. A *Major bug* is one that if removed could result in a problem that the customer will see. *Minor bugs* are those that include spelling errors, non-compliance with the firmware standards, and poor workmanship that does not lead to a major error.

Rework

- The Author makes all suggested corrections, gets a clean compile and sends it back to the Moderator.

Follow-up

- The Moderator checks the reworked code. If the Moderator is satisfied the inspection is formally complete and the code may be tested.

6.10.5.2 Checklist for Code Inspections

A typical checklist for Code Inspections is given as under:

1. Data Declaration

- (a) All variables defined?
- (b) Correct lengths, types, and storage classes assigned?
- (c) Initialisation consistent with storage class?
- (d) Arrays and strings initialised properly?
- (e) Any variables with similar names?

2. Data Reference

- (a) Subscript within bounds?
- (b) Record and structure attributes match?
- (c) Off-by-one errors in indexing or subscripting operations?
- (d) Correct attributes when aliasing?
- (e) String limits exceeded?

3. Computation

- (a) Mixed-mode computation?
- (b) Computation on non-arithmetic variables?
- (c) Computations on variables of different lengths?
- (d) Intermediate result overflow or underflow?
- (e) Division by zero?
- (f) Target size less than size of assigned values?
- (g) Variable's value outside meaningful range?

4. Comparison

- (a) Comparison between inconsistent variables?

- (b) Comparison relationship correct?
- (c) Boolean expressions correct?
- (d) Operator precedence understood?

5. Control Flow

- (a) Multi-way branches exceeded?
- (b) Will each loop terminate?
- (c) Will program terminate?
- (d) Any loop bypasses because of entry conditions?
- (e) Off-by-one iteration errors?

6. Input/Output

- (a) File attributes correct?
- (b) Open statements correct?
- (c) Format specifications match I/O statement?
- (d) Buffer size matches record size?
- (e) End-of-file conditions handled?

7. Interfaces

- (a) Number of input parameters matches argument list?
- (b) Types of input parameters match the argument variables?
- (c) Global variable definitions consistent across modules?
- (d) Constants passed as arguments?

8. Other Checks

- (a) Missing function?
- (b) Input checked for validity?
- (c) Any unreferenced variables in cross-reference listing?
- (d) Are compound conditions specified properly?

6.10.5.3 Benefits of Code Inspections

The following are some benefits of code inspections:

- *Indicates all the defects that otherwise are difficult to find during normal operational testing and usage.*
- *Lists all potential design flaws that can make software code less maintainable and costly to develop.*
- *It generates source code documentation that can make it easier for people to add new code and maintain existing code.*

6.10.6 Code Reviews and Walkthroughs

Regular code reviews are a very useful tool in development as people tend to overlook mistakes that they have implemented themselves. Code reviews also help to spread expert

knowledge throughout a development team. The suggestions and ideas which evolve during review sessions often take the process big steps forward.

Using various reviews, inspections, and code walk-throughs in a software development project almost always has a positive affect on software quality, readability, and ease of understanding (Pagan 1976; Myers 1978).

6.10.6.1 Rules for Code Reviews and Walk-throughs

Important rules for conducting effective code reviews and walk-throughs are:

- *The duration of a code review should be of a reasonable duration. A balance should be struck between the length of the review and the attention span of the reviewers.*
- *Participants must commit not only to attend the actual review, but also to read the material and attend a follow-up meeting to hear the presenter's response to comment made at the initial code review. .*
- *Meeting coordinators must not let the meeting degenerate into a problem-solving session. Problems should be identified and assigned as action items to discuss in a follow-up meeting, after a solution has been devised.*
- *The code review is designed to detect errors in the code or in the design, not to correct them.*

6.10.6.2 Benefits of Code Reviews and Walkthroughs

- *The earlier bugs are found in the life cycle of a product, the cheaper and easier they are to fix.*
- *If someone else looks at your code or design, they are likely to find mistakes you missed.*
- *When you know that someone else is going to be looking at your code, you are a lot more likely to tidy it up and make sure there is accurate and up-to-date documentation.*
- *You can learn a lot by reading other people's code.*
- *More than one person who is familiar with a program is the best insurance against "Mack Truck Syndrome", which is when the only person who understands the software gets hit by a truck, leaves the company or is for some reason not available for consultation.*
- *It can be a means of establishing quality metrics, so one can measure the effectiveness of different quality processes.*
- *The process of explaining your software to someone else can help you actually review your own program, rather than just looking at and seeing what you expect or want to see.*
- *When done right, code and design reviews can save time and improve quality over the entire project life cycle.*

6.10.6.3 Limitations of Code Reviews and Walkthroughs

The disadvantage of code reviews is that they take time, not only of the person

actively working on the project, but also for other people who are usually under deadline pressure themselves. Despite the fact that there are numerous studies showing that the overall number of human-hours spent on a project is lower when reviews are properly done, there is the constant temptation to bet that there really aren't any problems. That means waiting until the code is written and being debugged to try to find problems.

6.10.7 Coding Tools

The tools used in the coding phase should be familiar to most of the readers.

Most of the time a *text editor* is used to enter the program, a *compiler* is used to translate the program into machine code, and a *debugger* is used to find errors. Sometimes a source code control system (for instance SCCS or RCS) is used to keep track of the sources and to make sure that it is possible for several people to work on different parts of the program.

Other useful tools are *interpreters*, *parser generators (lex/yacc)*, or *programs* that keep track of what parts of a program have been altered (make).

6.10.8 Documents Generated From Coding

The main document resulting from the coding activity is the code itself. Contrary to what some people think, the main purpose of the code is as a document to be read by other programmers.

The fact that the code also serves as an input to a translator that generates machine-readable code, is secondary. Therefore, it is important to structure the code so that it is readable to humans. Comments are an important part of the code.

The code itself is usually supplemented by some other documentation. The purpose of this documentation is to explain choices that were made during the coding phase. That way, the program can be altered in a clean and consistent way during the maintenance phase.

SUMMARY

- The intent of software design in the field of software development, is to apply a set of principles, concepts and practices that lead to the development of high quality software product. The goal of software design is to create a model of software that will implement all customer requirements correctly and bring pleasure to those who use it. Software design can also be sensed as iterative process which transforms SRS to SDD or DDS i.e. from conceptual design to technical design.
- Software design process moves from a big picture view of software product to a more narrow view that defines the detail required to implement a system. This process is basically decomposed into three levels i.e. Architectural, Interface and detailed design. Where the conceptual integrity of the software system gets transformed into detailed specification of design. Though a set of process like modularisation, interfacing, fixing of functional responsibilities of modules, scope and visibility, algorithms and data structures etc.

- The quality of software design representations (SDD or DDS) is basically evaluated through predefined technical criterion. Some primply like modularised partitioning, coupling, cohesion etc. with respect to their preference level are entertained in software design. Design methods like function orient design, object oriented design, reuse based design are applied as per the product.
- Out through certain policies i.e. trade off analysis, complexity and through specific software like Unicon, Aesop, Acme etc.
- Software Design Document (SDD) is framed as per the recommendation of IEEE.
- User interface design is one of the major characteristics of software design process, which may character based or graphical user used interface and are accomplished through Golden rules. User interface may also be command language based, menu based or direct manipulation based. Quality of an user interface design is measured through its simplicity, consistency, intuitiveness, etc.
- Graphical user Interface (GUI) is the real advancement in the field of software design mostly accepted by the customers and is facilitated using X-windowing Techniques. Coding is a technique to put life into the DDS and comes under low level design and also supported through testing, certification and implementation. It provides a good programming style under a suitable programming environment. Program testing, reviewing of design and codes, code inspection etc. are the major activities carried out under this coding phase along with documentation, coding tools to develop the software product to the entire satisfaction of the customer.

EXERCISE

1. What is design ? Describe the difference between conceptual design and technical design.
2. Discuss the objectives of software design. How do we transform an informal design to a detailed design ?
3. Do we design software when we "write" a program ? What makes software design different from coding ?
4. What is modularity ? List the important properties of a modular system.
5. Define module coupling and explain different types of coupling.
6. Define module cohesion and explain different types of cohesion.
7. Discuss the objectives of modular software design. What are the effects of module coupling and cohesion ?
8. If a module has logical cohesion, what kind of coupling is this module likely to have with others ?
9. What problems are likely to arise if two modules have high coupling ?
10. What problems are likely to arise if a module has low cohesion ?
11. Describe the various strategies of design. Which design strategy is most popular and practical ?

12. If some existing modules are to be re-used in building a new system, which design strategy is used and why ?
13. What is the difference between a flow chart and a structure chart ?
14. Explain why it is important to use different notations to describe software designs.
15. List a few well-established function oriented software design techniques.
16. Define the following terms: Objects, Messages, Abstraction, Class, Inheritance and Polymorphism.
17. What is the relationship between abstract data types and classes ?
18. Can we have inheritance without polymorphism ? Explain.
19. Discuss the reasons for improvement using object-oriented design.
20. Explain the design guidelines that can be used to produce "good quality" classes or reusable classes.
21. List the points of a simplified design process.
22. Discuss the differences between object oriented and function oriented design.
23. What documents should be produced on completion of the design phase ?
24. Can a system ever be completely "decoupled" ? That is, can the degree of coupling be reduced so much that there is no coupling between modules ?
25. Discuss the importance of software design in software engineering.
26. Comment upon 'Software design as a problem solving'.
27. What is meant by software design strategy?
28. What is software design process? Illustrate through a suitable example.
29. Indicate the importance of interface design in overall software design process.
30. Differentiate between architectural design and detailed design.
31. Differentiate between good and bad design. 26. What is meant by a module?
32. What is meant by software design strategies? State their importance.
33. What is top-down design? State its objectives and usefulness.
34. What is bottom-up design? Discuss its benefits and limitations.
35. What are Jackson structure diagrams (JSD)? Illustrate their usefulness through suitable examples.
36. What do you mean by reuse-based design methods? Discuss their benefits.
37. Outline the criteria for selecting a software design method.
38. What do you mean by coding? State its purpose.
39. What are coding standards? Illustrate their usefulness.
40. What are coding conventions? Enumerate the coding guidelines.
41. What do you understand by programming style? Discuss its importance.
42. Illustrate the benefits of a good programming style.
43. What are the important elements of a good programming style? Explain.

44. What do you mean by code inspections? Discuss its benefits.
45. What do you understand by user management system?
46. What are the ways to improve readability? Explain.
47. What is meant by program robustness? Justify.
48. Enumerate the steps involved in code inspection process.
49. What is meant by verification? Explain its usefulness.
50. What do you mean by usability? State its importance.
51. What is the role of 'moderator' in code inspections? Explain.
52. What do you understand by module coherence?
53. What are the ways of performing verification? Discuss.
54. What are follow-ups? Who is responsible for these? Explain.
55. What do you mean by reviews of design and code? Discuss its benefits.
56. Enumerate the general programming guidelines.
57. How can we minimize scope of data? Explain.
58. What contributes to the efficiency feature of a programming style? Explain.
59. Give some good examples of coding tools and outline the specific purpose of each.
60. Outline the checklist for code inspections.
61. What are the steps involved in verification process?
62. What do you understand by code reviews? State their usefulness.
63. What are code walkthroughs? Illustrate their purpose.
64. Are there any limitations of code reviews and walkthroughs? Justify your answer.
65. Is verification a part of design or code or both? Justify your answer.
66. Discuss the benefits and limitations of code reviews and walkthroughs.
67. What are coding tools? Which of these preferred most and why? Illustrate.
68. What are the documents generated by coding process? Illustrate.
69. Who performs design and code reviews? Explain.
70. What do you mean by review for correctness? State its usefulness.
71. Illustrate the importance of coding in the overall software development life-cycle.



Software Testing

7.1 INTRODUCTION

Over the years, software has become a familiar and crucial component of almost many systems used in day-to-day life. System failure in any field causes loss on human lives, environment, economy etc.

[e.g. Systems like flight control, monitoring nuclear reactor, medical applications etc.]

As software is one of the vital components of the system, it needs to be tested in a well planned manner.

Software testing is a process of evaluating a system or system component by manual or automatic to “verify that it satisfies the requirement level as desired or to identify the deviation in the performance of a system, considering the expected and actual results.

But here in software engineering, *the real intension of software testing is to identify the errors and to take necessary measures for avoidance / elimination of the errors.*

7.2 TESTING AND SDLC : AN INTER-RELATIONSHIP

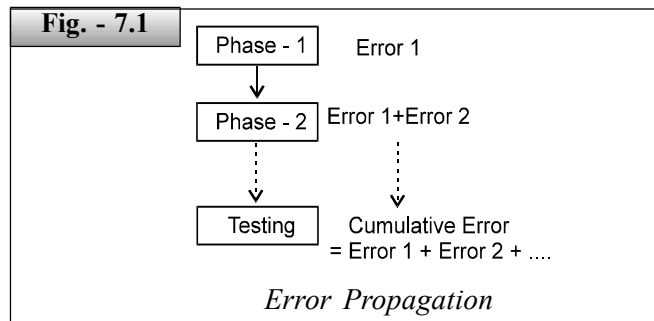
Software Testing is an integral part of the Software process. It is an activity which must be carried out throughout the SDLC.

Each phase in SDLC has a distinct end product such as SRS, DDS, etc. Each end product must be checked with the previous phase w.r.t. the actual requirements.

So, it is more advised to introduce the testing techniques in each and every phase of the SDLC. Else there is possibility of error incurred in the early phase / stage of the software project and it propagates from phase to phase resulting cumulative error; (as shown in the fig. 7.1)

It is too difficult for detection and removal of error. Ultimately the estimated cost of errors will be very high, which is not desired.

e.g. : In a banking system the potential cost of even a minor Software could be very high.



A list of different testing types required during different phases of SDLC are briefly given below :

- Requirements** : Requirements must be reviewed with the client; rapid prototyping can refine the requirements.
- Specification** : Specification document must be checked for feasibility, traceability, completeness, and absence of contradictions and ambiguities.
- Design** : The design must be checked for logic faults, interface faults, lack of exception handling etc.
- Implementation** : Code modules are informally tested by the programmer. Formal testing of modules (by testing team) i.e. non-execution-based methods and execution based method.
- non-execution based* : Code inspection and work through.
- execution based* : Black box and white box testing.
- Integration** : Appropriate testing Strategy, (Top-down/Bottom-up) for combination of modules together correctly to achieve a product w.r.t. the desired specification.
- Product testing** : Functionally checking of the whole product
Source code and documentation are checked for completeness and consistency.
- Maintenance** : Modified version of original product must be tested to ensure the correct implementation of changes.
Introducing regression testing (discussed later)

7.3 TESTING TERMINOLOGIES

Let's list out some terms used interchangeably in software testing.

- Error** : This may be syntax error or misunderstanding of specifications.
- Mistake** : Any thing not expected to do.
- Bugs** : It is resulted from mistakes (software and hardware bugs)
- Fault** : Representation of an error.
- Failure** : Execution of fault results failure.
- Verification** : It is primarily related to manual testing, because it requires looking out the documents and reviewing them.

It can also be defined as “*a process of evaluating a system on component to determine, whether the products of a given development phase / stage satisfy the condition imposed at the start of that phase.*”

Validation : It’s a process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specific requirements.

Debugging : It’s a process of analysing and locating bugs when software product does not behave as expected. A methodological approach to software testing is to identifying bugs.

Debugging is therefore *an activity that supports testing, but can’t replace testing.*

Test, test case and test suite :

Test and test case terms are used interchangeably, in practice both are same. Test case describes an input and expected output. Inputs are of two types : *pre conditions* (situation prior to the test case execution) and *actual inputs* (identified by testing methods). Expected outputs are also of post conditions and actual outputs.

During testing, necessary pre-conditions are set, required inputs are given and outputs are compared with the expected output to know the outcomes of a test case.

If expected and observed outputs are different, then there is a failure and must be considered to find out the cause of failure.

A good test case has high probability of finding an error. Test cases are useful and need to be developed, reviewed, used, managed and recorded.

A set of test cases is called test suite.

7.4 DEFINITIONS OF SOFTWARE TESTING

- ☛ Testing is a process of carrying out verification and validation of a software product.
- ☛ It’s a process of analysing a program with the intent of finding errors (Myers 1979)
- ☛ Testing is the process of analysing a software by the developers to detect the difference between existing and required condition (i.e. bugs) and to evaluate the features of the software items. (IEEE 1993)

7.5 PRINCIPLES OF TESTING

There are many principles those guide software testing before applying methods to design effective test cases.

A software engineer must follow up the basic principles of software testing while executing the testing phase. Following are the main principles of software testing.

1. All the tests are based on the user’s needs.

This principle detects any defects related to the software product on system. It also checks however the system (product) satisfies the user’s need or fails to meet the user’s requirements.

2. Tests should be planned very carefully.

This principle states that the test should be planned long before the beginning of the

testing process. The tests are planned, quickly after the completion of the requirement model [phase where SRS gets converted to DDS]

3. *Testing must be based on the part to principle. Pareto principle :*

It implies that 80% of all errors uncovered during testing will likely be traceable to 20% of all program components.

There is a problem to isolate the suspected components from the whole system and test them thoroughly.

4. *Testing should begin “in the small” and progress towards testing “in the large”.*

This principle implies that, firstly, test is planned and extended on individual components and as the testing progresses, the errors are eliminated and clustered with more number of components and ultimately the entire system is encountered.

5. *Third-party verification is an effective testing technique :*

This implies that in case of large project (product), the testing must be effective, if the test is done by an independent tester. He/She must learn about the whole system, the policy to modularize and partitioning the system without affecting the quality of the software product.

6. *Testing should be “non-exhaustive type” :*

It implies that the no. of path permutations for a standard sized program is exceptionally large.

Therefore, it is impossible to execute each combinations of paths during testing. But it is possible to adequately cover the program logic and to ensure that all conditions in the component level design have been executed.

7. *Testing time and resources are limited :*

It implies how to avoid redundant test.

7.6 OBJECTIVES OF TESTING

The objectives of testing is to test the codes and uncover all the errors.

It also demonstrates whether the components of a software product are working satisfactorily according to the Software Requirement Specification (SRS) with respect to the functionality, facilities and performance.

If the codes do not refers to the SRS and DDS, then the errors are detected.

The testing objectives are :

- it is intended to findout the error.
- An effective test case, finds out the as-yet undiscovered errors.

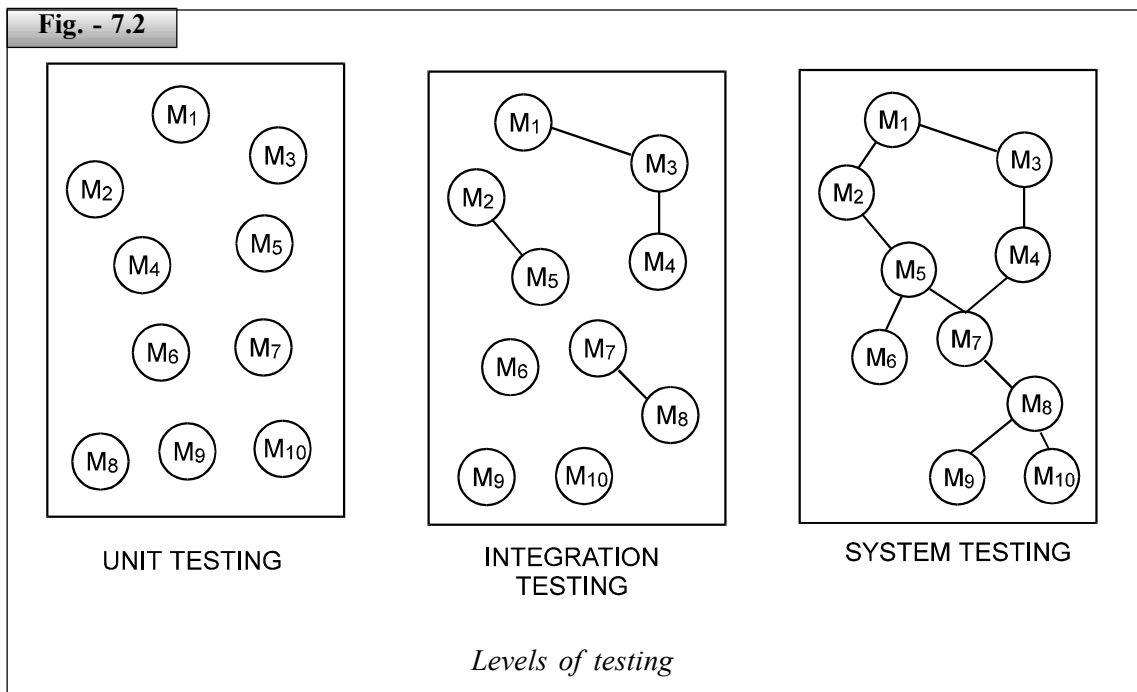
7.7 LEVELS OF TESTING

There are three levels of testing.

- a) Unit testing
- b) Integration testing.
- c) System testing.

These above levels are specified w.r.t. the individual module to the entire software system.

Fig. - 7.2



$M_1 \dots M_{10}$ = Modules

The levels of testing are selected w.r.t. the DDS for a particular software. It signifies that a particular level of testing is applicable for a particular test case.

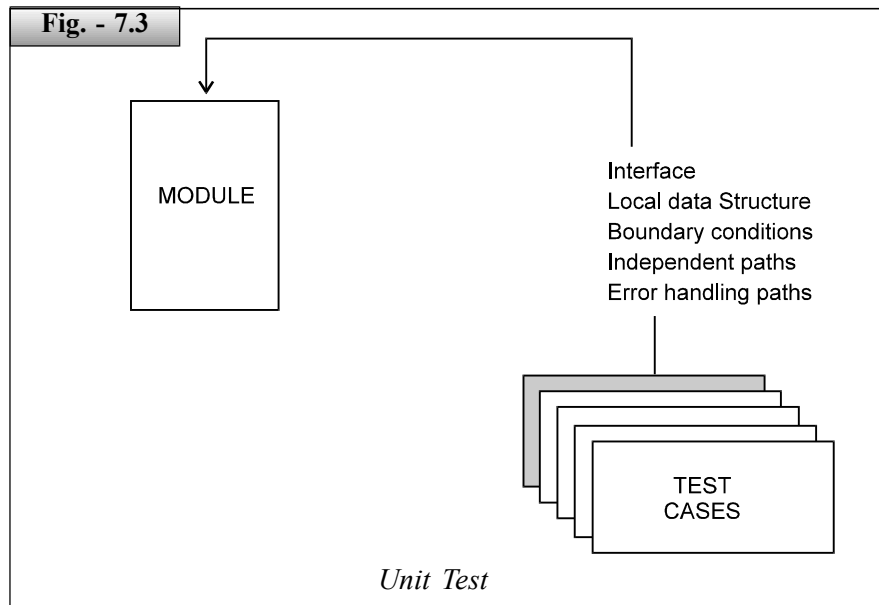
7.7.1 Unit testing

As a software system consists of or comprises a number of distinct modules or components, hence in unit testing, individual components are tested to ensure that they operate correctly or not. It enables to results a verification report to the designer about the correctness of the tested component. When unit testing is done over a component of the software system, it must need to be isolated from others.

There are no. of reasons in support of unit testing instead of testing the entire software product.

- The size of a single module (component) is so small that the errors are easily detected.
- As the size of the module (component) is small, so the process of unit testing is more effective.
- In case of multiple errors exist in the system component, the errors are easily eliminated.

Unit testing procedure



In unit testing, the interfaces of the module are tested to ensure that the information is properly flown in to and out of the program units, those are under test.

- The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in execution of an algorithm.
- Boundary conditions are tested to ensure that the module operates properly at the boundaries specified to restrict the processing activity.
- All the independent paths through the control structure are exercised to confirm that all statements in a module have been executed atleast once.

And finally all the paths for detecting the errors are tested minutely.

The following errors are eliminated during computation (examining codes)

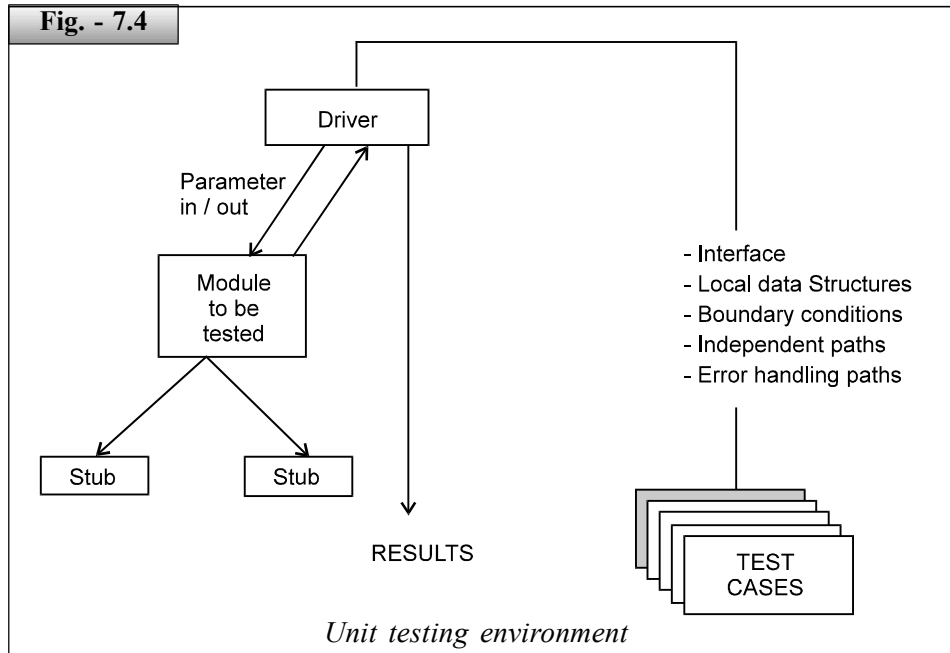
- Precision inaccuracy
- Mixed code operations
- Incorrect arithmetic precedence
- Incorrect symbolic representation of an expression
- Incorrect initialization.

In unit testing, when a test case is applied on a particular component of the whole system, the following errors are detected.

- Failure to exit when divergent iteration is encountered.
- Comparison of different data types.
- Expectation of equality when precision error makes equality unlikely.
- Incorrect logical operators or precedence.

- Incorrect comparison of variables.
- Improperly modified loop variables.
- Improper loop termination.

Principles of unit testing



In unit testing, “a driver” (main program) accepts the test case data and transfers the data into the component which is intended to test and prints the resulted output.

A *stub* is there to replace the modules or components that are subordinate to the tested components i.e. the subordinate components may be the components (called by) nearby it. The stub is used at the interfaces of the modules. It manipulates the data, prints the verification report of input and returns control to the module or component under testing.

The “drivers” and the “stub” both are the software those are written for test cases but must not be delivered with the final software product.

“The simplicity of the test cases depends upon the simplicity of the stub and the driver”. The unit test is more effective when it is applicable to a software component with high cohesion (functional cohesion).

7.7.2 Integration testing / Interface testing

It is the second level testing.

Integration testing is a systematic technique for constructing the program structure at the time of conducting the test to detect the errors. It is also called as an *interface testing*.

In integration testing, a number of units (tested units) or modules are combined into a no. of subsystems. Each subsystems are tested again to ensure that how far they are integrated/associated properly.

* Basically, integration testing tests the coupling among the modules.

Objectives of integration testing :-

The objectives of integration testing is “to test the interfaces of the module with a view to ensure that there is no error in the parameter passing when one module invokes another module”

During integration testing, several modules of the software system are integrated w.r.t. the integration plan. The integration plan specifies the steps and the orders in which modules are combined to represent the whole system.

After each step of integration testing, the system is partially tested so as to make it more effective.

Integration testing approaches :-

There are various approaches related to integration testing such as

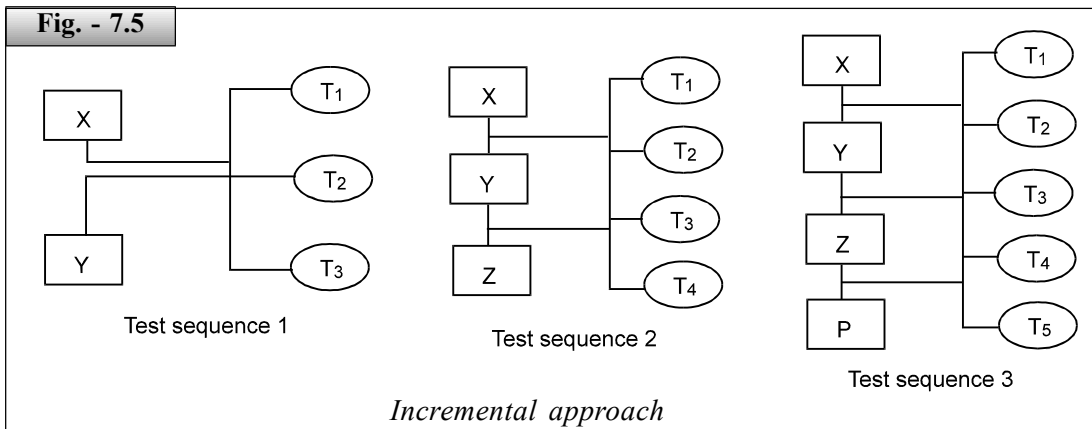
- Incremental approach
- Top-down integration
- Bottom-up integration
- Regression testing
- Smoke testing
- Sandwich intregration

1. Incremental approach :

Incremental means “combine the 2 things together and move forward”.

In this regard, at least two components of the system are combined together and the test is conducted. Hence, gradually the whole system is tested automatically.

In incremental approach, it is easy to isolate and detect the error from the whole system correctly.

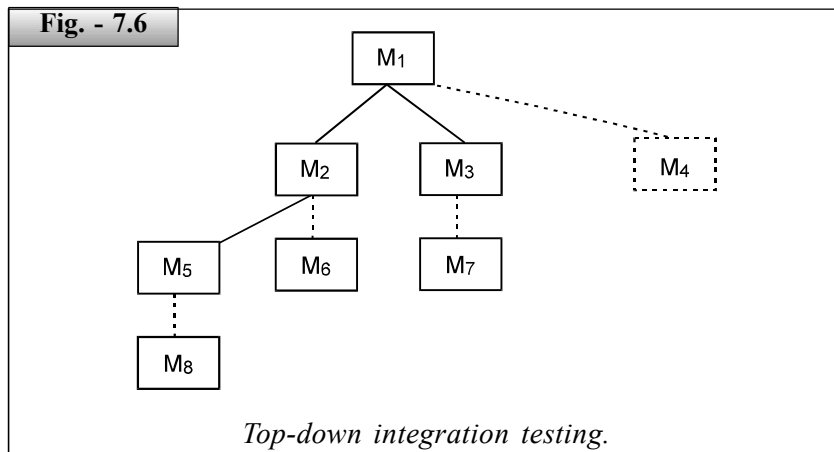


According to the figure, in the test sequence 1, test T_1 , T_2 and T_3 are assigned to X, Y modules. After the conformation that both X and Y modules are of error free, then the module Z is integrated with X and Y module (Test sequence 2). Then tests T_1 , T_2 and T_3 are repeated if problem is arised in these tests (T_1 , T_2 and T_3), then they interact with the new module. The main aim is to localize the arrived problem. Hence, the tests are simplified

and finally new modules are introduced with the commencement of new test cases also. (Finally module P is integrated and it is tested by using the existing tests T_1, T_2, T_3, T_4 and T_5 which is newly introduced). (In the above figure-7.5 it is represented in the test sequence 3).

2. Top down integration :

It is an incremental approach to establish a program structure. In top-down integration testing modules are integrated by moving downward through the control (driver) which begins with the main program.



There are 2 techniques to pass the control in top-down integration testing.

- i) Depth First Integration (DFI)
- ii) Breadth First Integration (BFI)

Depth first integration :

In this type, all the components are integrated vertically downwards from the top-to-bottom. The flow of control, here maintains the top down approach to pass itself. e.g. in depth first integration M_1, M_2 and M_5 are integrated first. Next M_8 or M_6 would be integrated. Then the central and right hand control paths are built up.

Breadth first integration :

In this technique, the flow of control maintains a horizontal approach. Here, all the components are directly subordinated at each level by moving across the structure horizontally. e.g. in the above figure, the components M_2, M_3, M_4 should be integrated first. The next control level M_5, M_6 and so on follows up.

The integration process can be carried out in five steps.

1. The main control module is used as a “test driver” and “the stubs” are substituted directly for all components those are subordinate to the main control module.
2. Depending upon the types of integration technique i.e. *DFI* or *BFI*, the subordinate stubs are replaced once at a time with the actual component.
3. Tests are conducted in such a way that each component would be integrated.
4. As soon as the test is completed, the old stub is removed and the new stub is replaced with the real components.

5. In case of *regression testing*, (is discussed in this chapter) the integration process is conducted to ensure that no new errors are introduced in the system, if the system has been already tested and error free.

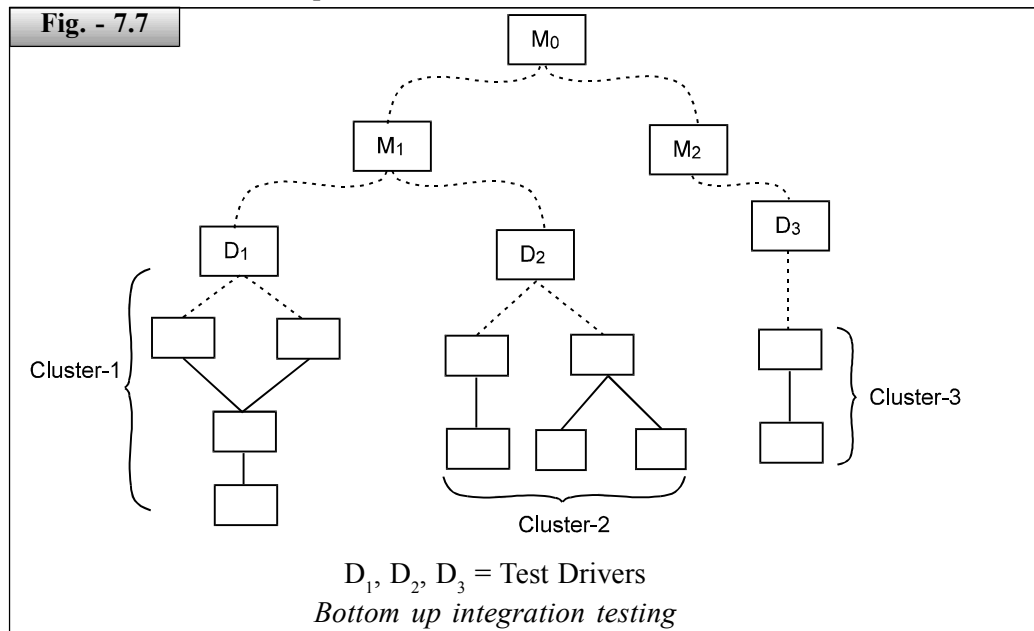
3. Bottom-up integration

It implies that the construction and testing with the components are done at the lowest level in the program structure.

Steps of bottom up integration testing :

A bottom up integration strategy can be implemented with the following steps.

1. Low level components are combined into clusters.
2. The driver of the test case is written in order to co-ordinate the input with the output.
3. The lower level components or the “clusters” are tested.



4. The drivers are removed and clusters are recombined by moving upward (vertical direction) in the program structure.

e.g. in fig. 7.7, the components are combined to form clusters 1, 2 and 3. Each of the cluster is tested by using the test driver. It is shown through the dashed block. The components in the cluster 1 and 2 are subordinates to M₁. The test drivers D₁ and D₂ are removed and the clusters are interfaced directly to M₁. Similarly, the test driver D₃ for cluster 3 is removed prior to the integration with module M₂. Both M₁ and M₂ will ultimately be integrated with the components M₀ and so on.

As the integration control moves in upward direction (vertically), there is no need to separate the test driver.

In case of the software having top-down structure, the no. of test drivers are reduced in case of testing. In this regard, the integrated clusters are simplified as there is less number of test drivers used for test cases.

4) Regression testing :

It is an activity of testing that helps to ensure the change or unchanged state of a software product after testing.

Regression testing prevents the modification to be introduced in the system.

It emphasises the testing, i.e. how to restrict the system from changing the behaviour, and commencement of additional errors into it.

Regression test contains 3 types of test cases, are as follows.

- Additional test that focus on the software functions.
- A representative sample of test that will exercise all software functions.
- Testing of the software components that have been already modified.

5) Smoke testing :

Smoke testing is an integration testing approach to test the shrink-wrapped software product.

Smoke testing is characterized as rolling integration approach as the software is rebuilt with new components and tested. Smoke testing is designed for critical project or for embedded software products.

Activities in smoke testing :

1. Software components are translated into specific codes. The codes are integrated to form a “build”. A build includes all data files, libraries, reusable modules and engineered components those are required to implement in case of product design.
2. A series of test cases are designed to explore the errors that would keep the “build” to perform it’s function properly.
3. The “build” is integrated with other builds and the entire product is undergone for smoke test regularly.

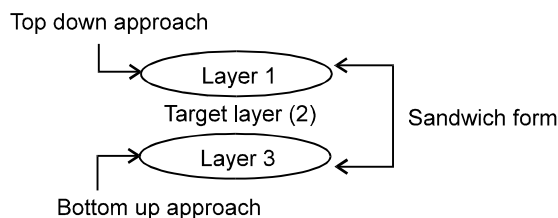
Benefits of smoke testing :

Smoke testing provides a no. of benefits when it is applied on a complex or embedded type of software product as follows.

1. The risk of integration is minimized.
2. It produces qualitative end product.
3. Process of error detection, error diagnosis and correction of errors are simple.
4. It enhances the access mechanism in an efficient manner.

6) Sandwich integration :

It is the combination of both top-down and bottom-up approach. In this approach, the whole system is divided into 3 layers just like a sandwich : the targeted component is kept in the middle layer and one layer is above the target layer and another layer is below the target layer.



In the upper layer, the top-down approach is used whereas in the lowest layer i.e. layer 3, bottom-up approach is used.

Testing process is encountered in the middle layer i.e. layer 2. It is selected on the basis of component structure and characteristics of the system.

It also combines the merits of both top-down and bottom-up approach. It also requires test driver. Stubs are also used to function the test driver.

7.7.3 System testing

During testing the whole system is classified into a number of subsystems. There are 3 kinds of system testing.

- Alpha testing
- Beta testing
- Acceptance testing.

a. Alpha testing :

Alpha testing refers to the testing of the system which is carried out by the test team within the software development organisation.

The alpha test is conducted at the developer site by the customers under the proper guidance of the developer team. In this test, users test the software on the development platform and indicates the error for correction.

Limitations of alpha testing :

1. Alpha test is conducted by few users of the development team.
2. It has limited ability to explore the errors and correct them.
3. Alpha test is conducted in a control environment.

Advantages of alpha testing :

1. Alpha testing is reliable for real time software development.
2. The alpha tested software product is ready for transition to the customer site for implementation and development.

Clearly, alpha test is concerned with the testing of system with raw data (old data) at the software developer organisation site.

b. Beta testing

- Beta testing is the system testing performed by a selected group of friendly customers.
- If the system is complex, the software is not taken for implementation directly. It is installed and all users are asked to use the software under testing mode. This is called the beta test.
- Beta tests are conducted at customer site in an environment where the software is exposed to a number of users.
- Beta test is a real life software experience without actual implementation.
- In this test, the end users record their observations, mistakes, errors and so on and report them periodically.
- In a beta test, the user may suggest a modification, a major change or deviation.
- The development has to examine the proposed change and put it into the change management system, to develop a revised and better software.

In a nutshell, beta testing is concerned with the testing of system with live data (current data) at the user's or customers organisation site.

c. Acceptance testing :

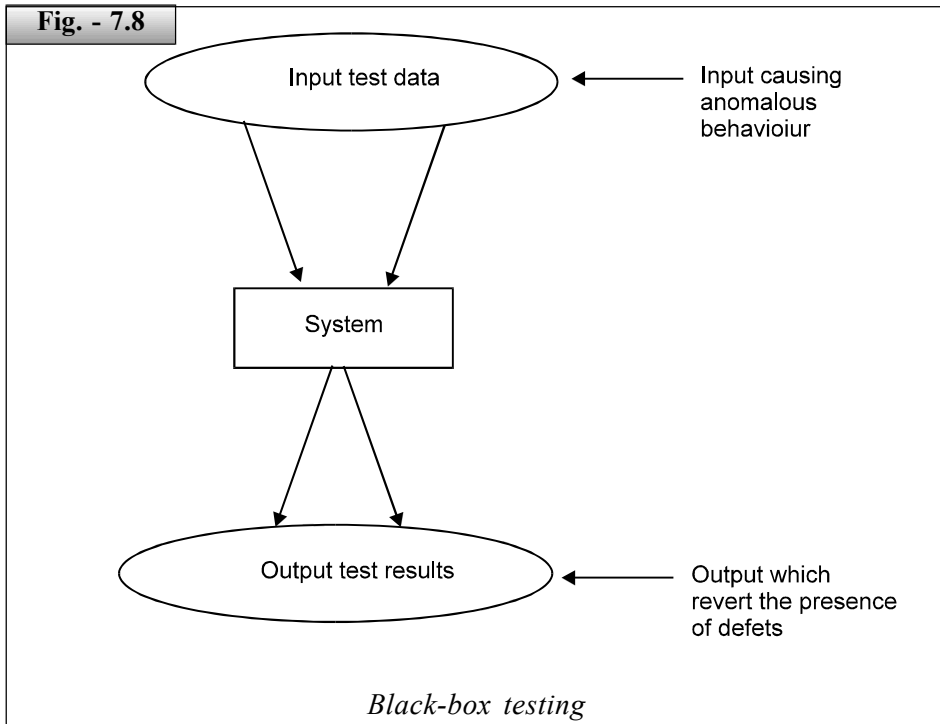
- Acceptance testing is concerned with the software development for specific users or customers.
- It is the system testing performed by the customer to determine whether to accept or to reject the delivery of the system.
- When customer software is built for a customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.
- It is conducted by the end user rather than software engineers. An acceptance test can range from an informal 'test drive' to a planned and systematically executed series of tests.
- Acceptance testing can be conducted over a period of weeks, or months, thereby uncovering cumulative errors that might degrade the system over time.

7.8 BLACK BOX (FUNCTIONAL) TESTING

- In functional testing, the structure of the program is not considered.
 - Test cases are decided on the basis of the requirements or specification of the program.
 - Functional testing refers to testing, which involves only observations of the output for certain input value and there is no attempt to analyse the code, which produces the output. The internal structure of the program is ignored for which it is also referred as black box testing.
 - Black box testing, also called behavioral testing, focuses on the functional requirements of the software.
 - Black box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements of the program.
 - Other names for black box testing includes : *specifications testing, behavioural testing, data-driven testing, functional testing* and *input/output driven testing*.
 - In black box testing, the tester only knows the inputs that can be given to the system and what output the system should given.
 - Black box testing is not an alternative to *white box techniques*; rather it is a complementary approach that is likely to uncover a different class of errors than white box methods.
 - Black box testing is classified into two categories :-
- i) **Positive functional testing :** This testing entails exercising the application's functions with valid input and verifying that the outputs are correct.
Example : Continuing with the word processing example, a positive test for the printing function might be to print a document containing both text and graphics to printer that is on-line, filled with paper and for which the correct drivers are installed.
 - ii) **Negative functional testing :** This testing involves exercising application functionality using a combination of invalid inputs, unexpected operating conditions and other "out-of-bounds scenarios".

Example-7.1

A negative test for the printing function might be to disconnect the printer from the computer while a document is printing.

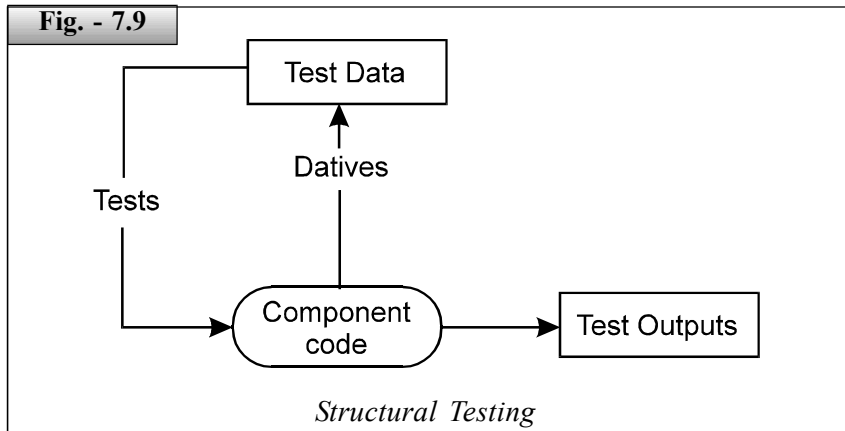


- **Merits :**

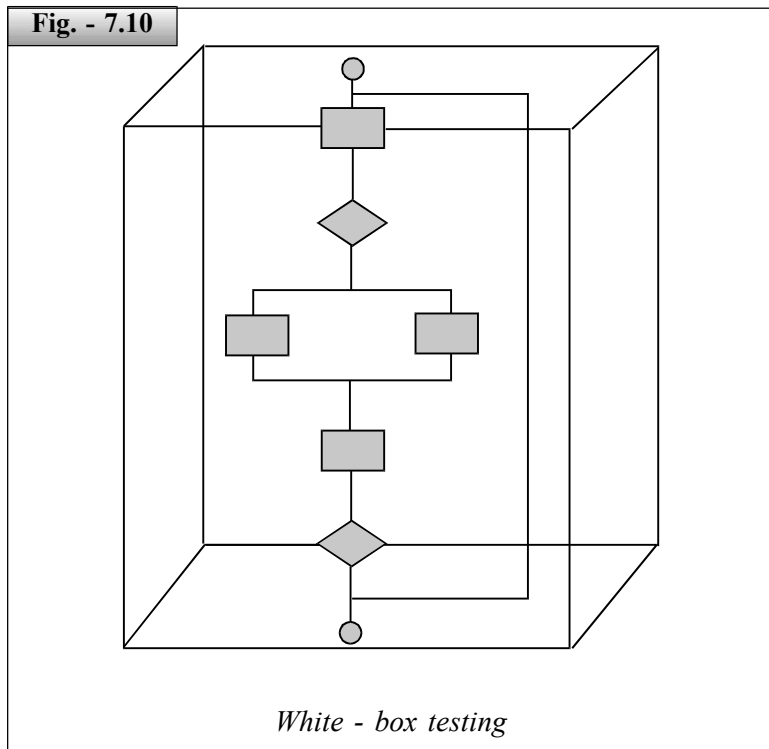
- ☛ The test is unbiased because the designer and the tester are independent of each other.
- ☛ The tester does not need knowledge of any specific programming languages.
- ☛ The test is done from the point of view of the user, not the designer.

7.9 WHITE BOX TESTING / STRUCTURAL TESTING

- A complementary approach to functional or black box testing is called structural or white box testing.
- In this approach, test group must have complete knowledge about the internal structure of the software and their implementation.
- Structural testing is usually applied to relatively small program units such as subroutines or the operations associated with an object.
- In this test, the tester analyse the code and use knowledge about the structure of a component to derive test data.

Fig. - 7.9

- White box test is also known by other names such as *Glass-box testing*, *structural testing*, *clear-box testing*, *open-box testing*, *Logic driven testing* and *path-oriented testing*.
- In white-box testing, test cases are selected on the basis of examination of the codes rather than the specifications.
- White box testing is illustrated in the figure 7.10.

Fig. - 7.10

- Here is a Java code for the voting checker program.

```
Public void actionPerformed (ActionEvent Event) {  
    int age ;  
    age = integer.parseInt (textField.getText());  
    if (age > = 18) {  
        result.setText ("you can vote");  
    }  
    else {  
        result . setText ("you cannot vote");  
    }  
}
```

In this program, there are two paths (because **if** has two branches) and therefore two sets of data will serve to ensure that all statements are executed at some time during the testing.

Test Number	Data	Expected Outcome
1	12	Cannot vote
2	21	Can vote

If we are cautious, we realise that errors in programming are often made within the condition of **if** and **while** statements. So we add a further two tests to ensure that the **if** statement is working correctly :

Test Number	Data	Expected Outcome
3	17	Cannot vote
4	18	Can vote

Thus we need four sets of data to test this program in a white box fashion. This happens to be the same data that we devised for black box testing. Suppose, for example, the program used an array, named *table*, with one element for each age specifying whether someone of that age can vote. Then the program is simply the following statement to look up eligibility :

```
result.setText(table[age]);
```

and the white box testing data is different.

Example - 7.2

A program's function is to find the largest of three numbers. Devise white box test data for this section of program. The code is :

```
int a, b, c;  
int largest;  
if (a >= b) {  
    if (a >= c) {  
        largest = a;  
    }  
    else {
```



```

        largest = c;
    }
}
else {
    if (b >= c) {
        largest = b;
    }
    else {
        largest = c;
    }
}
}

```

Example - 7.3 :

In a program to play the game of chess, the player specifies the destination for a move as a pair of “integer indices, the row and the column number. The program checks that the destination square is valid that is not outside the chess board.

Devise white box test data to check that this part of program is working correctly.

Solution : The code is :

```

if ((row > 8) || (row < 1)) {
    JOptionPane.showMessageDialog(null, "error");
}
if ((col > 8) || (col < 1)) {
    JOptionPane.showMessageDialog(null, "error");
}

```

- *White box testing is carried out to test whether :*
 - ☛ all paths in a process are correctly operational.
 - ☛ all logical decisions are executed with true and false conditions.
 - ☛ all loops are executed with their limit values tested and
 - ☛ to ascertain whether input data structure specifications are tested and then used for other processing.
- **Merits :**
 - ☛ Forces test developer to reason carefully about implementation.
 - ☛ Approximates the partitioning done by execution equivalence.
 - ☛ Reveals error in hidden code.

7.10 STATIC TESTING STRATEGIES :

- In order to show that certain properties are true regardless of the execution path of the program, a systematic examination of the program structure is done, known as static testing.
- The actual behaviour of a model, based on the program’s semantic feature and structure are represented by *static testing*.

7.11 FORMAL TECHNICAL REVIEWS

- Review is defined as : A meeting at which the software element is presented to project personnel, managers, users, customers or other interested parties for comment or approval.
- The discovery of any errors in analysis, design and coding, testing and implementation phase of a software development cycle is the main objective of software review.
- Uniform and manageable manner of application of procedures are also checked by the review.

Reviews objectives are :

- ☛ Conformation of specifications by the software element is ensured.
- ☛ Development of the software element with plans, standards and guidelines applicable for the project.
- ☛ Implementation and affect of changes to software elements, on the specified system areas, is one of the important objectives of system review.

Types of reviews :

- Reviews are generally of two types, such as;
- *Informal technical review* : It includes informal meeting and informal desk checking.
- *Formal technical review* : It is a formal software quality assurance activity through various approaches such as structural walk throughs, inspection etc.
- In this techniques, scrutinization of the product is done for completeness, correctness, consistency, technical feasibility, efficiency and adherence of establishing standards and guidelines by the client organisation.

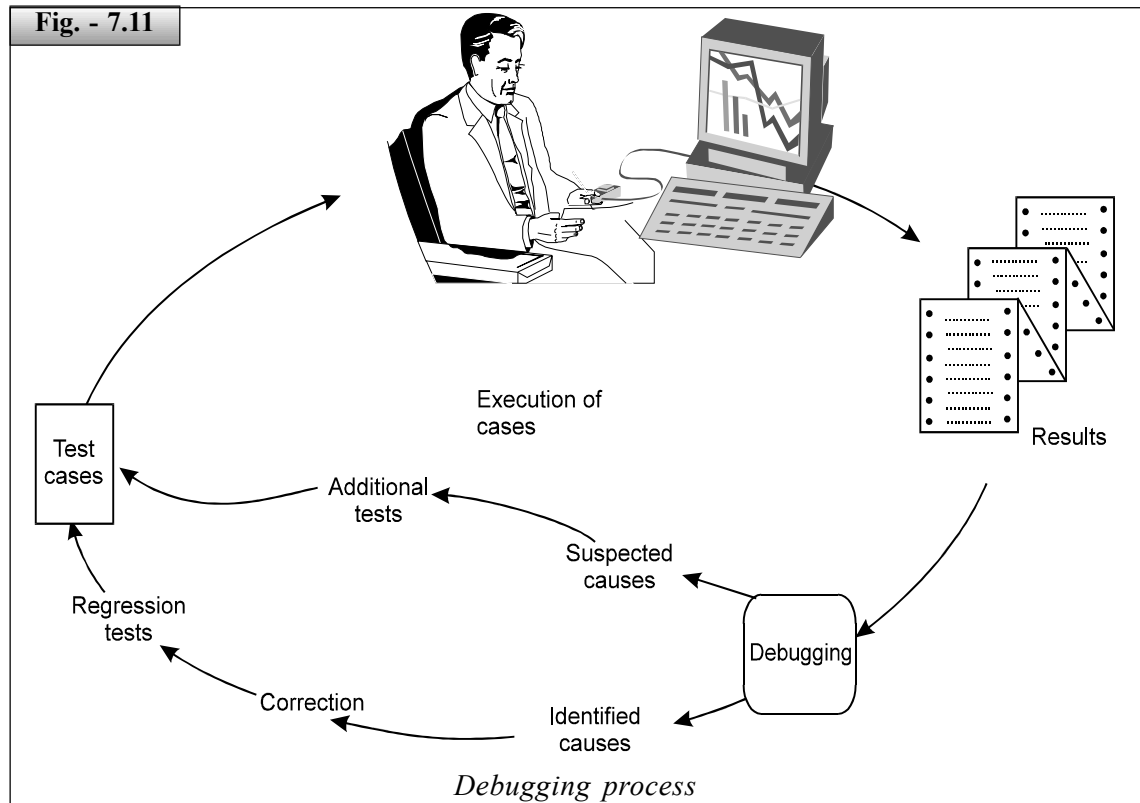
7.12 DEBUGGING

- Identification, point-out and correction of bugs from a program is called as *debugging*.
- During compilation phase, the source file is accessed and if errors are found, then that file is edited and the corrections are pasted in the file.
- After detection of the errors and inclusion of corrections in the source file, recompilation of the file is done. This error detection and removal is called debugging.
- This process of compilation, debugging and correction pasting in the source file continues until all syntactical errors are removed completely.
- There are various categories for debugging are :-
 - ☛ Brute force debugging.
 - ☛ Backtracking
 - ☛ Cause elimination
 - ☛ Program slicing
 - ☛ Fault - tree analysis.
- **Brute force debugging** : The programmer appends the print or write statement which, when executed, display the value of a variable.

- **Backtracking** : In this technique, the programmer back tracks from the place or statement which gives the error symptoms for the first time.
- **Cause elimination** : It is manifested by induction or deduction and introduces the concept of *binary partitioning*.
- **Program slicing** : This technique, is similar to backtracking. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.
- **Fault-tree analysis** : The fault-tree are used to support fault correction or tolerance, depending on the design strategy.

7.12.1 Debugging process :

- Debugging always occurs as a consequence of testing.
- As shown in the figure below, debugging process begins with the execution of a test case.



Errors :

- Error is a discrepancy between actual value of output given by the software and the specified correct value of the output for that given input.
- Error is also used to refer to the wrong decision in a given case, as compared to what is expected to be the right one.

- It also refers human activities that results in software containing a defect or fault.
- Error can be classified into two categories, such as :
 - ☛ **Syntax error** : A syntax error is a program statement that violates one or more rules of the language in which it is written.
 - ☛ **Logic error** : A logical error deals with incorrect data fields, out of range terms and invalid combinations.

7.13 SPECIAL SYSTEM TESTS

Type of test	Description
Peak load test	Determines whether the system will handle the volume of activities that occur when the system is at the peak of its processing demand. e.g. all terminals are active at the same time.
Storage testing	Determines the capacity of the system to store transaction data on a disk or in other files. e.g. verify documentation statements that the system will store 10,000 records of 383 bytes length on a single flexible disk.
Performance time testing	Determines the length of time used by the system to process transaction data. e.g. response time for inquiry when system is fully loaded with operating data.
Recovery testing	Determines ability of user to recover data or restart system after failure. e.g. load backup copy of data and resume processing without data or integrity loss.
Procedure testing	Determines clarity of documentation on operation and use of system by having users do exactly what manuals request. e.g. powering down system at end of week or responding to paper out light on printer.
Human factors testing	Determines how users will use the system, when the system is processing data or preparing reports. e.g. activities of user, when there is not an immediate response to an inquiry.

SUMMARY

- Software testing is accountable for a highest percentage of technical effort in a software development process. It's objective is to uncover the errors, which is accomplished through a set of steps like unit, integration, validation and system tests are planned and executed. Unit and integration tests focus on functional verification of a component and incorporation of components into a software architecture. Validation testing demonstrates the traceability to the product requirement, where as system testing validates the software product once it is implemented on the larger system.
- Software testing is also entertains a system based on its functional and structural behaviour i.e. Black box and white box testing.
- Unlike testing, Debugging must be viewed to find out the cause of an error through Brute force technique, Back tracking, fault tree analysis etc.
- More sophisticated approaches of testing are required for higher-quality software.

EXERCISES

1. What is the purpose of software testing?
2. When is the role of software testing start in software life cycle? When can planning for software testing start ?
3. Define the following terms:
(i) fault (ii) failure (iii) bug (iv) mistake
4. Discuss the principles of software testing.
5. What is the difference between
(i) Alpha testing & beta testing
(ii) Development & regression testing
(iii) Functional & structural testing.
6. Discuss the limitations of testing. Why do we say that complete testing is impossible?
7. Will exhaustive testing (even if possible for very small programs) guarantee that the program is 100% correct?
8. 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.
9. What is black-box testing ? List its advantages and disadvantages.
10. Outline the differences between black-box and glass-box testing.
11. White-box testing is complementary to Black-box testing, not alternative. Why ? Give an example to prove this statement.
12. Explain why validation is more difficult than verification.
13. Where does software testing fit in Software Quality Assurance (SQA)?



Software Certification

8.1 INTRODUCTION

We have already come across a no. of different phases, policies, methods, Techniques during the entire development life cycle of a software product which is exclusively meant for the stakeholders (users).

But the question arises, how can user rely on a software product ?

It is only possible to answer it through “*software certification*”.

Software certification is a measure on the highlighted attributes of a software product and to ensure that the product can be used by the stakeholders.

There are many important attributes / qualities of software products, which may also be applicable both to the product and to the process used to produce the product.

The quality attributes are

- ☛ Correctness
- ☛ Reliability
- ☛ User friendliness
- ☛ Verifiability
- ☛ Reusability
- ☛ maintainability
- ☛ Efficiency
- ☛ Portability etc.

Software certification is to qualify the product on its quality, which has an important role in the field of software engineering addressed by several standardization bodies such as *ISO, ANSI, IEEE etc.*

Here, the purpose of certification can be achieved through some procedures, techniques, tools applied by the software professionals to ensure that the software product meets or exceeds pre-specified standards w.r.t. the industrial or commercial acceptable level of excellence.

Certification can also be viewed as a benchmark set by some recognised professionals and the software development organisations have to go through it.

Let's discuss two important preliminaries through which the quality of a software product can be measured.

- *Verification*
- *Validation*

8.2 VERIFICATION AND VALIDATION

Verification and Validation is the name given to the checking and analysis process, ensuring that, the software conforms to its specification and meets the need of the customers who are paying for that software.

- Verification and validation is a whole life cycle process. It starts with requirement analysis, continues through design and coding process ends at testing.
- These two are not the same but still they create confusion.

The difference between them is expressed by Boehm.

☛ **Validation :** *Are you building the right product ?*

☛ **Verification :** *Are you building the product right ?*

Verification : Verification is the process of determining whether the output of one phase of software development confirms to that of its previous phase.

or

Verification involves checking of functional and non-functional requirements that the software confirms to its specification.

Validation : Validation is the process of determining whether a fully developed system confirms to its requirement specification.

or

Validation is an analysis process that is done after checking conformance of the system to its specification.

Thus, the goal of verification and validation process is to establish confidence in customer that the software system is "fit to them".

8.3 SOFTWARE QUALITY ASSURANCE

The aim of software quality assurance (SQA) process is to develop high quality software product. Software quality assurance is a set of activities designed to evaluate the process by which software is developed and / or maintained. The purpose of quality assurance group is to provide assurance that the procedures, tools and techniques used during product development and modifications are of desired level. The software quality assurance includes various processes such as :

- ☛ defines the requirements for software-controlled system failure detection, isolation and recovery.
- ☛ reviews the software development processes and products for software error prevention and / or control change to reduced functionality states; and
- ☛ defines the process for measuring and analysing defects as well as reliability and maintainability factors.

8.3.1 SQA objectives

- The SQA process is used to meet the following objectives :
 - ☛ Quality Management Approach
 - ☛ Measuring and reporting mechanisms
 - ☛ Effective software engineering technology.
 - ☛ A procedure to assure compliance with software development standards where applicable.
 - ☛ A multi-testing strategy is drawn.
 - ☛ Formal technical reviews that are applied throughout the software process.

8.3.2 SQA plan

- The SQA plan provides a path way for software quality assurance.
- The plan serves as a template for SQA activities that are instituted for each software project.
- The outline structure for a quality plan includes.
 - *Product introduction* : A description of the product, its intended market and the quality expectation of the product.
 - *Product plans* : The critical release dates and responsibilities for the product along with plans for distribution and product servicing.
 - *Process description* : The development and service processes that should be used for product development and management.
 - *Quality goals* : The quality goals and plans for the product including an identification and justification of critical product quality attributes.
 - *Risks and risk management* : The key risks that might affect product quality and the action to address their risks.

Preparation of a software quality assurance plan for each software project is a primary responsibility of the SQA group which includes topics on various fields, such as :

- Purpose - scope of plan ;
- List of references to other documents;
- Management, including organisation, tasks and responsibilities;
- Documents to be produced;
- Reviews and audits;
- Testing;
- Tools, technologies and methodologies for problems reporting and corrective action.
- Records collection, maintenance and retention;
- Training and risk management.

8.4 SOFTWARE QUALITY

Software quality is defined as : *conformance to explicit stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.*

Software quality is used for conformation of requirements and development criteria of a proposed software.

8.4.1 Classification of software quality

The producers of the software want the software products to be verifiable, maintainable, portable and extensible and also reliable, efficient and easy to use.

For this purpose, there are many desirable software qualities; as follows :

i) External versus internal qualities :

The external qualities are visible to users of the system which are mostly used by them.

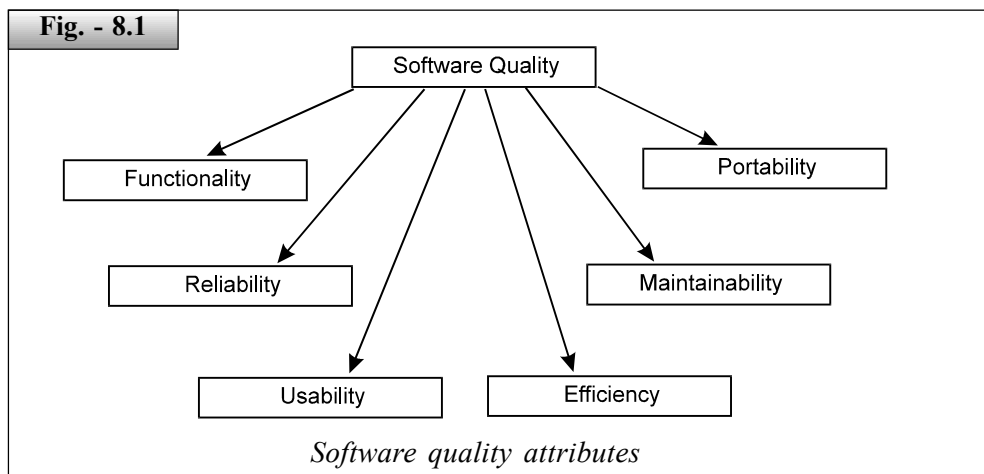
Internal qualities are those that concern the developers of the system with the structure of the software, that help developers to achieve external qualities.

ii) Product and process qualities :

We use process to produce the software product. If the process is carefully planned with data before design and development of the system, products reliability will increase. In general, software product not only comprises of the object code and user manual that are delivered to the customer but also the requirements, design, source code, test data etc.

8.4.2 Software quality attributes

Software quality comprises of six main attributes as shown in the figure 8.1



Functionality : The capability to provide functions which meet desired needs when the software is used.

Reliability : The capability to maintain a specified level of performance.

Usability : The capability to be understood, learned and used.

Efficiency : The capability to provide appropriate performance relative to the amount of resources used.

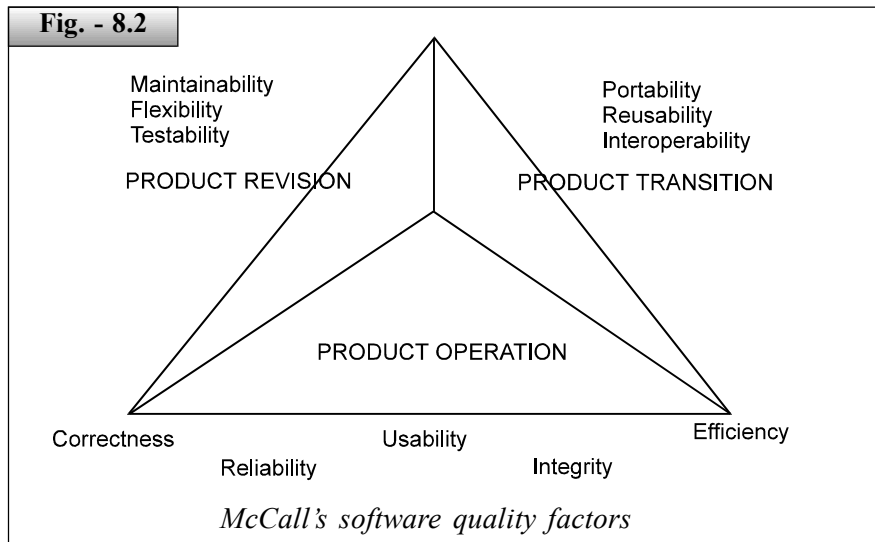
Maintainability : The capability to be modified for purposes of making corrections, improvements and adoption.

Portability : The capability to be adopted for different specified environments without applying actions.

8.4.3 McCall's quality factors

McCall, Richards and Walters propose a useful categorization of factors that affect software quality.

These software quality factors are shown in the figure which focus on three important aspects of a software product : its operational characteristics, its ability to undergo change, and its adaptability to new environments.



McCalls divides quality factors into three groups :-

- Product operation qualities ;
- Product revision qualities ;
- Product transition qualities.

8.4.3.1 Product operation quality factors

Correctness : The extent to which a program satisfies its specifications and fulfills the user's objectives.

Reliability : The extent to which a program can be expected to perform its intended function with precision.

Efficiency : The amounts of computer resources required by the software.

Integrity : The extent to which access to software or data by unauthorized persons can be

controlled.

Usability: The effort required for learning, operating, preparing input and interpreting output.

8.4.3.2 Product revision factors

Maintainability : The effort required to locate and fix an error in an operational program.

Testability : The effort required to test a program to ensure it performs its intended function.

Flexibility : The efforts required to modify an operational program.

8.4.3.3 Product transition quality factors

Portability : The effort required transferring a program from one hardware configuration and software System to another.

Reusability : The extent to which a program can be used in other applications.

Inter-operability : The efforts required to couple one system to another.

8.4.4 Criteria for software quality

<i>Quality factor</i>	<i>Criteria</i>
Correctness	: Traceability, consistency, correctness
Reliably	: Error tolerance, accuracy, simplicity
Efficiency	: Execution efficiency, storage efficiency
Integrity	: Access control, access audit.
Usability	: Operability, training, communicativeness, input / output volume, input / output area.
Maintainability	: consistency, simplicity, conciseness, modularity, self-Descriptiveness.
Testability	: Simplicity, modularity, instrumentation, self-descriptiveness.
Flexibility	: Modularity, generality, expendability.
Portability	: Modularity, machine-independence, software system independence.

8.4.5 Quality representation

The most important qualities of software product and process are as follows :

Correctness : A program is functionally correct if it behaves according to the specification of the function it should provide.

Reliability : software is reliable, if the user can depend on it.

Robustness : A program is robust if it behaves reasonably, even in unfavourable circumstances.

Performance : Performance is important because it affects the usability and scalability of the software system.

Verifiability : A software system is verifiable if its properties can be verified easily. It is usually an internal quality.

Repairability : A software system is repairable if it allows the correction of its defects with a limited amount of work.

Evaluability : software produces are modified over time to provide new functions or to change existing functions.

Understandability : Some software systems are easier to understand than others. Abstraction and modularity enhance a system's understandability.

Inter-operability : It refers to the ability of a system to coexist and cooperate with other systems.

Productivity : Productivity is a quality of the software production process; it measures the productivity and efficiency of the process.

Timeliness : Timeliness is a process-related quality that refers to the ability to deliver a product in time.

Visibility : A software development process is visible if all of its steps and its current status are documented clearly.

8.4.6 Importance of software quality

Software quality is very important because of its special characteristics, particularly intangibility and complexity.

1. **Increasing criticality of software** : The user are generally more anxious about the quality of the software, especially its reliability. This increases the safety critically.
2. **The intangibility of software** : The results of software tasks can be made tangible by demanding that the developers produce 'deliverables' that can be examined for quality.
3. **Accumulating errors during software development** : As the computer system development is made up of number of steps, where the output of one step is the input for another step; the next step, the errors gradually increases; as a result software qualities are used for the eradication of these errors.

For these reasons quality management is an essential of effective overall project management.

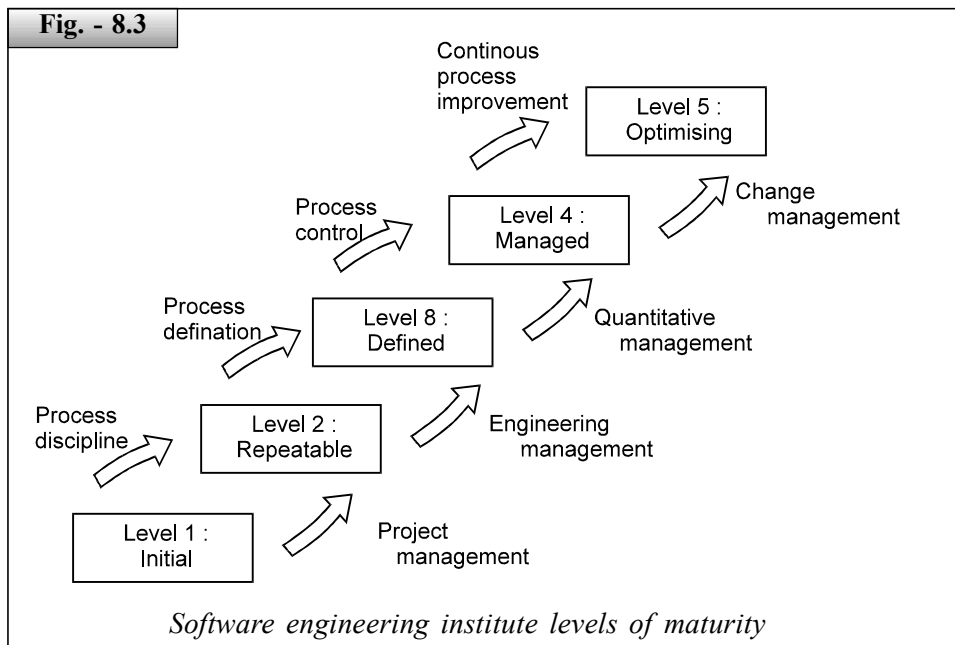
8.5 CAPABILITY MATURITY MODEL (SEI - CMM)

- SEI - CMM stands for *Software Engineering Institute Capability Maturity Model*.
- The CMM was developed by the *Software Engineering Institute (SEI) of the Carnegie Mellon University in the USA*.
- It is a model, which helps in judging the maturity of software process of an organisation.
- It also helps to identify the main process that will help in increasing maturity of these processes.
- It is used as a standard for accessing and improving software process :
- SEI-CMM can be used in two ways :
 - *Capability evaluation*
 - *Software process assessment*
- Capability evaluation provides a way to access the software process capability of an organisation. The results of capability evaluation is used to select a contractor.

- On the other hand, software process assessment is used by an organisation to improve its process quality. Thus it is for purely internal use.
- The sophistication and quality of software production are described by five levels of process being proposed by the SEI-CMM for any software industry.

These levels are designed as follows :

1. **Level 1 (Initial)** : The software process is adhoc and even chaotic in properties.
 - The organisation whose success depends on individual effort and processes, are not described by this level.
 - Organisations at this level benefits most by improving project management, quality assurance and change control.
2. **Level 2 (Repeatable)** : Here the company defines the basic management practices such as tracking cost.
 - Here the efforts done previously for success may repeat.
 - In this level, project commitments are realistic and based on passed experience with similar projects, cost and schedule are tracked, problems are resolved when they arise and software project standards are defined and followed.
 - The figure given below shows the different levels of maturity



3. **Level 3 (Defined)** : The organisation-wide software process includes management and engineering procedures.
 - These procedures are well defined, documented, standardized and integrated.
 - ISO 9000 aims at achieving this level.

4. Level 4 (Managed) : At this level, the focus is on the software matrices.

There are two types of matrices :

Product matrices measures the characteristics of the product being developed such as its size, reliability, time complexity, understandability etc.

Process matrices reflects the effectiveness of the product being used, such as the average defect correction time, productivity, the average of number of defects found per hour of inspection, and the average number of failures detected during testing per LOC.

- It is used to check if a project performs satisfactorily.

5. Level 5 (Optimized) : At this level, an organisation is committed to continuous process improvement.

- Process improvement is budgeted, planned and is an integral part of the organisation's process.
- Except for level 1, each maturity level is characterised by several Key Process Areas (KPA's) that indicate the areas an organisation should focus on, to improve its software process to the next level.

This is shown in the table as follows :

Table - 8.1 : Levels of SEI - CMM

MM level	Characteristics	Focus	Key Process-Areas
1. Initial	Chaotic, unrepeatable high risk of non-performance	Competent people	Not applicable
2. Repeatable	Methodical in umbrella activities and processes. Performance is repeated but not improved	Project Management	Configuration management, quality assurance, sub-contract management project tracking and oversight, project planning.
3. Defined	Improved performance in cost, schedule quality and risk management	Definition of processes.	Peer reviews, inter-group co-ordination, software product engineering, training programme, integrated software management and organization process definition.
4. Managed	Learns from project experience and continuous performance improvement in all key areas of project.	Product and process quality	Quality management process measurement and analysis.
5. Optimizing	All-round performance improvement through learning. High performance in all quality attribute and risk management through RMMM.	Continuous process improvement	Process change management, technology innovation, defect prevention.

8.6 INTERNATIONAL STANDARD ORGANISATION (ISO)

- The International Organisation for Standardisation (ISO) is worldwide federations of national standards bodies over some 100 countries.
- ISO is a non-governmental organisation established in 1947.
- The ISO 9000, standard specifies quality assurance elements in generic terms, which can be applied to any business, regards of the product or services being offered.
- The ISO 9000 standards view an organisation as a set of interrelated processes.
- In order to pass the criteria for ISO 9000 compliance, the processes must address the identified areas and document and practice them. When a process is documented, it is better understood, controlled and improved.
- ISO 9001 is a quality assurance standard that is specific to software engineering. It specifies 20 standards with which an organisation must comply for an effective implementation of the quality assurance system.
- ISO-9000 is a series of five related standards that are applicable to a wide variety of industrial activities, including design / development, production, installation and servicing.
- ISO-9000 series is a set of documents dealing with quality systems that can be used for quality assurance purpose.

8.6.1 Need of ISO certification for software industry

There are several reasons why a software industry must get an ISO certification : Some of the important reasons are as follows :

- It is sign of customer's confidence.
- It is a motivating factor to the business organisations.
- It makes the process more focused, effectively low cost and efficient.
- It helps in designing high quality software products.
- It facilitates the development of optimal process and total quality measurement.
- It emphasizes the need of proper documentation.

8.6.2 Steps for ISO 9000 certification

An organisation intending to obtain ISO 9000 certification must apply for a registration process which consists of the following stages.

1. **Application :** The organisation must apply to a registrar for registration, to get ISO 9000 certification.
2. **Pre-assesment :** During this stage, the registrar makes a rough assessment of the organisation.
3. **Document review and adequacy of audit :** During this stage, the registrar reviews the documents submitted by the organisation and makes suggestions for possible improvements.
4. **Compliance audit :** During this stage, the registrar checks the suggestions made by him/her during the review have been complied with by the organisation.

5. **Registration** : The registrar awards the ISO 9000 certificate after successful completion of all previous phases.
6. **Continued surveillance** : The registrar continuous to monitor the organisation, though periodically.

8.6.3 Benefits of ISO-9000 certification

Benefits from ISO-9000 certification to various organisation are as follows :

1. **Continuous improvement** : Each procedure and work instructions must be documented for continuous improvement.
2. **Eliminate variation** : Documented processes helps to eliminate variation within process, which results in improved efficiency and reduction in cost.
3. **Higher-real and perceived quality** : With the development of solid corrective and preventive measures, permanent and company-wide solutions to quality problems are found.
4. **Improved customer satisfaction** : Customer satisfaction and more importantly customer loyalty, grows as a company transforms from reactive to preventive organisation.
5. **Increased employee participation** : Employee participation increases due to reduction in problems in an organisation.
6. **Better product and services** : This results from continuous Improvement processes.
7. **Greater quality assurance** : It is everyone's responsibility to develops software qualities.
8. **Reduced cost** : ISO standard results in a reduced cost of the product.
9. **Improved profit levels** : It results as productivity improves and rework costs are reduced.

8.6.4 Uses of ISO

- ISO certification is used as a reference model for a contract between the external independent parties.
- It focuses mainly on the processes.
- ISO is now being used as the standard and basis of evaluation for international bidding.
- ISO 9000 specifies a set of guidelines for repeatable organisations (i.e. organisation where the production process is person independent)

ISO 9126

- ISO 9126 standard was published in 1991 to tackle the question of the definition of software quality.
- This 13 page document was designed as a foundation upon which further, more detailed, standards could be built.
- ISO 9126 identifies six software quality characteristics such as
Functionality : Covers the functions that a software product provides to satisfy user needs.
- The sub-characteristics of functionality are suitability, accuracy, Inter-operability, compliance and security.

Reliability : It relates to the capability of the software to maintain its level of performance.

- Its sub-characteristics are maturity, fault tolerance, recoverability.

Usability : It relates to the effort needed to use the software.

- Its sub-characteristics are understandability, learnability and operability.

Efficiency : It relates to the physical resources used when the software is executed.

- Time behaviours and resource behaviour are described as the sub-characteristics of efficiency.

Maintainability : It relates to the effort needed to make changes to the software.

- It is sub-characterised as analyzability, changeability, stability and testability.

Portability : It relates to the ability of the software to be transferred to a different environment.

- Its sub-characteristics are adaptability, installability, conformance, and replaceability.
- ISO 9126 provides guidelines for the use of the quality characteristics.
- Once the requirements of the software product have been established the following steps are suggested :

- Quality matrices selection
- Ratings level definition
- Assessment criteria definition.

8.6.5 Comparison between ISO 9000 certification and SEI-CMM

Comparison of some of the key characteristics of ISO 9000 certification and SEI-CMM model are as follows :

- The emphasis of the SEI-CMM is on continuous process improvement where as ISO 9000 addresses the minimum criteria for an acceptable quality system.
- The CMM is a five-level framework for measuring software engineering practices as they relate to process where as ISO 9000 defines a minimum level of generic attributes for a quality management programme.
- ISO 9000 certification can be quoted by an organisation in official document, in communication with external parties and in tender quotations where as CMM is purely for internal use.
- ISO 9000 Standards were basically designed to audit manufacturing / service organization where as CMM was developed specifically for software industry.
- The SEI-CMM focuses strictly on software, while ISO 9000 has a much wider scope : Hardware, software, processed materials and services.
- SEI-CMM aims for achieving Total Quality Management (TQM) where as ISO 9000 aims at level 3 (defined level) of SEI-CMM model.
- ISO 9000 requires that procedures for handling storage, packaging and delivery established and maintained, while replication, delivery and installation are not covered in the SEI-CMM.

8.6.6 Classification of failures

- Recoverable : System are recovered with or without operator intervention.
- Transient : Occur only with certain inputs.
- Unrecoverable : System cannot recover without operator intervention or in it the system may need to be restarted.
- Corrupting : Failure corrupts system data.

8.6.7 Limitation of ISO 9000 certification

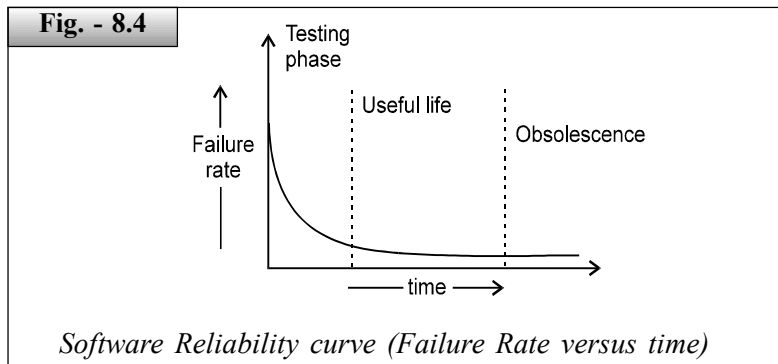
ISO 9000 certification has the following limitations :-

- ✚ ISO 9000 does not provide any guidelines for defining an appropriate process.
- ✚ ISO 9000 certification process is not fool proof and no international accrediting agency exists.
- ✚ ISO 9000 does not automatically leads to continuous process improvement.

8.7 RELIABILITY ISSUES

Software reliability

- Software reliability is defined as the probability of failures- free operation of a computer program in a specified environment for a specified time.
- Reliability of a software product can also be defined as the probability of the product working correctly over a given period of time.
- The expected curve for software is given in the figure 8.4.



Some contributing factors of software are given below :

- Change in environment.
- Change in infrastructure / technology.
- Major changes in requirements.
- Increase in complexity
- Extremely difficult to maintain.
- Deterioration in structure of the code.
- Slow execution speed.
- Poor graphical user interface.

8.7.1 Software reliability specification

Reliability is a complex concept, because the components in a system are interdependent, a failure in one component can be propagated throughout the systems and affects the operation of other components.

Considering system reliability, there are three categories.

- i. **Hardware reliability** : What is the probability of a Hardware component failing and how long does it take to repair that component ?
 - ii. **Software reliability** : How likely is it that a software component will produce an incorrect output ? It can continue operating correctly after an incorrect result has been produced.
 - iii. **Operator reliability** : How likely is it that the operator of a system will make an error ?
- The reliability of a system depends upon a number of factors, then the overall probability is the sum of all the probabilities.

$$P_S = P_A + P_R + \dots + P_N$$

where, P_S = Probability of system failure.

P_A = Probability of component A failure.

P_B = Probability of component B failure.

P_N = Probability of component N failure.

8.7.2 Reliability terminologies

- **System failure** : An event that occurs at some point of time when the system does not deliver a service as expected by its users.
- **System error** : Erroneous system behaviour where the behaviour of the system does not conform to its specification.
- **System fault** : An incorrect system state i.e. a system state that is unexpected by the designers of the system.
- **Human error or mistake** : Human behaviour that results in the introduction of faults into a system.

8.7.3 Reliability metrics

Reliability metrics are used to quantitatively express the reliability of a software product. Some reliability metrics which can be used to quantify the reliability of a software product are :

i) MTTF (Mean Time to Failure)

- The MTTF is the mean time for which a component is expected to be operational.
- MTTF is the average time between observed system failures.
- An MTTF of 500 means that one failure can be expected every 500 time units. The time units are totally dependent on the system and it can even be specified in the number of transactions, as is the case of database query systems.
- MTTF is relevant for systems with long transactions i.e. where system processing takes a long time.

- MTTF should be longer than transaction length. For example it is suitable for CAD systems where a designer will work on a design for several hours, word processor systems.

ii) MTTR (Mean Time to Repair) :

- MTTR measures the average time it takes to track the errors causing the failure and then to fix them.
- MTTR is the average time to replace a defective component.

iii) MTBF (Mean Time Between Failures) :

- The MTTF and MTTR metrics are combined to make MTBF.

$$\Rightarrow \boxed{\text{MTBF} = \text{MTTF} + \text{MTTR}}$$

- Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected to occur only after 300 hours.

iv) POFOD (Probability of Failure on Demand) :

- POFOD is the likelihood that the system will fail when a service request is made.
- A POFOD of 0.001 means that one out of a thousand service requests may result in failure.

v) ROCOF (Rate of Occurance of Failure) :

- ROCOF is the frequency of occurrence with which unexpected behaviour is likely to occur.
- A ROCOF of 2/100 means that two failures are likely to occur in each 100 operational time units. This metric is sometimes called as failure intensity.

vi) AVAIL (Availability)

- Availability is the probability that the system is available for use at a given time.
- Availability of 0.998 means that in every 1000 time units the system is likely to be available for 998 times.
- **Non-corrupting** : Failure does not corrupt system data.
- **Permanent** : Occur for all input values while invoking a function of the system.

8.7.4 Measurement of Reliability and Availability

Measurement of reliability :

- Most hardware related reliability models are predicted on failure due to wear rather than failure due to design defects.
- If we consider computer - based system, a simple measure of reliability is Mean-Time-between-Failure (MTBF) and it can be expressed on

$$\Rightarrow \boxed{\text{MTBF} = \text{MTTF} + \text{MTTR}}$$

where, MTTF = Mean time to failure
MTTR = Mean time to repair

Measurement of availability :

- In addition to reliability measure, we must develop a measure of availability.
- Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as :

$$\Rightarrow \text{Availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})} \times 100\%$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measurement is more sensitive to MTTR.

Example - 8.1 : For a software project, the following data is given below

MTTR = 10 days

MTTF = 35 days

Calculate MTBF for the above software project.

Solution : We know that,

$$\begin{aligned} \text{MTBF} &= \text{MTTF} + \text{MTTR} \\ &= 35 + 10 = 45 \text{ days} \end{aligned}$$

Example-8.2 : Calculate the availability of the software of
MTTR = 20 hours, MTTF = 10 hours.

Solution : The availability of software project is defined as

$$\begin{aligned} \text{Availability} &= \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\% \\ &= \frac{10}{10 + 20} \times 100\% \\ &= \frac{10}{30} \times 100\% \\ &= 33.3\% \end{aligned}$$

Example-8.3 :

Calculate the productivity of the software project if, the total line of code = 5000 SLOC and implementation days = 500 working days.

Solution : Productivity of software project

$$\begin{aligned} &= \frac{\text{total line of code}}{\text{Implementation days}} \\ &= \frac{5000}{500} = 10 \text{ SLOC / days} \end{aligned}$$

Example - 8.4 :

A software project has 5000 SLOC and it took 400 work days to implement. Some problem increase 100 SLOC and which took 20 work-days to implement this. Calculate the extendibility of the software project ?

Solution : productivity of the original system.

$$= \frac{5000}{400} = 12.5 \text{ SLOC/Staff day.}$$

$$\begin{aligned} \text{Productivity after maintenance : } & \frac{100}{20} \\ &= 5 \text{ SLOC / Staff day} \end{aligned}$$

$$\begin{aligned} \text{Thus, extendibility} &= \frac{5}{12.5} \times 100\% \\ &= 40\% \end{aligned}$$

8.7.5 Reliability growth modelling

- A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired.
- Thus, reliability growth model is used to determine when to stop testing to attain a given reliability model level.
- Several reliability growth models has been proposed as follows.

Jelinski-moranda model

- The Jelinski - Moranda model is the earliest and probably the best-known reliability model.
- It proposed a failure intensity function in the form of

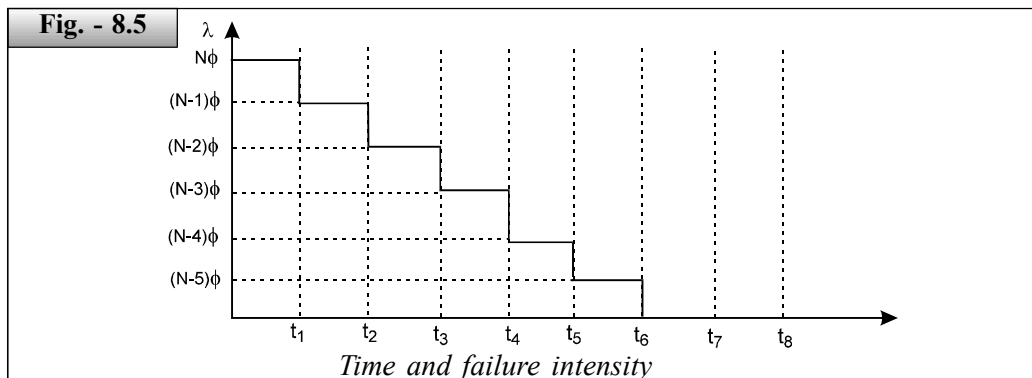
$$\lambda(t) = \phi(N - i + 1)$$

where, ϕ = constant for proportionality

N = Total numbers of errors present.

i = Number of errors found by time interval 'in'i.

- This model assumes that all failures has the same failure rate. So, every failure contributes equally to the overall reliability.
- The failure intensity is directly proportional to the number of remaining errors in a program.
- The relation between time and failure intensity is shown in figure.

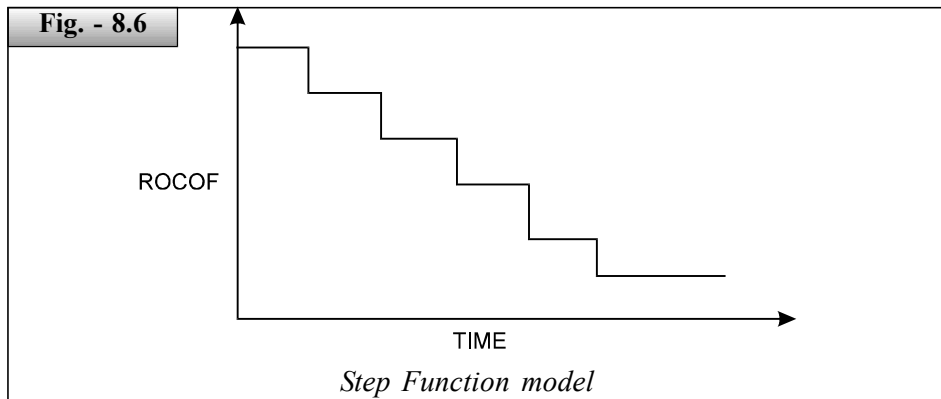


- The time interval $t_1, t_2, \dots, t_k$ may vary in duration depending upon the occurrence of a failure.

Between $(i - 1)^{\text{th}}$ and i^{th} failure, failure intensity function is $(N - i + 1)\phi$.

2. Step function model :

- The simplest reliability growth model is a step function model, where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired.
- A step function model is shown in the figure below



3. Little Wood and Verall's Model :

- This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors.
- It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases.

Example (of Jelinski - Moranda Model)

Example : 8.5

There are 50 errors estimated to be in a program. We have experienced 30 errors. use jelinski - Moranda model to calculate failure intensity with a given value of $\phi = 0.03$. What will be the failure intensity after experience of 40 errors ?

Solution : $N = 50$ Errors
 $i = 30$ failures
 $\phi = 0.03$

We know, $\lambda(t) = \phi (N - i + 1)$
 $= 0.03 (50 - 30 + 1)$
 $= 0.63$ failures / CPU hr.

After 40 failures,

$\lambda(t) = 0.03 (50 - 40 + 1)$
 $= 0.33$ failures / CPU hr.

Hence, there is continuous decrease in the failure intensity as the number of failure experienced increases.

8.8 PERSONAL SOFTWARE PROCESS

In the year of 1997 the PSP was proposed by David Humphrey.

PSP is a version of industrial software process (ISP).

PSP is suitable for individual use. PSP recognizes that the process for individual use is quite different from the process that for the group purpose.

PSP is a framework for the measurement of the developer's work performance. It also improves the rate of development process.

PSP is helpful to improve the developer's skill through proper project planning, project estimation, and process tracking i.e. PSP planning is time measurement.

- **Time measurement :**

PSP stresses that the developers should follow the time management policy. To avoid the mismanagement of time and to finish the development process in the specified time schedule, the actual time should be measured with the help of a stop clock.

The stop clock measures the actual time spends for design, coding and testing.

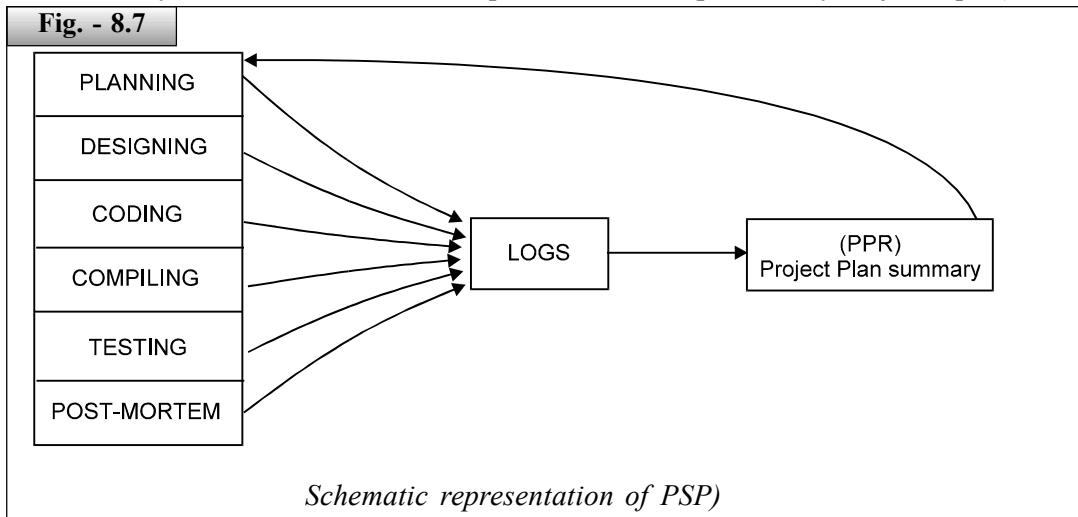
Actually the developers measure the time with the help of stop clock.

8.8.1 PSP planning

Planning is done by the individuals, for their projects. They plan about the Lines of Codes standards. (LOC) required to write the codes.

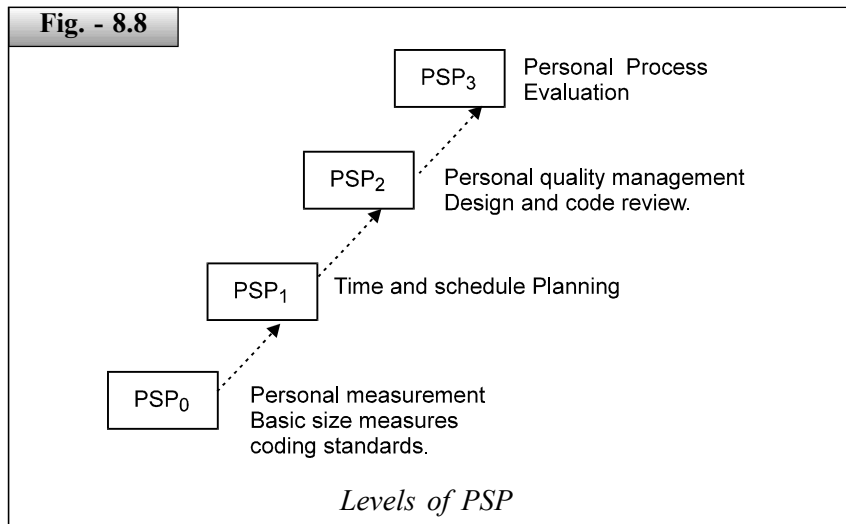
The developers use their productivity in minutes / LOC to workout the maximum, minimum and average development time.

They must record the estimated plan in the PPR. (preliminary Project report)



As mentioned above, during the course of development of different phases, the developer must record the log data by using time measurement.

During post-mortem, they can compare the log data with their project plan to achieve better planning in the future project with a view to enhance their process.



In the above figure the PSP_2 commences defect management through the use of checklists for code and design reviews.

The checklists are developed from gathering and analysing the defect data from the earlier projects.

8.9 SIX SIGMA

Six sigma is a disciplined, data driven approach to eliminate the defects in any process from manufacturing to transactions and from product to service.

It was introduced by the *General Electric* in 1995.

Six sigma is concerned with, “*how a process is performing qualitatively*” ?

To achieve six-sigma, there is a restriction i.e. a process must not produce more than 3-4 defects per million opportunities.

A six sigma effect is defined as any system behaviour that is not as per customer specification.

The total no. of six sigma opportunities is the total no. of chances for a defect. Process can be evaluated by using a six sigma calculator.

8.9.1 Objectives

Its objective is to implement a measurement based stresses that stresses on process improvement and variation reduction through the application of Six sigma improvement projects. For this purpose. Two sub-methodologies of the six sigma are used these are *DMAIC* (define, major, analyse, improve, control) and *DMADV* (define, major, analyse, design, verify).

The *DMAIC* process includes the activities like institution, measurement, analysis, improvement and central.

The *DMAIC* Process is an improvement system for existing Process following below specification and thus looking for increment improvement.

The Six Sigma *DMADV* Procures includes the activities like initialisation, measurement, analysis, design, and verification.

It is meant for an improvement of system under to develop new process or products at six sigma quality level. It can also be applicable when a current process requires more than just incremental improvement.

SUMMARY

- Software certification is a measure on the highlighted features of a software product w.r.t. a benchmark as set or framed by software Quality Assessment Organisation, like ISO, ANSI, IEEE etc., that identifies the degree of quantity of the product as well as the satisfaction of stakeholders.
- To incorporate both quality control and quality assurance in the different stages of software development process an activity called software quality management is carried out; through SQA (Software quality Assurance) tools. SQA helps a project team in assessing and improving the quality of software work product, based on its functionality, reliability, usability, maintainability etc.
- SEI-CMM helps in judging the maturity level of software development process. Software reviews are one of the most important quality control activities.
- In a nutshell, software certification is a quality assurance activity, for mapping the Managerial precepts and design disciplines onto the applicable managerial and technological space of software engineering.

EXERCISES

1. What is the significance of certification for software development process. Discuss in association to SDLC process.
2. State some of the Software Quality factors that are proposed by McCall?
3. Write a short note on Representative Qualities?
4. Describe the various classifications of Software Quality?
5. How do you define Reliability? Discuss various models for reliability allocation?
6. What do you mean by Software Quality Assurance?
7. What are the goals of Software Quality Assurance?
8. What are the objectives of Software Quality Assurance?
9. Write a short note on SQA Plan?
10. Why is it important for a software development organization to obtain ISO 9000 certification?

11. Discuss main requirements of ISO 9001 and compare it with SEI capability maturity model?
12. Discuss how the reliability changes over the lifetime of a software product and a hardware product?
13. Write short notes on: (i) Software Quality (ii) Reliability metrics (iii) Software reliability
14. Give Comparison between: (i) ISO and SEI-CMM (ii) Software and Hardware reliability.
15. Explain in detail the SEI Capability Maturity Model (SEI-CMM)? Also differentiate it with ISO?
16. Give the shortcomings of ISO 9000 certification?
17. Explain any two types of Reliability Growth Model?
18. What do you mean by Software reliability and Software availability? Also discuss how they are measured.
19. What do you mean by Software Quality?
20. Explain the stages, which an organization will, take in order to obtain ISO 9000 certification?
21. Answer the following questions:
 - (i) Can a program be correct and still not be reliable? Explain.
 - (ii) What is ISO 9000 certification?
 - (iii) What are the salient features of ISO 9001 requirements?
 - (iv) What is software reliability? Explain.



Software Maintenance

9.1 INTRODUCTION

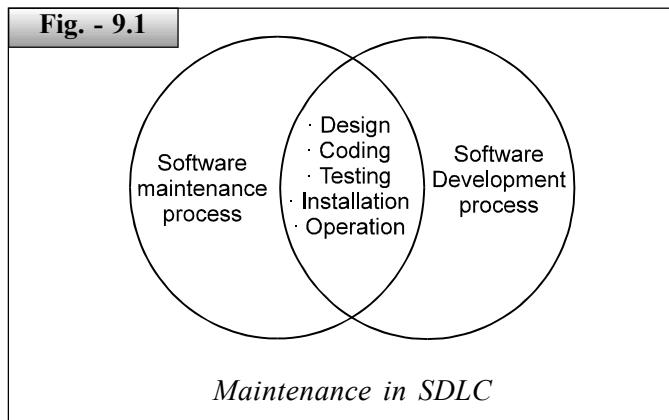
Recent years of software development, treat maintenance as a major activity in the industry.

Software maintenance describes the activities that follow the delivery of the initial complete version of the software product to the client. Any changes like, change in code, documentation, manuals or any other components of the product (software system) are categorised as maintenance.

Maintenance should be incorporated throughout the development and more time is spent on maintenance than any other phases of software life cycle.

Surveys and estimates made between 1988 and 1990 provide a clear picture that an average of 70% of a software project budget is utilised towards the maintenance activities over the life of a software product.

Software development and software maintenance, the two processes in software Engineering (SE), constitute a cycle for the entire life span of a software system. It can be shown in the following figure.



9.2 NEED FOR SOFTWARE MAINTENANCE

- The term '*software maintenance*' denotes any changes made to a software product after it has been delivered to the customer.

- Software maintenance can be defined as a set of activities undertaken on a software system, its release for operational use.
- Software maintenance is the activity associated with keeping operational computer system continuously in tune with the requirements of users and data processing operation. Software maintenance process is expensive and risky and is very challenging.

There is need of software maintenance due to following reasons.

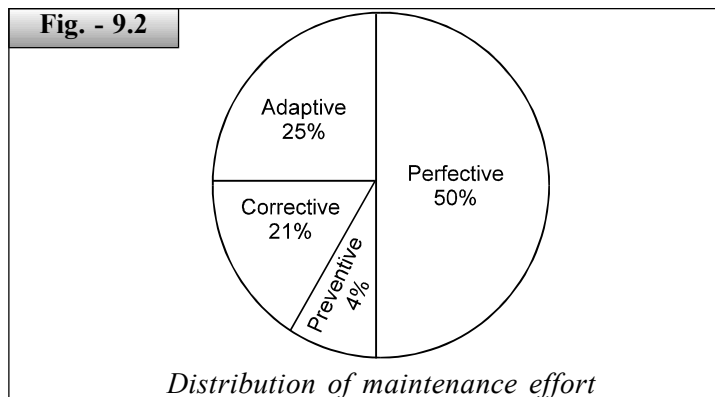
- ☛ Changes in user requirements with time.
- ☛ Program / system problems.
- ☛ Changing hardware / software environment.
- ☛ To improve system efficiency and through-put.
- ☛ To modify the components.
- ☛ To test the resulting product to verify the correctness of changes.
- ☛ To eliminate any unwanted side effects resulting from modification.
- ☛ To augment or fine-tune the software.
- ☛ To optimize the code to run faster.
- ☛ To review of standards and efficiency.
- ☛ To make the code easier to understand and work with.
- ☛ To eliminate any deviations from specifications.

Maintenance is concerned with the correction of errors, correction of requirements and design specification, migration of *legacy system (old system)* so as to make the enhancement in the productivity and the quality of the software.

9.3 CATEGORIES OF MAINTENANCE

Maintenance may be classified into the four categories as follows

Corrective	:	reactive modification to correct discovered problems.
Adaptive	:	modification to keep it usable in a changed or changing environment.
Perfective	:	improve performance or maintainability.
Preventive	:	modification to detect and correct latent faults.



- **Corrective Maintenance :**

Corrective maintenance means, repairing of processing or performance failures or making changes because of previously uncorrected problem.

Corrective maintenance refers to modifications initiated by defects in the system software. This type of maintenance is also called *bug fixing*. A defect can result from the following errors.

- (a) **Design errors :**

Design errors occurs when changes made to the software are incomplete, incorrect, wrongly communicated or the change request is misunderstood.

- (b) **Logic errors :**

Logic errors result from invalid test and conclusions, incorrect implementation of design, specifications, faulty logic flow or incomplete test data.

- (c) **Coding errors :**

Coding errors are caused by incorrect implementation of detailed design and incorrect use of the source code logic.

- **Adaptive maintenance :**

Adaptive maintenance means changing the program function. This is done to adapt to the external environmental change.

for e.g : the current system was designed so that it calculates taxes on profits after deducting the dividend on equity shares.

The government has issued orders now to include the dividend in the company profit for tax calculation. This function needs to be change to adapt to the new system.

- **Perfective maintenance :**

Perfective maintenance means enhancing the performance or modifying the programs to respond to the user's additional or changing needs. For examples earlier data was sent from stores to headquarters on magnetic media but after stores were electronically linked via leased lines, the software was enhanced to send data via leased lines.

As maintenance is very costly and very essential, efforts have been done to reduce its cost. One way to reduce the costs is through maintenance management and software modification audit. Software modification consists of programs rewriting system level upgradation.

- **Preventive maintenance :**

Preventive maintenance is the process by which we prevent our system from being obsolete. Preventive maintenance involves the concepts of reengineering and reverse engineering in which an old system with an old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

9.4 CHALLENGES IN MAINTENANCE

The most important problem during maintenance is that before correcting or modifying a program, the programmer must first understand it. Then, the programmer must understand the impact of the intended change. Some problems are discussed below.

- Often, the program is written by another person or group of persons working over the years in isolation from each other.
- Often the program is changed by person who did not understand it clearly, resulting in a deterioration of the program's original organisation.
- Program listings even those that are well organised are not structured to support reading for comprehension.
- There is a high staff turnover within Information technology industry. Due to this, many systems are maintained by persons who are not the original authors. These persons may not have adequate knowledge about the system.
- Some problems only become clear when a system is in use. Many users know what they want but lack the ability to express it in a form, understandable to programmers analysts. This is primarily due to information gap.
- Systems are not designed for change. If there is hardly any scope for change, maintenance will be very difficult. Therefore approach of development should be the production of maintainable software.

9.5 SOLUTION TO MAINTENANCE CHALLENGES

- A number of possible solutions to maintenance problem have been suggested. They include : budget and effort reallocation; complete replacement of existing systems; and enhancement of existing systems.

- **Budget and effort reallocation**

It is now-a-days suggested that more time and resources should be invested in the development specification and design of more maintainable systems rather than allocating resources to develop unmaintainable or difficult to maintain systems.

The use of more advanced requirement specification approaches, design techniques and tools, quality assurance procedures and standards such as ISO 9000, CMM and maintenance standards are aimed at addressing this issue.

- **Complete replacement of the system**

If maintaining an existing system costs as much as developing a new one, why not develop a new system from scratch. This point of view is understandable, but in practice it is not simple. The risk and costs associated with complete system replacement are very high. Corrective and preventive maintenance take place periodically at relatively small maintenance changes, while at the same time, supporting more ambitious and financially demanding projects may not be possible.

The creation of another system is no guarantee that it will function better than the old one.

On installation, errors may be found in functions which the old system performed correctly and such errors will need to be fixed as they are discovered.

- **Maintenance of existing system**

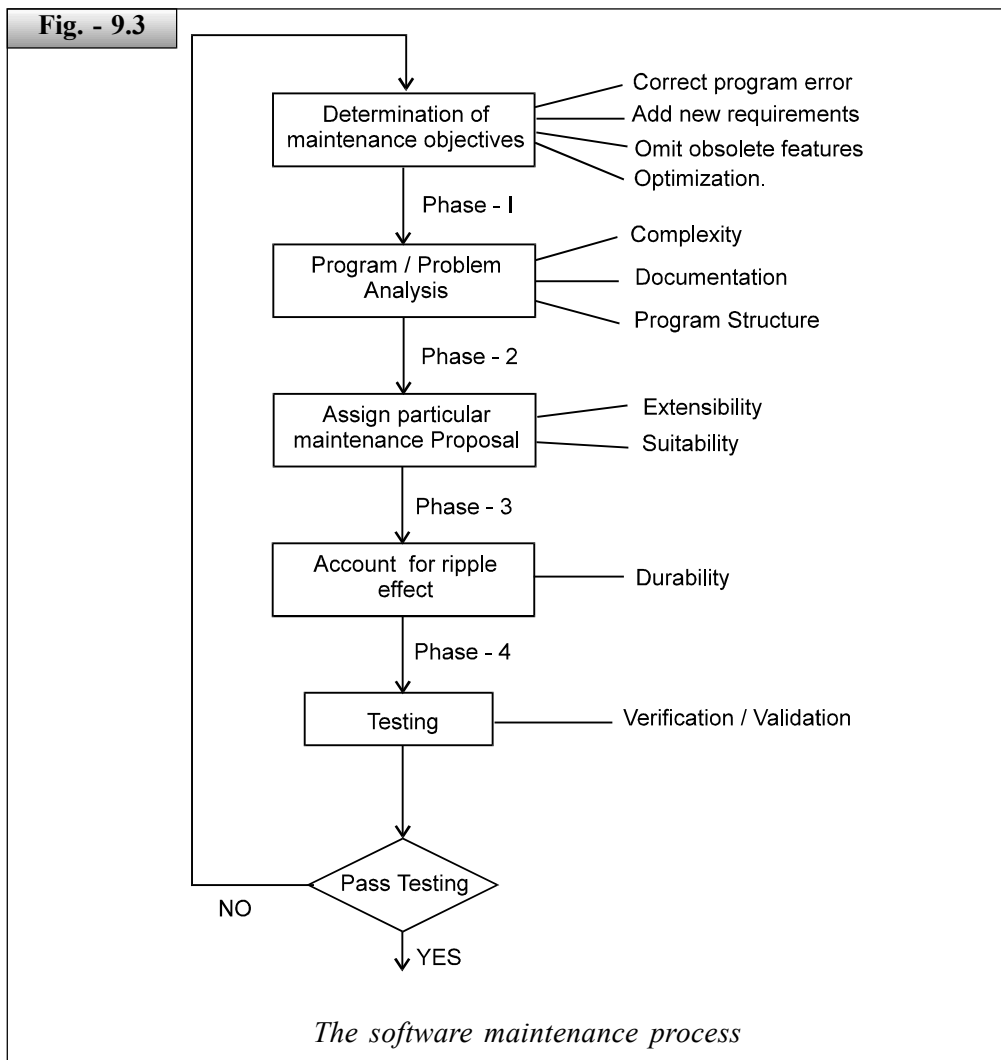
Complete replacement of the system is not usually a viable option. An operational system in itself can be an asset to an organization in terms of the investment in technical

knowledge and the advancement in working culture. The current system may need to have the potential to evolve to higher state, providing more sophisticated user driven functionality. The capacity of developing cutting edge technology, and of allowing the integration of other systems in cost effective manner.

9.6 MAINTENANCE PROCESS

As soon as the maintenance objective is setup, the maintenance personnel should be clear about the modification (what to modify). Then the program is modified to satisfy the maintenance objective. After the completion of maintenance process, the personnels who do it, they must ensure that *“the modification does not affect other portion of the program”*. Finally the modified programs are tested successfully with the objectives as of the client / user.

The maintenance process can be carried out in following phases. (as shown in fig.9.3)



The different phases of maintenance process are briefly described below.

Program / Problem Analysis

This phase is intended for analysing the problem (Program) to understand several attributes like complexity of the program, self- descriptiveness of the program under reference.

Complexity is a measure of effort to understand the problem and descriptiveness is measured in easy to read, understand and use.

Assignment of maintenance proposal

This phase is used to accomplish the implementation of the maintenance objective. This requires a clear understanding of both the objectives of maintenance and the program to be modified.

However, assigning a maintenance proposal is influenced by the extensibility of program attributes i.e. to what extent the program can support its extensions.

Ripple effect

It is a phenomena, where the modification of a particular problem (program) module affects the functionality (performance) of other problem modules, which is not at all desired. Therefore, it is necessary to understand the potential effect of a module under modification from both a logical and performance point of view.

Testing

This phase entertains the testing of a modified problem (program) module in cost-effective way.

It can be accomplished by using some techniques i.e. a measure of effort required to test the program w.r.t. some pre-defined test conditions.

Each of the four phases and their associated software attributes are critical to the maintenance process. So the maintainability should be made qualitatively for the better understanding of the software, its correctness, adaptability / interoperability and enhancement of software under modification (maintenance).

9.7 MAINTENANCE MODELS

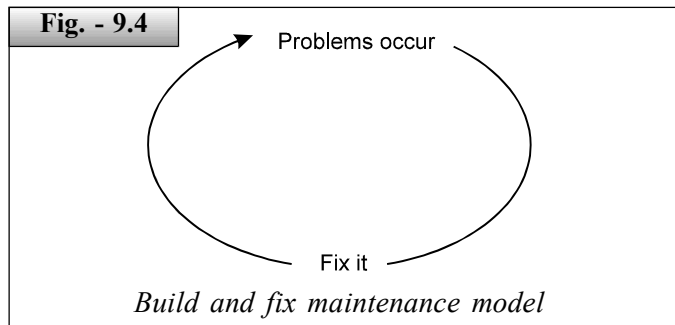
Why it is required ?

In most of the maintenance cases, the SRS and DDS are not traced-out, Hence some times it is very difficult to understand old codes.

Therefore several maintenance models were introduced to resolve the problems in maintenance.

9.7.1 Build and Fix model

Build and Fix is an ad hoc approach to maintain the software. It is just like a five fighting approach, waiting for the occurrence of the problem then trying to fix it as soon as possible.



In this model, the problem is fixed without analysing the details and its long term effects. For example, ripple effects (as described earlier) through the software or on the code structure;

Suppose a system is developed and maintained by a single person, he or she can come to learn the system well enough to be able to manage without detailed documentation, to be able to make instinctive judgements about the modification.

The task is done very quickly and in a cost effective manner.

If customer are demanding the rectification of errors, then they have to go through in such a way that, no risk and no detail verification of the system is required.

If the organisations follow up the build and fix model to develop the software, the organisation has to set up long term planning for maintenance purpose.

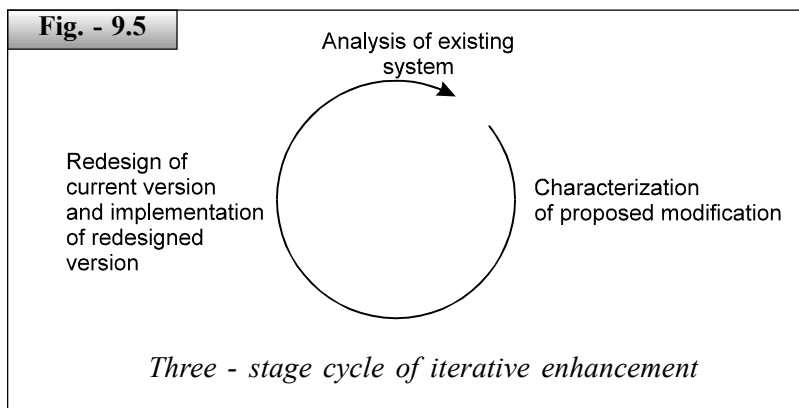
9.7.2 Iterative enhancement model

This model implies the changes in the software throughout its life time through an iterative process.

The iterative enhancement model is preferred to be applied in case of software development, where user requirements are not clearly defined to the designers.

This model is effective under three stages of life cycle.

- 1) Analysis
- 2) Characterization of proposed modification.
- 3) Redesign and implementation.



The existing documentation of each stage such as

Requirement,

Design

Coding

Testing and analysis

is modified, starting with the highest level document affected by the proposed changes. These modifications are propagated through a set of documents and the system to be redesigned.

This model exclusively supports reuse and also accommodates other model such as build and fix models.

The pressure of the maintenance environment often dictates that a quick solution is found but, as it is seen that, the use of the quickest solution can lead to more problems than it solves.

In the first iteration, problem areas would be identified and next iteration would specifically address the problem.

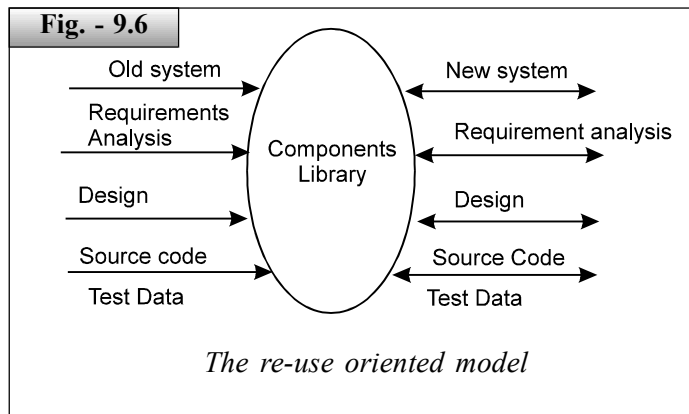
The problems with this model are the assumptions made about the existence of full documentation and the ability of the maintenance team to analyse the existing software product in full. But the problem is that in the present era of software development, it is too difficult to keep-up updates for the documentation of the software to be maintained.

9.7.3 Reuse - oriented model

This model is based on the principles that, *“Maintenance could be viewed as an activity involving the reuse of existing program components”*.

This model has four main steps :

- i) Identification of the part of the old system that can be reused.
- ii) Analysis of these identified parts to be reused.
- iii) Modification of these old system parts w.r.t. the new requirements.
- iv) Integration of these modified parts with new systems.

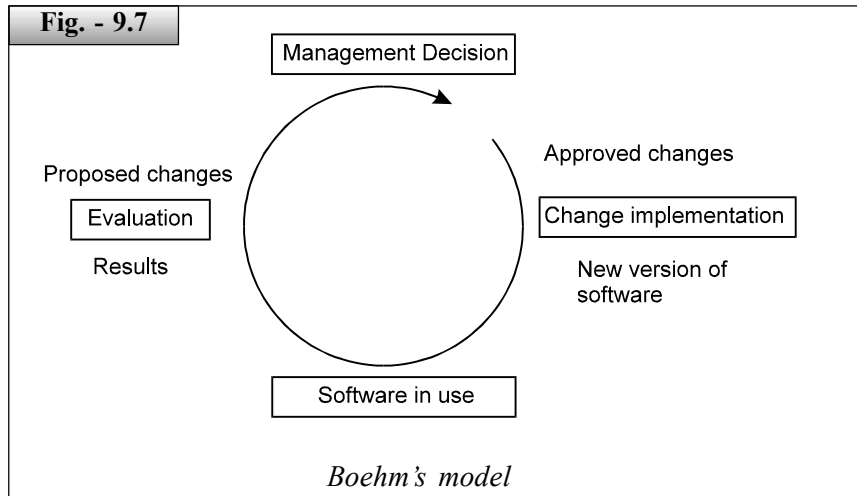


Like other models, this model comprises the life-cycle phases such as requirements, the design, the code or the test data.

9.7.4 Boehm's model

This model was proposed by Boehm in 1983. This model based on the principles of economic model. This model not only improve the productivity is the maintenance but also helps to understand the process.

Boehm represents the maintenance process as a closed loop cycle. This model has four stages of activities.



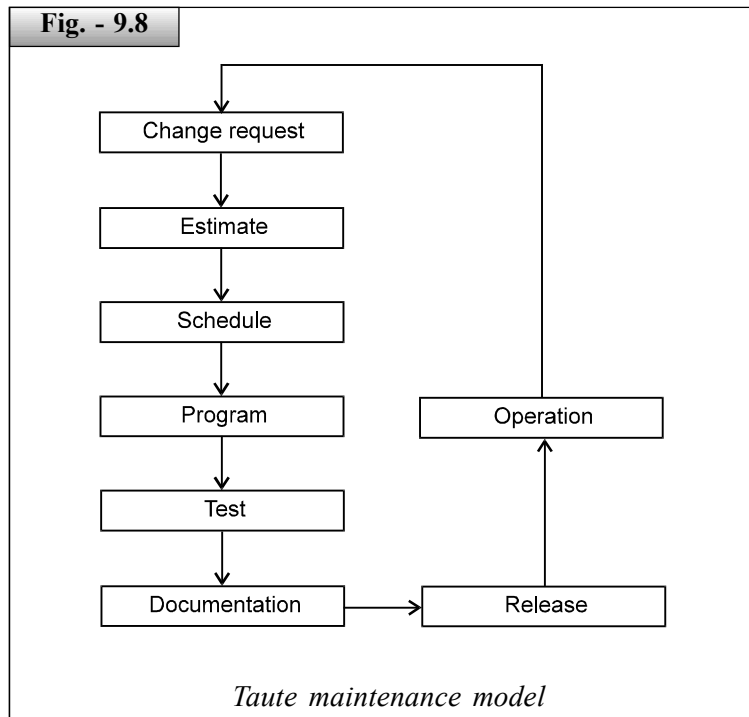
These stages include the *strategic decisions* to execute the processing activities.

In these stages, a set of approved changes is determined by applying particular strategy and cost-benefit evaluation to a set of proposed changes. The approved changes are accompanied by their own budgets which will largely determine the extent and type of resources expanded.

Boehm proposes the maintenance manager's task as one of the balancing the pursuit of the objectives of maintenance against the constraints imposed by the environment in which maintenance process carried out. Thus the overall maintenance process is controlled by the maintenance manager's decisions which are based on the balancing of objectives against the constraints.

9.7.5 Taute maintenance model

This model was proposed by *B.J. Taute* in the year of in 1983. It is very easy to understand and to implement. It is a typical maintenance model and it has eight phases in a cyclic fashion as shown in fig. 9.8.



P₁ : Change request phase

In the first phase, the maintenance team receives a request in a prescribed format from the client / clients regarding the changes of the system. This change may be any category of maintenance activities (corrective / adaptive / perfective / preventive) and assign a unique identification number to the request.

P₂ : Estimate phase

This phase is accountable to estimate the time and effort required to modify the system. Though it is too difficult to make an exact estimation for it, but it estimates these factors reasonably to the exact one. This phase analyses the factors related to the changes of the system and minimises the *ripple effects*.

P₃ : Schedule phase :

This phase identifies the change requests for the next scheduled release and also prepare a document that are necessarily required for planning.

P₄ : Programming phase :

This phase includes the modification of source code and its complementation according to the user requirement.

The final output of this phase is obtained as a “test version” of the source-code.

P₅ : Test phase :

In this phase the code is tasted in order to ensure that modification is correctly implemented. In this phase new test cases are also designed by using the previously available test cases. This is also same as regression testing.

P₆ : Documentation phase :

When the new test cases are obtained (*regression testing*), the system documentation as well as the user documentations are prepared and updated before releasing the system. It relates the code with its documentation properly.

P₇ : Release phase :

In this phase a new software product along with all possible updated documentation is delivered to the client. In this phase acceptance testing is carried-out by the user on the system.

P₈ : Operation phase :

After acceptance testing, the released software product is operated or used. During the period of use, if other problems or difficulties are encountered or new functionality requirement is desired or it is required to improve the existing capability of the system, again a “change request” process is established. In this approach, the whole phases are to be repeated to implement the change in the system.

9.8 MAINTENANCE COST ESTIMATION

It has been already discussed that the maintenance phase of a software product consumes 40 - 70% of the cost of the entire life cycle of software product. However this cost may vary widely from one application domain to another.

Therefore it is essential to invest more effort in early phases of software life-cycle to reduce the maintenance cost. The defect repair ratio increases heavily from analysis phase to implementation phase as given below.

Table-9.1 : Defect Repair Ration (DRR)

Phase	Ratio
Analysis	01
Design	10
Implementation	100

Hence, more effect during development will certainly reduce the cost of maintenance.

Note : Good software engineering techniques such as precise specification, loose coupling, and configuration management all reduce maintenance cost.

Belady and Lehman model

This model proposes that the effort and cost can increase exponentially if poor software development approach is used and the person or group that used the approach is no longer available to perform maintenance. It can be expressed by the following equation.

$$M = P + K_e(c-d)$$

Here

M → Total maintenance effort expended.

- P $\longrightarrow$ Productive effort that involves analysis, design, coding, testing and evaluation.
- K $\longrightarrow$ An empirically determined constant.
- C $\longrightarrow$ Complexity measure due to lack of good design and documentation.
- d $\longrightarrow$ Degree to which maintenance team is familiar with the software.

Notes : *The value of 'C' will be higher for a large software product with a high degree of systematic structure than a small one with the same degree.*

If the software is maintained without an understanding of the structure, function and purpose of the software, then the value of "d" will be low.

9.9 CHARACTERISTICS OF SOFTWARE EVOLUTION

In 1980, *Belady* and *Lehman* postulated the characteristics of evolution of several software products. They have expressed their observations in the form of laws. These important laws are given below.

Lehman's first law :

"A software product must change continually or become progressively less useful."

According to this law :

"A software product continues to evolve through maintenance effort after its development. Large software products under go operation for a long duration as it is very difficult and expensive to afford the maintenance cost and effort needed for it.

This law indicates that every software product must undergo maintenance irrespective how well it might have been designed.

When it is not affordable to give maintenance to a software product, it must be assumed that, that software product attains at its retirement stage.

Lehman's second law :

"The structure of a program tends to degrade as more and more maintenance is carried out on it".

The structure of a software product degrades because of new or new functions to it during the maintenance process. In this manner, the existing software system is redesigned in order to reduce the complexity occur due to addition of new functionality, and the final software product is likely to be of a simple structure type.

Lehman's third law :

"Over a program's lifetime, its rate of development is approximately constant".

The matrices like *Lines of codes (LOC)*, *Function Points (FP)* or *Feature points (F_p)* are the quantifiable measurement for measuring the rate of development or rate of modification of a software product.

This law states that the rate at which the code is written or modified is approximately the same during development and maintenance.

Example-9.1 :

The Development effort for a software project is 1500 person-months. The empirically determined constant (K) is 0.9. 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 project ($d = 0.1$).

Solution :

$$\begin{aligned}\text{Development Effort } (P) &= 1500 \text{ PM} \\ K &= 0.9 \\ C &= 8\end{aligned}$$

- (i) Maintenance team has good level of understanding ($d = 0.9$)

$$\begin{aligned}M &= P + Ke^{(C-d)} \\ &= 1500 + 0.9e^{(8-0.9)} \\ &= 1500 + 1090.77 = 2590.77 \text{ pm.}\end{aligned}$$

- (ii) Maintenance team has poor level of understanding ($d = 0.1$)

$$\begin{aligned}M &= P + Ke^{(C-d)} \\ &= 1500 + 0.9e^{(8-0.1)} \\ &= 1500 + 2427.55 = 3927.55 \text{ pm.}\end{aligned}$$

Hence, it is clear that effort increases exponentially, if poor software engineering approaches are used and understandability of the project is poor.

In 1981, Boehm proposed a formula for estimating maintenance costs as a part of COCOMO Model. By using the collected data from different software projects.

This formula was derived in terms of maintenance effort. Boehm used a quantity named Annual Change Traffic (ACT).

The Annual Change Traffic (ACT) is defined as :

“The fraction of the source instructions of a software product undergo changes (maintenance) during a year either through addition, deletion or modification”.

The Annual Change Traffic (ACT) is clearly related to the number of change requests.

$$ACT = \frac{KLOC_{\text{added}} + K_{\text{LOC deleted}}}{K_{\text{LOC Total}}}$$

Here $KLOC_{\text{added}}$ is the total kilolines of source code added during maintenance.

$KLOC_{\text{deleted}}$ is the total $KLOC_{\text{deleted}}$ during maintenance.

Hence the code that is changed should be counted in both the code added and the code deleted.

The Annual Maintenance Effort (AME) in person-months can be worked out as

$$AME = ACT \times SDE$$

here, SDE $\longrightarrow$ software Development Effort in person-months.

ACT $\longrightarrow$ Annual Change Traffic.

Suppose, a software project required 800 person-months of development effort and it was estimated that 25% of the source code could be modified in a year. The Annual Maintenance Effort (AME) will be $0.25 \times 800 = 200$ person months.

Boehm proposed that this calculation can be rectified by evaluating the importance

of those factors, affecting the cost estimation techniques.

By using these factors, the Effort Adjustment Factor (EAF) can be evaluated as in the case of COCOMO model.

The modified equation is given below :

$$AME = ACT \times SDE \times EAF.$$

Most of the maintenance cost estimation models, however, approximately estimates the cost, as they do not take into account several factors such as the experience level of software developers, and their attachment with the software product, hardware requirements and software complexity.

Example-9.2

Annual Change Traffic (ACT) for a software product is 10% per year. The development effort is 400 PMS. Calculate the Annual Maintenance Effort (AME), if the life time of the software product is 12 years. Also calculate the Total maintenance effort of the software product.

Solution :

The Development effort = 400 PMS.

Annual Change Traffic (ACT) = 10%

Total duration for which effort is to be estimated = 10 years.

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

$$\begin{aligned} AME &= ACT \times SDE \\ &= 0.10 \times 400 = 40 \text{ PM} \end{aligned}$$

∴ Maintenance effort for 10 years = $12 \times 40 = 480 \text{ PM}$.

∴ Total effort = $400 + 480 = 880 \text{ PM}$.

Example-9.3

A software product has development effort 1000 PM. It is assumed that 20% code will be modified per year.

Some of the cost multiplying factors are given as :

- i) Required software Reliability (RELY) : Very high = 1.40
- ii) Data base size (DATA) : Very high = 1.16
- iii) Analyst capability (ACAP) : high = 0.71
- iv) Application experience (AEXP) : Very high = 0.82.
- v) Programming Language Experience (LEXP) : high = 0.95

Other multiplying factors are normal

Calculate the Annual Maintenance Effort (AME).

Solution :

Annual Change Traffic (ACT) = 20%

software Development Effort (SDE) = 1000 PM.

The effort adjustment factors are :

$$\text{RELY} = 1.40$$

$$\text{ACAP} = 0.71$$

$$\text{AEXP} = 0.82$$

$$\text{LEXP} = 0.95$$

$$\text{DATA} = 1.16$$

other factors are assumed to be normal.

Therefore

$$\text{EAF} = 1.40 \times 0.71 \times 0.82 \times 0.95 \times 1.16 = 0.8982$$

$$\begin{aligned}\text{AME} &= \text{ACT} \times \text{SDE} \times \text{EAF} = 0.2 \times 1000 \times 0.8982 = 179.64 \text{ PM} \\ &= 179.643 \text{ PM} \quad (\text{Ans}).\end{aligned}$$

9.10 SOFTWARE CONFIGURATION MANAGEMENT

During software development process the results are obtained in the form of objects (source code, design document, SRS document, test document, and user's manual etc.). These objects are usually analysed and modified by a number of software engineers throughout the software life cycle.

The state of all these objects at any point of time is referred to as the configuration of the software product. The state of each result or output are referred to as deliverable.

Software configuration management deals with effectively tracking and controlling the configuration of a software product during its life cycle.

The software configuration management (S.C.M) is associated with “*version* and *reversion*” of a software product.

The configuration management team is responsible for assuring that each *version* and *releases* is correct and stable before it is released for use, and that changes are made accurately and promptly. Accuracy is critical because we want to avoid generating new faults while correcting existing ones.

Similarly, promptness is important, because *fault defection* and *correction* are proceeding at the same time that the test team searches for additional faults. Thus, those who are trying to repair system faults should work with components and documentation that reflect the current state of the system.

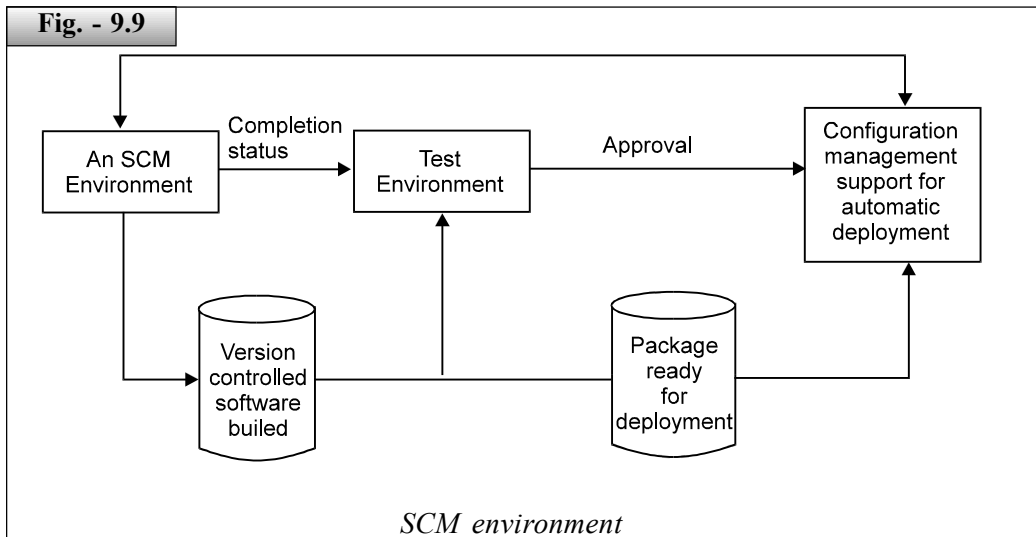
Hence, Software Configuration Management is defined as following :

It is an act of identifying, organizing and controlling modifications to the software being built by a programmer team.

The software Configuration Management maximizes the productivity and Minimizes the risks associated with the software development.

OR Software Configuration Management (SCM) is a set of activities designed to control change by identify the work product that are likely to change, establishing relationship among them, defining mechanism different versions of these products, controlling the changes imposed, auditing and reporting on the changes made.

Configuration Management produces an environment that includes the processing systems, tools, methods to identify different version of the system.



Configuration management plan written during the project planing phase lists all controlled objects.

9.10.1 Version and Releases

A new *version* of a software product is created when there is significant change in functionality, technology or the hardware on which it runs. For example

One *version* of a mathematical computation package might run on unix-based machines, another on microsoft windows and so on. As a software is released and used by the customer, errors are discovered that need correction. Enhancement to the functionalities of the software may also be needed. A new release of software is an improved system intended to replace an old one. Often the systems are described as **version m, release n**, or simple *m.n*. Finally a history relation, *is version of*, can be defined between objects. This relation can be split into two subrelations : *is revision of* and *is variant of*. We first discuss the necessity of *configuration management* and subsequently we discuss the configuration management and subsequently we discuss the configuration management and tools.

Therefore, a configuration for a particular system is sometimes called a *version*. Thus, the initial delivery of a software package may consist of several versions, one for each platform or situation in which software will be used. For example, aircraft software may be built so that version 1 runs on Air force planes, and version 3 runs on commercial airliners.

Likewise a new release of the software is an improved system intended to replace the old one. Often, software systems are described as *version n, release m*, as version *m.m*, where the number reflect the system's position as it grows and matures. Version *n* is sometimes intended to replace version *n-1*, release *m* supersedes *m-1*.

9.10.2 Version and Release management

Version and release management are the processes of identifying and keeping track of different versions and releases of a system. New system version should always be created, by the CM team rather than the system developer, even when they are not intended for external release. This makes it easier to maintain consistency in the configuration database as only the CM team can change version information.

A system version is an instance of a system that differs, in some way, for other instances. New versions of the system may have different functionality, performance or may repair system faults. Some versions may be functionally equivalent but designed for different hardware or software configuration. If there are only small differences between versions, one of them is sometimes called a variant of the other.

A system release is a version that is distributed to customers. Each system release should either include platform. There are always many more versions of a system than releases as version are created within an organisation for internal development or testing that are never released to customers.

9.10.3 What is Milestone and Deliverable ?

Milestones are the end product of a particular development phase in a software development process.

Deliverable are the output or results obtained from the system wherever required by the user to be handed over.

In software configuration management, a number of milestones are there to have a single deliverable.

According to IEEE : *The Software Configuration Management (SCM) is a process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items.*

9.10.4 Software Configuration Management activities

The Software Configuration Management (SCM) undergoes three vital activities during the software development process, where versions, reversions, releases, deliverables and / milestones are identified and measured. These activities are :

1. Configuration identification
2. Configuration control
3. Configuration accounting

1. Configuration identification : it includes the identification of these part of the system, should be track of during the development process.

Generally, the Project Manager classifies the object associated with the software development process in to three main categories :

- *Controlled*
- *Pre-controlled &*
- *Un-Controlled*

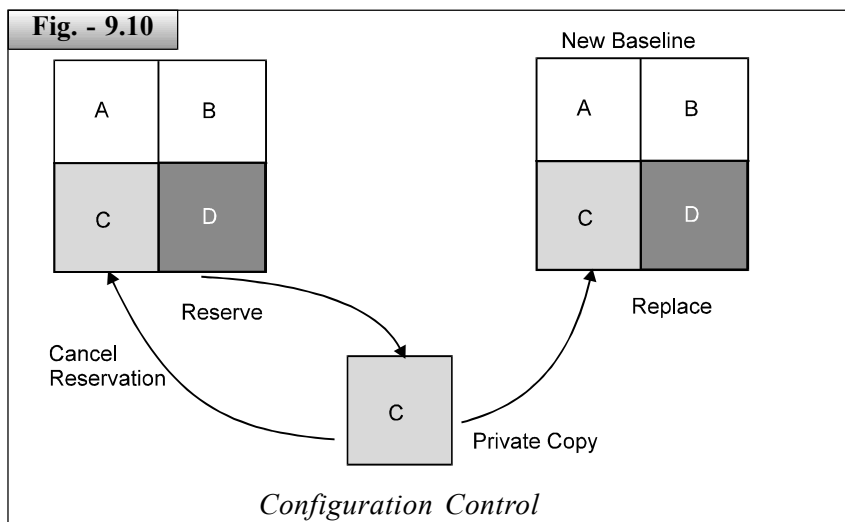
- a. Controlled object are those, which are already put under configuration control.
- b. pre-controlled object are not yet under configuration control, but will eventually be under configuration control.
- c. Uncontrolled objects are not and will not be subject to configuration control.

The controllable objects include both controlled and pre controlled objects.

- Requirement specification document
- Designing document.
- Tools used to build the system, such as Compilers, Linkage Lexical analysers, parsers etc.
- Source code for each module.
- Test cases
- Problem reports.

2. Configuration Control :

- Configuration control is the process of managing the modification or changes in “controlled objects”.
- The configuration control s/m prevent unauthorized modifications or changes with the controlled objects.
- In order to modify a controlled object, a private copy of the module on which the modification is required, prepared by the software developer through a reserve operation as shown in following figure.



- In this case, the configuration management tools permit only 1 person or programmer / developer to reserve a module at any instant of time for operation at any instant of time.
 - Once an object is reserved for operation, it doesn't allow anyone else to reserve this module again until the reserved module is restored.
- Similarly, by preventing more than one software developer to simultaneously reserve a particular module, the problems associated with simultaneous access are overcome.

Configuration Control Board (CCB)

The configurations of a software project can be controlled by an expert groups of people or software engineers responsible for software configuration management, called Configuration Control Board (CCB).

- The CCB evaluates the responses / request based on its effect on the software project, and the benefits due to change in the software configuration.
- The aim of configuration control is to develop a software product in a stable environment.

As mentioned in the above figure for e.g. module A is required to be integrated with module B and module C, if module C is kept on changing, there is a problem of continue the development process in module A & B.

- As soon as a module under configuration control is updated, the updated version is frozen and it is called a '*baseline*'.

3. Configuration Accounting

- It includes two essential methods
 - 1) Status accounting
 - 2) Configuration Audit

Status accounting :

- If the changes are occurred there in the baseline, some mechanism must be used to record how the system evolves and what its current stage is ?

This task is accomplished by status accounting. Its basic function is to record the activities related to other software configuration management functions.

The status accounting is also known as configuration status reporting. It emphasizes following questions ? During the process :

- i) What happened ?
- ii) Who did it ?
- iii) When it happened ?
- iv) What will be affected ?

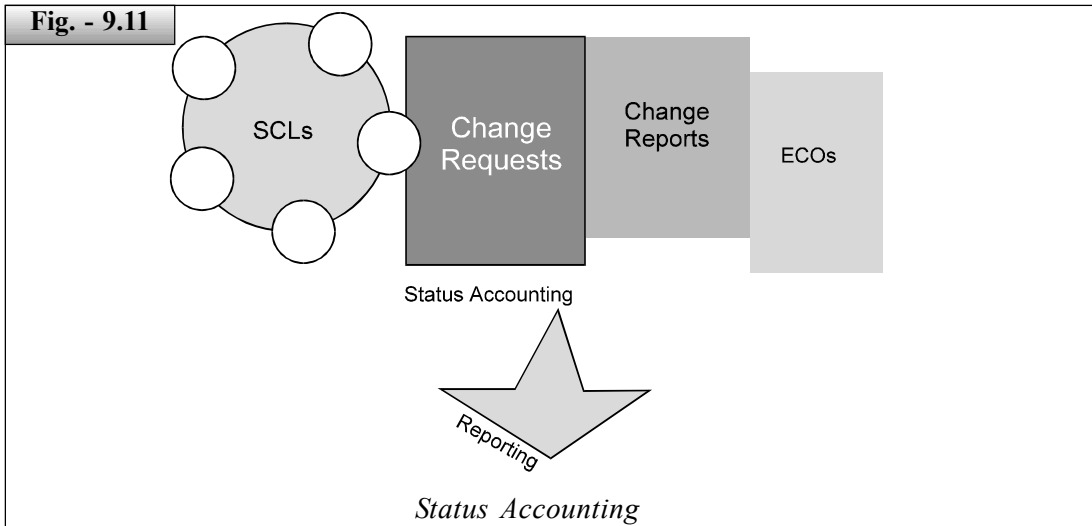
The flow of information for configuration status reporting (CSR) is maintained during the change control process.

In each time a Software Configuration Identification (SCI) is assigned new or an updated identification, a CSR entry is made.

In each time the change is approved by the configuration control authority (CCA) (i.e. an ECO is issued) a CSR entry is made.

In each time a configuration audit is made / conducted and the results are reported as a party of the CSR task.

Fig. - 9.11



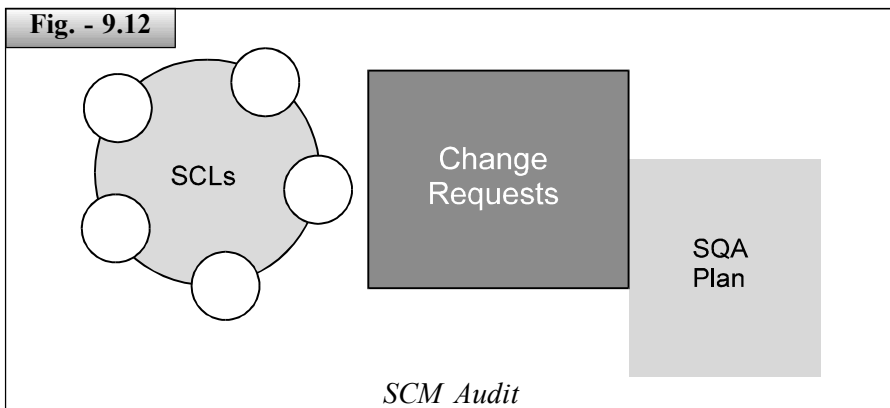
Configuration audit

Software configuration audit complements the Formal Technical Review (FTR) by accessing a configuration object for characteristics that are generally not comes into account during review the audit task and encounters following challenges :

- Has the change is made in the ECO or CCA specification ? or any additional modifications are incorporated ?
- Has formal Technical Review (FTR) been conducted to access technical correctness ?
- Has the change been “highlighted” there in the SCI ? have the date of change and author of change been specified ? Do the attributes of the configuration objects reflect the changes ?
- Have SCM procedures for “no-change”, recording it, and reporting it been followed ?
- Have all related SCIs been properly updated ?

The SCM audit revolves around the software configuration items, change requests, and software quality Assurance plan.

Fig. - 9.12



9.11 CHANGE CONTROL PROCESS

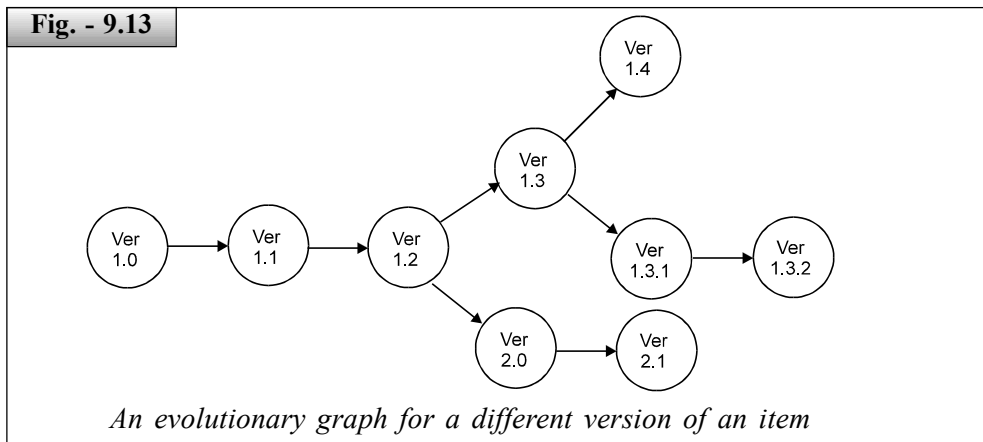
At any moment, the configuration management team must know the state of any component or document in the system. Consequently, configuration management should emphasize communication among those whose actions affect the system. Cashman and Host (1980) suggest that we always know the answers to the following questions :

- Authentication : Was the change made correctly ?
- Authorization : Who authorized that the change be made ?
- Cancellation : Who can cancel the request for change ?
- Delegation : Who is responsible for change ?
- Identification : Who made the change ?
- Naming : What components of the system were changed ?
- Synchronization : When was the change made ?
- Valuation : What is the priority of the change ?

Software Version Control :

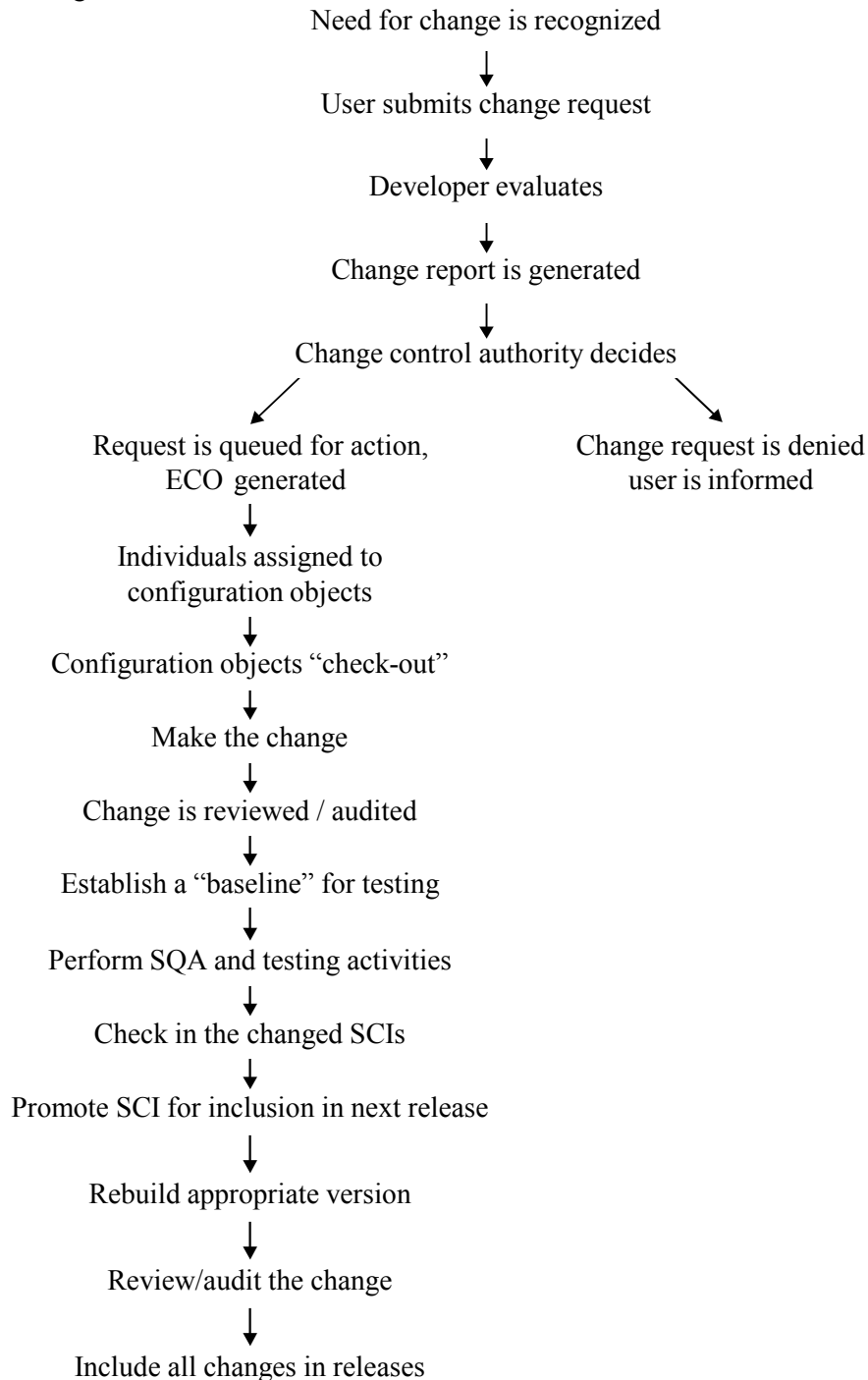
During the process of software evolution, many objects are produced, for example files, electronic documents, paper documents, source code, executable code and bitmap graphics. A version control tool is the first stage towards being able to manage multiple versions. Once it is in place, a detailed record of every version of the software must be kept. This comprises the

- Name of each source code component, including the variations and revisions,
- The versions of the various compilers and linkers used,
- The name of the software staff who constructed the component,
- The date and the time at which it was constructed.



The above evolutionary graph depicts the evolution of a configuration item during the development life cycle. The initial version of the item is given version number ver. 1.0. Subsequent changes to the item which could be mostly fixing bugs or adding minor functionality is given as ver 1.1 and ver 1.2. After that, a major modification to ver 1.2 is given a number ver 2.0 at the same time, a parallel version of the same time without the major modification is notice that these questions are management questions, not technical ones. We must use procedures to manage change carefully.

Change control combines human procedures and automated tools to provide a mechanism for the control of change. The change control process is illustrated schematically in figure below.



maintained and given a version number 1.3.

Commercial tools are available for version control, which perform one or more of following tasks :

- Source code control
- Revision control
- Concurrent version control

There are many commercial tools like Rational Clear Case, Microsoft Visual Source Safe and a number of other commercial tools to help version control.

SUMMARY

- Software maintenance is an major activity in software engineering, and is incorporated throughout the software development process for any changes in codes documentation, manuals and other components of the software product. Software maintenance activity is classified as Corrective, Adaptive, Perfective and Preventive for rectification of problems, environmental changes, performance improvement and detection and correction of internal faults etc.
- Maintenance process is comprised of a set of proposals along with some well defined models like- Build and Fix, Iterative model, Re-use model, Boehm's maintenance model, Taste maintenance model etc.
- Maintenance cost is a major challenge in the maintenance process, which is tactfully estimated by using some pre-defined laws.
- Like maintenance, software configuration management activity is applied throughout the software development process. SCM (Software Configuration Management) identifies, controls, audits and reports modifications that invariably occur while software product is being developed and after it has been released to customers.
- The software configuration is composed of a set of interrelated objects called software configuration items, that are produced as a result of some software engineering activity. In addition to documents, programs and data, the development environment that is used to create software can also be placed under control.
- Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. Configuration audit helps to ensure that the quality of the product is maintained as changes are made.

EXERCISES

1. What is software maintenance ? Describe various categories of maintenance. Which category consumes maximum effort and why?
2. Some people feel that "maintenance is manageable". What is your opinion about this issue?

3. Discuss Lehman's law for software maintenance. Why software needs maintenance?
4. Explain the Boehm's maintenance model with the help of a diagram.
5. Describe various maintenance cost estimation models.
6. Write a short note on Belady and Lehman model for the calculation of maintenance effort?
7. Annual change traffic (ACT) in a software system is 40% per year. The initial development cost was Rs. 25 lacs. Total life time for software is 12 years. What is the total cost of the software system.
8. What is the need of maintenance?
9. What are different types of maintenance that a software product might need? Why is such maintenance required?
10. What do you understand by software configuration? What is meant by software configuration management? How can you manage software configuration (only mention the names of the principle activities involved).
11. What is the difference between a revision and a version of software products?
12. What do you understand by the terms change control and version control? Why are these necessary?
13. Explain how change and version control are achieved using a configuration management tool.
14. Write short notes on
 - (a) Status Accounting
 - (b) Configuration Audit
15. What do you mean by software configuration management? Also discuss in brief the various software configuration management activities.
16. Write a note on configuration identification.
17. What are configuration management activities? Draw the schematic diagram of change control process.
18. List three common types of risks that a typical software project might suffer from?
19. Write a short note on "software version control"?
20. Using a schematic diagram show the order in which the following are estimated in the COCOMO estimation techniques: cost, effort, duration, size.
21. What are the various reasons for poor/inaccurate estimation.
22. Discuss various types of COCOMO model. Explain the phase wise distribution of effort.
23. Describe the basic COCOMO model in detail?
24. Why cost estimation plays an important role in software development process? Discuss COCOMO model in detail?
25. What is risk? Is it economical to do risk management? What is the effect of this activity on the overall cost of the project?
26. What are the two major sources of risk? Explain the three categories of risks that a software project might suffer from.

Software Re-engineering

10.1 INTRODUCTION

Let's consider a product (system) that has served you well. You use it regularly, but it's getting old (don't you agree ?) it breaks to often, takes longer to repair than you would like, and no longer meets the newest requirements.

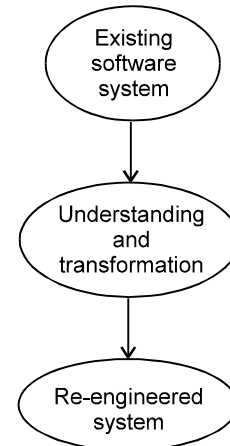
What to do ? If a product is a hardware, you may feel to throw it out and buy a new one, but if it's a software, you must think twice to do so. Obviously, you need to re-build it. You will create a product (Software) with some added (required) functionality, better performance, reliability and improved maintainability.

Yah.... we may sense it as re-engineering.

Definition : There is no universal definition to software- Re-engineering, rather it can be stated as follows.

- It's a process of modifying the internal mechanism of a system / Software product without changing its functionality.
- It's a technological activity to keep pace of a product (system) in the competitive environment for changing world's customers.
- It is an intensive way of creating new versions / release of existing product (Software) for better quality and maintainability.
- It's an incremental approach to improve the performance of existing system at lower cost, lower risk and efficient use of available staff through technical way.

Fig. - 10.1



Abstract view of Re-engineering

“Re-engineering is a rebuilding activity” : An illustration

Suppose we have purchased a house in another location far from us, i.e. we have never seen the property, but purchased it at an exciting low price with a warning that it might have to be completely rebuilt, now the question is how we proceed ?

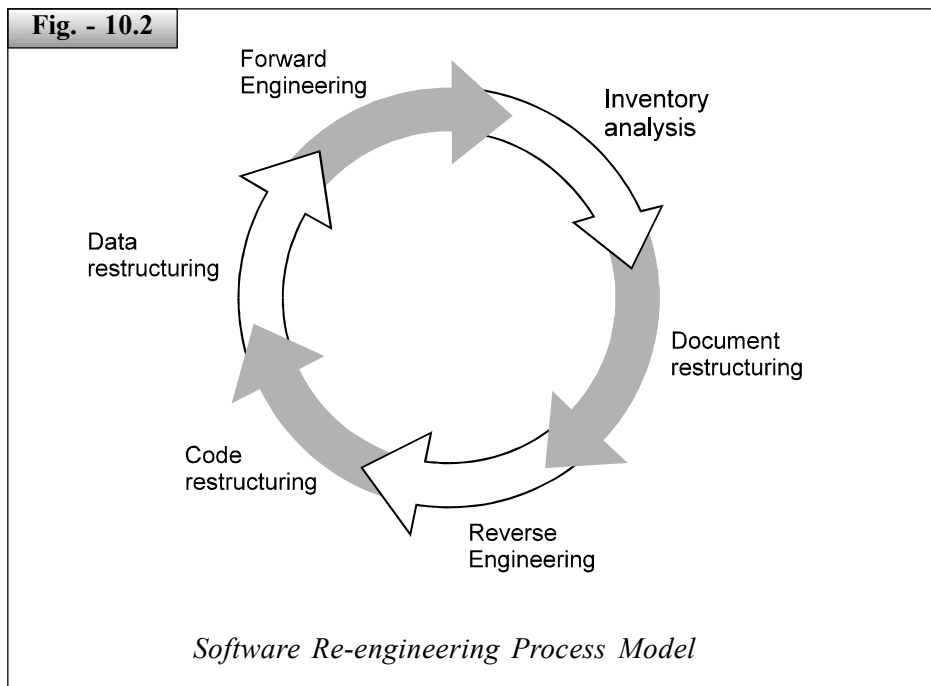
- Before we start, we must inspect the house to verify whether it needs to be rebuilt. We must have a list of criteria for inspection by us or a professional group.
- We must ensure that the structure of the house is weak or it is good enough, so that it may be possible to “*remodel*” without rebuilding at much lower cost and lesser time.
- Before we start rebuilding, we must ensure to gather the information i.e. how the original house was built. It will help to start construction.
- If we start to rebuild, we would use only the most modern, long lasting materials. No doubt, we prefer to use the latest technology (if available) to do so, it may cost a bit more, but we can avoid expensive and time consuming maintenance later on.

Although these principles focus on the rebuilding of a house, they can be applied equally well to the re-engineering of computer based systems and applications.

Here, we look for the *software Re-engineering* process model.

10.2 SOFTWARE RE-ENGINEERING PROCESS MODEL

This process model defines six different activities as shown in the fig.(10.2). In general activities occur in a linear sequence, but it is not always the case. For example reverse engineering may have to occur before the document re-constructing can commence.



10.2.1 Inventory analysis

Every software developer group or organisation should have an inventory of all their products (software). Inventory is something in form of a spread sheet containing the detail description of the software. e.g. size, age, status, maintainability and any other constraints and influencing factors etc.

It is important that the inventory should be revisited at a regular time interval as the status of product (software) can change as a function of time and as a result, priorities for Re-engineering will shift.

10.2.2 Document restructuring

A software organisation must choose one of the following options while restructuring the documents.

- *Documentation is too time consuming.*

If an existing software is working and is relatively static, and about reaching to the end of its useful life and can't be changed significantly, better leave as it is, without consuming any more time.

- *Documentation must be updated, but with limited resource.*

It may not be necessary to fully document the software, rather the portions of the product currently under change are fully documented.

- The product is critical and need to be fully re-documented.

In this case, an intelligent approach is required to do so.

10.2.3 Reverse engineering

It is a process of analyzing a program in an effort to create a representation of the program (software) at a higher level of abstraction than the source code. It can also be a process of design recovery. It improves the software for increased maintainability, reusability etc.

It can also be sensed as a two step process of “information extraction” followed by “information abstraction”.

- analyse the subject product (software) to identify its components and their inter-relationship.
- create representation of the product in another form or at a higher degree of abstraction.

The new approach of *Reverse engineering* was reconstructed by *Scott Tilley in 1995*, that refines the two step approach of information extraction and abstraction into three step approach as follows :

- Construct “*domain-specific models*” of the application (product) using conceptual modelling techniques.
- Gather the raw data from the subject system (software) using appropriate “*extraction mechanism*”.
- Create abstraction that facilitates program understanding, analysis and presentation of resultant information structures.

Reverse engineering enables to understand the process through

- identification of artifacts,
- their relationship
- and generation of abstraction.

10.2.4 Code re-structuring

The software product may have a very good program architecture, but individual modules have coded in a way that makes them difficult to understand, test and maintain. In such cases, the code within the suspect modules can be restructured.

Here the source code is analysed using restructuring tool. The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Finally the internal code documentation is updated.

10.2.5 Data re-structuring

Unlike code-restructuring, which occurs at a relatively low level of abstraction, data restructuring is a full scale *Re-engineering* activity. Actually data architecture has a better importance to define product *viability* than the source code itself. Where a product having weak data architecture, it is dissected and necessary data models are defined. Data objects and attributes are identified and existing data structures are reviewed for quality.

10.2.6 Forward engineering

It is also called “*renovation or reclamation*”, not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. It helps to re-implement the function of the existing system and also adds new functions and / or improves overall performance.

10.3 ADVANTAGES OF SOFTWARE RE-ENGINEERING

Software Re-engineering has the following advantages :

- **Lower cost :**
Re-engineering an existing software product costs comparatively less than new system development.
- **Lower risks :**
Re-engineering is based on incremental development of systems, rather than radical system replacement. So there is very lower risk of losing critical software knowledge.
- **Efficient use of existing staff :**
Existing staff can be expertised for the re-engineering process because of the incremental nature of re-engineering which is less risk and less expensive.
- **Incremental development :**
Re-engineering can be carried out in stages, as per the budget and resource available. The end users are able to gradually adapt to the re-engineered software product as it is delivered to them in increments by the operational organisation who re-engineers.

10.4 REVERSE, FORWARD AND RE-ENGINEERING : A COMPARATIVE STUDY

Software Re-engineering is a two level process i.e.

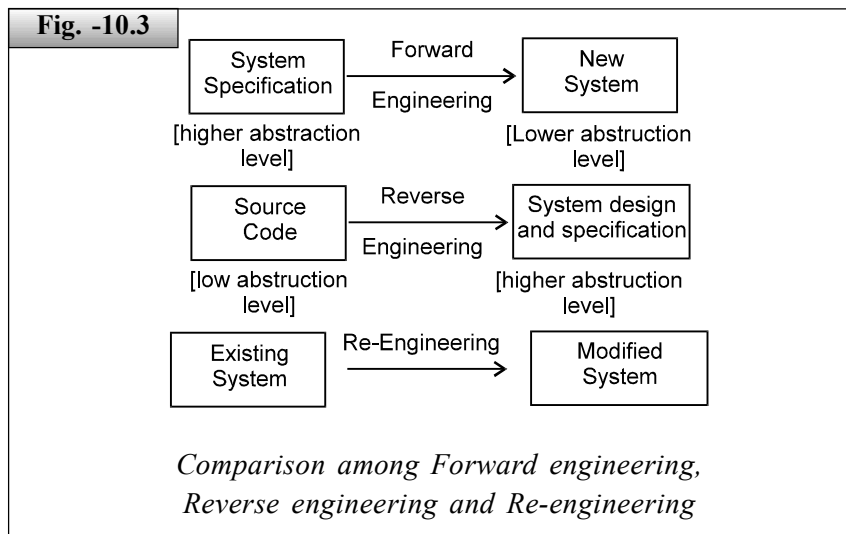
- *Reverse Engineering*
- *Forward Engineering*

Generally, reverse engineering is to understand the structure and behaviour of the software product, to process the product and describe information on an abstract level.

The objective of reverse engineering is to derive the design and specification of a system from its source code.

Likewise, forward engineering corresponds to the customary procedures of the system generation. It is to modify the existing software product until the product is fully developed as desired.

An illustration can be made as shown in the fig. 10.3.



10.5 IMPORTANCE OF REVERSE ENGINEERING

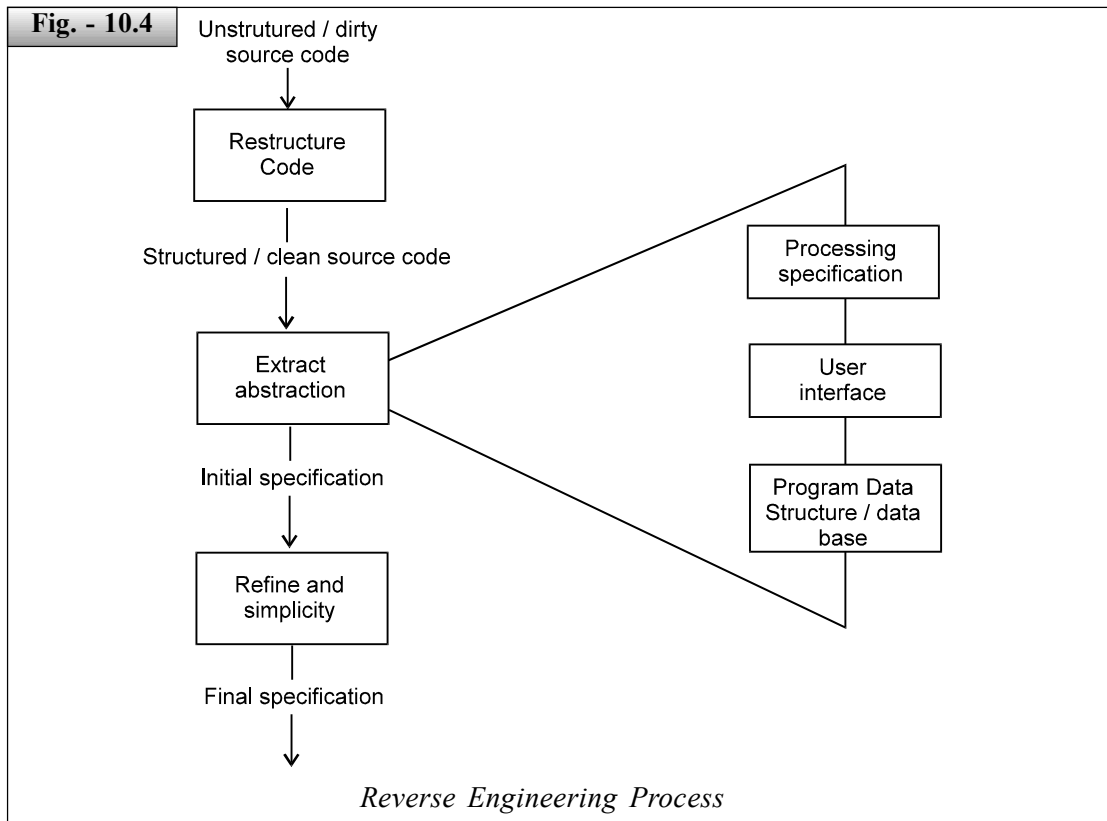
We list out some situations where the presence / involvement / implementation of reverse engineering can be felt.

- Inadequate documentation of the original design.
- Loss of design documentation.
- Product (existing) having anomalies
- To analyze and improve the product performance and features.

10.6 REVERSE ENGINEERING PROCESS

Reverse engineering process entertains a set of phases starting from existing source code (product under reference) to the final specification of the modified product.

The process can be represented as shown in fig. 10.4

Fig. - 10.4

During Reverse engineering process, the unstructured or dirty source code is fed into the “Restructure code” phase to extract the design information from the source code. (*structured / clean code*) next, the structured / clean code undergoes the process of abstraction (*processing specification, user interface, data structure / database*) which is capable of deriving procedural design representations (*low level abstraction*), programs, data structure information (higher level abstraction), object models, data and control flow models (most higher level abstraction).

The completeness of the documentation, the degree to which the software tools and software analyst work collectively and the directionality (how to proceed ?) of the process are highly variable.

Finally a clear, refined and simple specification is produced for further reference.

If the directionality of the process is one way, then all information extracted from the source code is provided to the software developer.

If it is of two ways, the information is fed to a Re-engineering tool (discussed later) to restructure / re construct the old product (software)

10.7 LEVELS OF REVERSE ENGINEERING

As the prime activity of Reverse engineering is to deal with the degree of abstraction of a software product, based on this (*degree of abstraction*), three important types (*levels*) of Reverse engineering are

- Re-documentation
- Structural redocumentation
- Design recovery

10.7.1 Redocumentation

It is one of the oldest forms of reverse engineering. It produces an exact documentation for an existing software system, where the system / product of a Reverse engineering process is at the same level of abstraction as of the original system.

The importance of this process is,

- to create alternative views of the system to enhance understanding generation of hierarchical data flow or control flow diagram from source code.
- to improve current documentation, which may be required during development and updation of a system.
- to generate documentation for a newly modified program (*facilitates preventive maintenance on the system*).

10.7.2 Structural redocumentation

As of redocumentation, structural redocumentation does not involve physically restructuring rather to reconstruct the architectural aspects of a software system, especially to expose the structure of large software systems.

This process enables the software engineers, technical managers, maintenance personnel to explore the low-level source code and combine it with the redocumented material to form high-level structural models. (*large documentation*).

10.7.3 Design Recovery

Design recovery deals with identification and extraction of meaningful higher-level abstractions which is not obtained directly from examining the source code. This may be achieved from the combination of code, existing design documentation, past experience. Knowledge of problem and application domain. The recovered design can then be used for redeveloping the system i.e. the resulted design becomes a baseline for future modification of the system.

There are different approaches used to recover the designs. Different programming constructs which vary from language to language are used to identify, understand the design decisions in a source code. The program constructs are control and data structures, variables, procedures and functions, definition and implementation modules, and class hierarchies.

10.8 REVERSE ENGINEERING TOOLS

Reverse-engineering process is carried out with the help of certain tools, without what it could have quite impossible to do the job in time and within the budget.

Some of the Reverse engineering tools are :

- **Restructurer** : detects poorly structured code fragments and replaces them by suitable equivalent code.
- **Cross referencer** : Lists the locations / places where each variable is defined and used.

- **Static analyser** : detects anomalous constructs like uninitialised variables, undefined symbols dead codes etc.
 - **Text editor** : Supports browsing and editing the source code.
- In addition to the above tools, some other reverse engineering tools may be used for GUI (Graphical User - Interface) and other visualisation tools.
- e.g. : ➤ RIGI (Interactive graph editor for graphical representation of a software system)
- Refine Language Tools (for analyzing and re-engineering code in programming languages including Ada, C, COBOL, FORTRAN etc.)

SUMMARY

- Re-engineering occurs at two different levels of abstraction i.e. at business level and at software level.
- At business level, re-engineering focuses on the business process with the intent of making changes to improve competitiveness in some area of the business, where as at software level, re-engineering examines information systems and applications, with the intent of re-structuring or reconstructing them so that they exhibit higher quality
- Software re-engineering encompasses a series of activities, that include inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. The intent of these activities is to create versions of existing programs that exhibit higher quality and better maintainability.

EXERCISE

1. What is software re-engineering ? Write down its importance.
2. Sketch and discuss in detail about the S.R. process model.
3. Discuss a comparison among Reverse, forward and Re-engineering.
4. Write down the importance of reverse engineering.
5. List out the situations, where all these three approaches are supplemented.



Computer Aided Software Engineering

11.1 INTRODUCTION

CASE stands for *Computer Aided Software Engineering*.

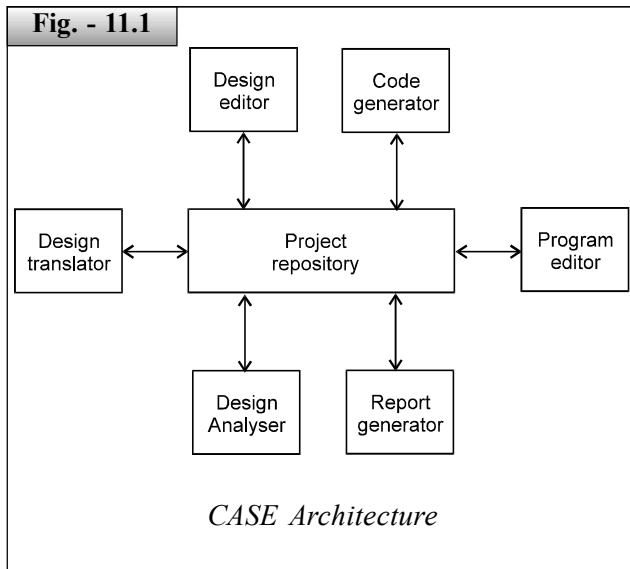
It aids a software engineer to maintain and to develop software. The CASE is a Productive hand-support to the Integrated Project Support Environment (IPSE), the workshop for software engineering.

The CASE is applicable in any software engineering process like automated tools. So it is also referred as a supporting tool to the automated tools.

The CASE tool is applicable in the following software engineering processes such as :

- Translation of users requirements into software requirements (SRS)
- Translation of software requirements into design specification i.e translation of SRS to DDS / SDD.
- Implementation of design into code.
- Testing of the code.
- Documentation.

The CASE tools are also helpful for making automation of the following activities involved in software development like :



- The development of graphical system models as part of the requirements specification or the software design.
- Understanding a design using a data dictionary, which keeps all the relevant information about the entities and their associationship in a design.
- The generation of user interfaces from a graphical interface description, which is created interactively by the customer.
- Program debugging through the provision of information about an executing program.
- The automated translation of programs from an old version of a programming language such as COBOL to a more recent version.

By using the CASE tools, the effort required for software development is reduced and the quality of the developed software is improved there by managing the configuration changes (*configuration control*) throughout the software development life cycle.

The CASE is also helpful to the project manager, the software developer and other key personnel to improvise their productivity in the development team.

11.2 LEVELS OF CASE

The CASE technology is diversified into three different levels w.r.t its utilisation purposes.

1) Production Process Support Technology :

This includes support for process activities such as specification, design, implementation, testing and so on.

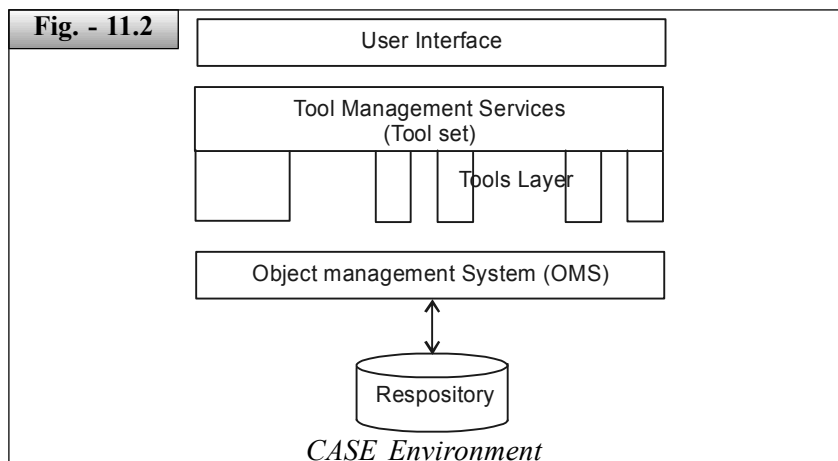
2) Process Management and Technology :

This includes tools to support process modelling, and process management. These tools are used for specific support activities.

3) Meta-CASE Technology :

Meta-CASE tools are generators, which are used to create production process and are management support tools.

11.3 ARCHITECTURE OF CASE ENVIRONMENT



The architectural components of a modern CASE environment are the user Interface, tools management system (Tool set), the object management system (OMS) and the information repository.

These various components are discussed below..

11.3.1 User Interface / Interface Generator

It provides a consistent framework for accessing different tools. System interfaces are the means by which users interact with an application both to input information and data or to receive information.

Interface generator provide the capability to prepare mock-ups and prototypes of user interfaces. Typically they support the rapid creation of demonstration system menus. presentation screens, and report layouts.

Interface Generators are important elements for application prototyping, although they are useful with all development methods.

11.3.2 Tools Management Services (Tools Set)

The tools set section holds the different types of improved quality tools. The tools layer incorporates a set of tools management services with the *CASE* tool themselves. Tools Management Service (*TMS*) control the behaviour of tools within the environment. If multitasking is used during the execution of one or more tools, *TMS* performs multitask synchronization and communication, co-ordinates the flow of information from the repository and object management system into the tools, accomplishes security and auditing functions, and collects metrics on tool usage.

11.3.3 Object Management System (OMS)

The object management system maps these (specification design, text data, project plan etc.) logical entities into the underlying storage management system i.e., repository.

Working in conjunction with the CASE repository, the *OML* provides integration services a set of standard modules that couple tools with the repository. In addition, the *OML* provides configuration management services by enabling the identification of all configuration objects performing version control, and providing support for change control, audits and status accounting.

11.3.4 Repository

The capture, analysis, processing and distribution of all systems information is aided by a centralized information repository or data dictionary. The dictionary contains the details of system components, such as data items, data flows, and processes and also includes information, describing the volume and frequency of each activity.

When dictionaries are designed so that the information is easily accessible, they also include built-in controls and safe guards that preserve the accuracy and consistency of system details.

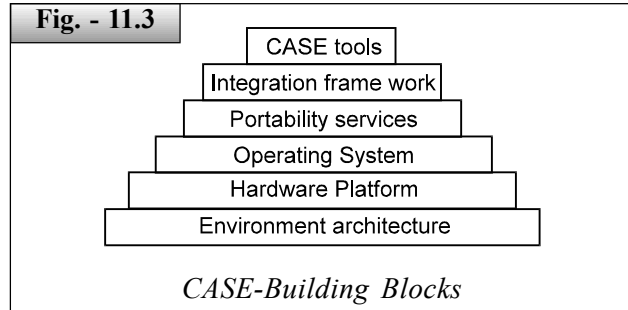
The use of

- 1) authorization levels
- 2) process validation, and
- 3) procedures for testing consistency of descriptions ensures that access to

definitions and the revisions made to them in the information repository occur properly according to prescribed procedures.

11.4 BUILDING BLOCKS FOR CASE

The building blocks for CASE are illustrated in figure 11.3.



a. Environment architecture :

The environment architecture, composed of the Hardware platform and operating system support including networking and database management software, lays the groundwork for *CASE* but the *CASE* environment itself demands other building blocks.

b. Portability Services :

A set of portability services provides a bridge between *CASE* tools and their integration framework and the environment architecture.

- These portability services allow the *CASE* tools and their integration framework to migrate across different hardware platforms and operating systems without significant adaptive maintenance.

c. Integration framework :

It is a collection of specialised programs that enables individual *CASE* tools to communicate with one another, to create a project database.

d. CASE tools :

CASE tools are used to assist software engineering activities (*like analysis, modeling, code generation etc.*) either communicating with other tools, project database or as point solution.

11.5 CASE SUPPORT IN SOFTWARE LIFE CYCLE

There are various types of supports that *CASE* provides during the different phases of a software life cycle.

1. Prototyping support :

The prototyping is used for understanding the requirements for complex software products to market new ideas

- The prototyping *CASE* tools requirements are as follows :
 - * Define user interaction
 - * Define the system control flow
 - * Store and retrieve data required by the system.

- Incorporate some processing logic.
Few features, which are supported by prototyping tools, are
- Integrate well with that data dictionary of a *CASE* environment.
- Main use of prototyping *CASE* tools is developing Graphical User Interface (*GUI*) development.
- It should be able to integrate with the external user-defined modules written in high level languages.
- The user should be able to define the sequence of states through which a created prototype can run.
- The prototype should support mock up run of the actual system and management of the input and output data.

1. **Structured Analysis and Design**

A *CASE* tool should support one or more of the structured analysis and design techniques. It should also support making of the fairly complex diagrams and preferably through a hierarchy of levels. The tool must also check the incompleteness, inconsistencies and anomalies across the design and analysis through all levels of analysis hierarchy.

- Analysis and design tools enable a software engineer to create models of the system to be built. The models contain a representation of data, function, and behaviour, and characterizations of data, architectural, component level, and interface design.
- By performing consistency and validity checking on the models, analysis and design tools provide a software engineer with some degree of insight into the analysis representation and help to eliminate errors before they propagate into the design, or worse, into implementation itself.

3. **Code Generation :**

- A support expected from a *CASE* tool during the code generation phase comprise of the following.
- The *CASE* tool should support generation of module skeletons or templates in one or more popular programming languages.
- The tool should generate records, structures, class definitions etc. automatically from the contents of the data dictionary in one or more popular programming languages.
- It should be able to generate database tables for relational database management system.
- The tools should generate code for user interface from prototype definitions for X-windows and MS-window base applications.

4. **Test CASE Generator :**

- The *CASE* tools for test case generator should have following features.
- It should support both design and requirement testing.
- It should generate test set reports in *ASCII* format which can be directly imported into the test plan document.

- Under testing phase, test management tools are used to control and coordinate software testing for each of the major testing steps. Testing tools manage and coordinate regression testing, perform comparisons that ascertain differences between actual and expected output, and conduct batch testing of programs with interactive human computer interface. In addition to the functions noted, many test management tools also serve as genuine test drives. A test driver reads one or more test cases from a testing file, formats the test data to conform to the needs of the software under test.

11.6 OBJECTIVES OF CASE

Computer Aided Software Engineering bears following objectives.

1. Improve Productivity

Most organizations use *CASE* to increase the speed with which systems are designed and developed. Imagine the difficulties, the carpenters would face without hammers and saws. Tools increase the analysts productivity by reducing the time needed to document, analyze, and construct information system.

2. Improve Information System Quality

- Inner tools improve processes, and usually they improve the results as well.
- Ease and improve the testing process through the use of automated checking.
- Improve the integration of development activities via common methodologies.
- Improve the quality and completeness of documentation.
- Help standardize the development process.
- Improve the management of the project.
- Simplify program maintenance.
- Promote reversibility of modules and documentation.
- Shortens the overall construction process.
- Improve software portability across environments.
- Through reverse engineering and re-engineering, *CASE* products extend the file of existing systems. Despite the various driving forces (*objectives*) for the adoption of *CASE*, there are many resisting forces also that preclude many organizations from making investment in *CASE*.

3. Improve Effectiveness

Effectiveness means doing the right task. Tools can suggest procedures to approach a task. Identifying user requirements, stating them in an understandable form, and communicating them to all interested parties can be an effective development process compared to moving quickly into coding.

4. Organizations Reject CASE Because

- The start-up cost of purchasing and using *CASE*.
- The high cost of training personnel.
- The big benefits of using *CASE* come in the later stages of the *SDLC*.

- *CASE* often lengthens the duration of early stage of the project.
- *CASE* tools can not easily share information between tools.
- Lack of methodology standards within organizations, *CASE* products forces analysts to follow a specific methodology for system development.
- Lack of confidence in *CASE* products.
- In personnel view, *CASE* as a threat to their job security.

11.7 CASE REPOSITORY

A *CASE* repository is a system developer database. Synonyms include dictionary and encyclopedia. It is a place where developers can store system models, detailed descriptions and specifications and other products of system development.

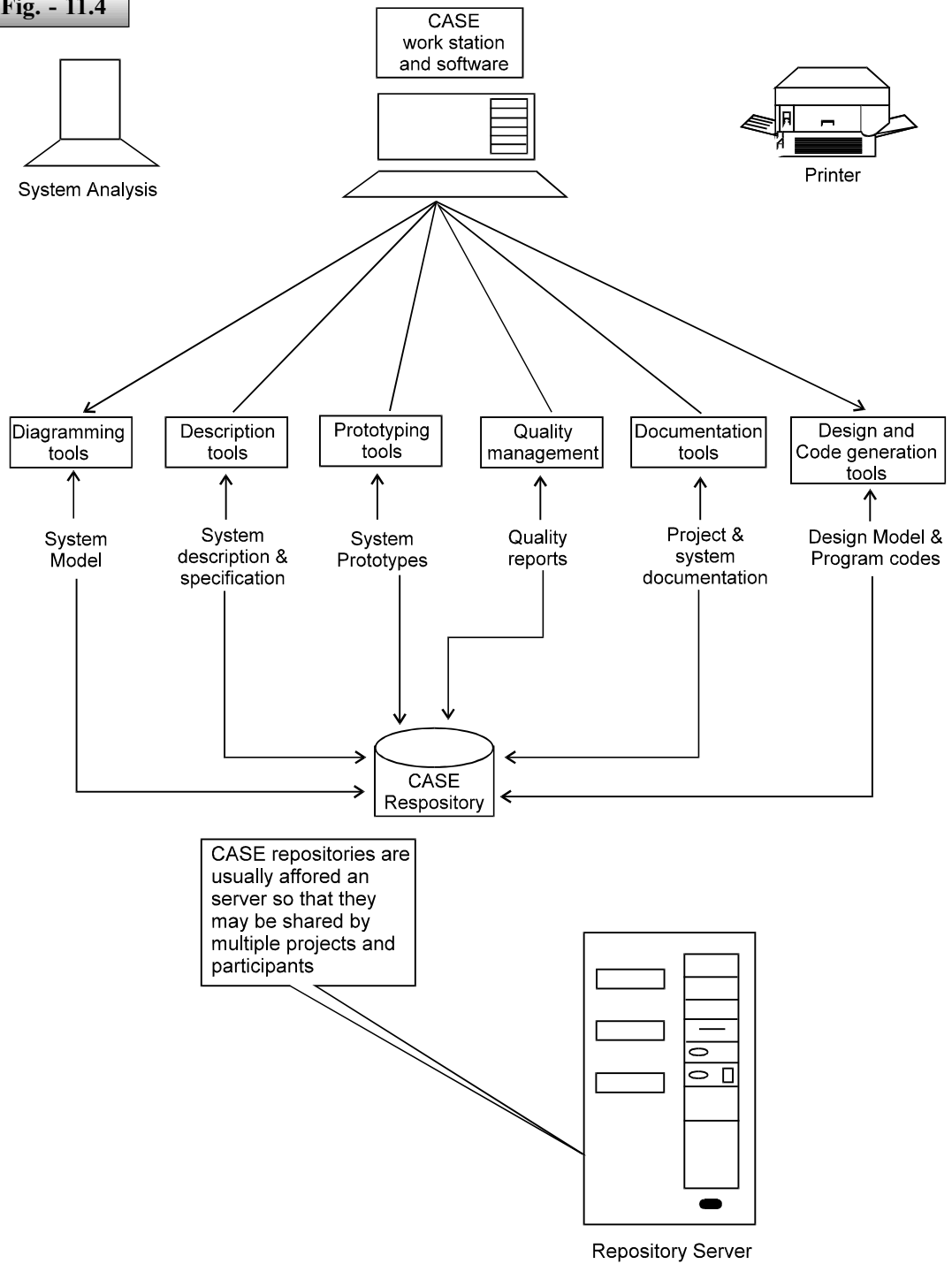
Analysts use *CASE* repositories for five important reasons

- To manage the details in large systems.
- To communicate a common meaning for all system elements.
- To document the features of the system.
- To facilitate analysis of the details in order to evaluate characteristics and determine where system changes should be made.
- To locate errors and omissions in the system.
- To limit the amount of narrative needed to describe relationships between data items and at the same time to show the structural relationship clearly, analysts after use formal notation in data dictionary component of *CASE* repository.

Data dictionary can be developed manually or using automated systems. Automated systems offer the advantage of automatically producing data element, data structure, and process listings; they also perform cross-reference checking and error detection. The data dictionary is a repository of all data definitions for all organizational applications and is used to manage and control access to the information repository, another component of *CASE* repository. Information repository provides automated tools used to manage and control access to business information and application portfolio.

CASE repository is an idea central to *I-CASE*. Integrated-*CASE* tools rely on common terminology, notations and methods for systems development across all tools. Within an *I-CASE* environment, all diagrams, forms, reports and programs can be automatically updated by the single change to the data dictionary definition. Besides specific tools integration, there are two additional advantages of using a comprehensive *CASE* repository that relates to project management and software reusability. The *CASE* repository provides sufficient information to the project manager for better controlling and estimation of the software project to be developed.

Fig. - 11.4

*CASE Repository*

11.8 CHARACTERISTICS OF CASE TOOLS

- The *CASE* tools have following characteristics.
- A graphic interface to draw diagrams, charts, models, (*upper case, lower case*)
- An information repository, a data dictionary for efficient information management selection, usage, application and storage.
- The common user interface for integration of multiple tools used in various phases.
- Automatic code generators.
- Automatic testing tools.

11.9 CLASSIFICATION OF CASE TOOLS

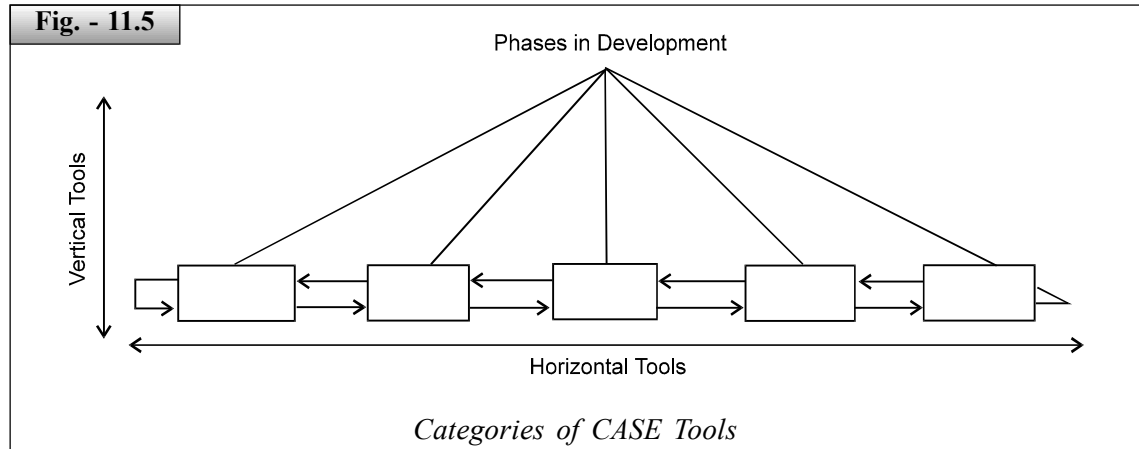
- *CASE* classification helps us to understand various types of *CASE* tools along with their utility in supporting software process activities.
- A *CASE* tool is classified as follows :
 - 1) Functional prospectives : where *CASE* tools are classified according to their functionality.
 - 2) Process prospectives : where *CASE* tools are classified according to their processing activities.
 - 3) Integration prospectives : Where *CASE* tools are classified according to how they are organised into integrated units and provide supports for one or more process activities.

Table-11.1 : List of CASE Tools

Application	CASE Tool	Purpose of Tool
1. Planning	Excel-spread sheet, MS-Project, PERT/CPM Network, Estimation tools.	Functional Application : Planning, scheduling and control.
2. Editing	Diagram editors, Test editors, word Processing.	Speed and Efficiency.
3. Testing	Test data Generators, file comparators.	Speed and Efficiency.
4. Prototyping	High level Modeling language, User-interface Generators	Confirmation and Certification of RDD and SRS.
5. Documentation	Report Generators, Publishing imaging, PPT presentation.	Faster structural documentation with quality of presentation.
6. Programming and Language processing integration.	Program Generators, Code Generators, Compilers, interpreters, interfaces and connectivity.	Programming of high quality with no errors, system integration.
7. Templates	--	Guided systematic development.
8. Re-engineering tools	Cross reference system program, re-structuring systems.	Reverse engineering to find structure, design and design information.
9. Program analysis tools.	Cross reference generators static analyzers, dynamic analyzers.	Analyses risks, functions and features.

11.10 CATEGORIES OF CASE TOOLS

The schematic of CASE tools is shown in fig.11.5



Smith and Oman have defined *CASE* tools which are divided into the following two categories.

1. Vertical *CASE* tools.
2. Horizontal *CASE* tools.

1. Vertical CASE Tools

Vertical *CASE* tools provide support for certain activities within a single phase of the software life cycle.

There are two sub categories of vertical *CASE* tools.

- **First category :** It is the set of tools that are within one phase of life cycle. These tools are important so that development in each phase can be as quick as possible.
- **Second category :** It is a tools that is used in more on next phase, but does not support moving from one phase to the next.

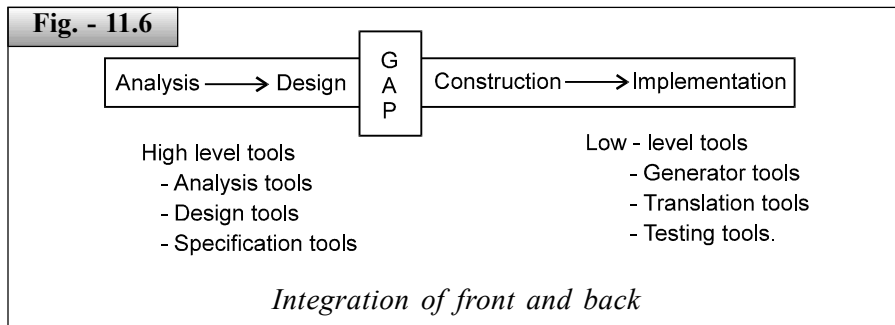
2. Horizontal CASE Tools

These tools support automated transfer of information between the phase of a life cycle. These tools include project management, configuration management tools and integration services.

The above two categories of *CASE* tools can further be broken down into the following.

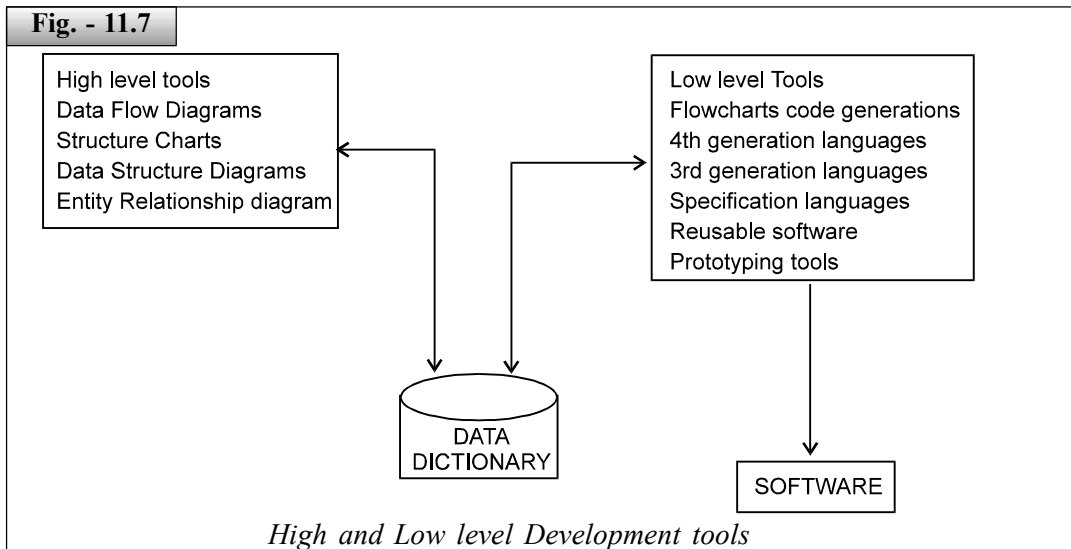
(a) Upper CASE Tools / Front - end CASE Tools

CASE tools designed to support the analysis and design phases of *SDLC*. All the analysis, design and specification tools are front-end tools. These tools also include computer-aided diagramming tools oriented towards a particular programming design methodology, more recently including object oriented design.



The general types of upper *CASE* tools are listed below :

- **Diagramming tools** : Diagramming tools enable system process, data and control structures to be represented graphically. They strongly support analysis and documentation of application requirements.
- **Form and Report Generator tools** : They support the creation of system forms and reports in order to show how systems will “look and feel” to users.
- **Analysis tools** : These tools enable automatic checking for incomplete, inconsistent, or incorrect specifications in diagrams, forms and reports.



(b) Lower CASE or Back-end CASE tools

CASE tools designed to support the implementation and maintenance phases of *SDLC*. All the generator, translation and testing tools are back-end tools.

The general types of Lower *CASE* tools are :

- **Code Generators** : Code generators automate the preparation of computer software. Code generation is not yet perfect. Thus, the best generator will produce approximately 75 percent of the source code for an application. Hand coding is still necessary.

(c) **Cross life cycle *CASE* or Integrated tools :**

CASE tool is applicable in each phase of SDLC. As *CASE* tool includes front-end and back-end development features, hence it provides an efficient environment for the creation, storage, manipulation and documentation of systems.

(d) **Reverse Engineering Tools :**

These tools help in the process of analyzing the existing application, performance, database source code for higher level representation to prepare new source code for desired system.

11.11 ADVANTAGES OF CASE TOOLS

By using the *CASE* tools for analysis, design, development, testing and implementation of a software system, the following merits are obtained.

- Improvisation of productivity
- Simplified documentation
- Higher degree of quality management
- Decreasing in maintenance through out the lifetime of software.
- The software to be developed must be cost effective.
- High quality documentation (*SRS / SDD / DDS*)
- Improved co-ordination among staff members who are working on a large software project.
- An increase in project control through better planning, monitoring and communication.

11.12 DISADVANTAGES OF CASE TOOLS

Computer Aided Software Engineering (*CASE*) tools has following disadvantages.

1) **Procurement of *CASE* tools is a tedious task**

The *CASE* tools is of very high cost. Hence, small software development farms don't dare to invest the capital in *CASE* tools.

2) **Learning hell**

In most of the cases, the programmer productivity gradually decreases in the initial phase of implementation. So a qualitative training program about the new technology is badly required to provide for the user or programmer.

3) **Mixture of tools**

As *CASE* tool contains a number of tools in it, hence while selecting the proper tools for application, the system analyst get puzzled to select the proper one.

SUMMARY

- In this chapter some of the important features of the present-day CASE tools and also discussed the emerging trends has been highlighted.
- Use of CASE tools is becoming almost indispensable for large software projects where a team of software engineers work together. The trend is now towards distributed workstation-based CASE tools.
- Also here some of the desirable features of the distributed workstation-based CASE tools has been pointed out.

EXERCISES

1. Explain Computer Aided Software Engineering (CASE) and various types of CASE tools?
2. What is CASE? Explain three most used CASE tools?
3. What are the advantages and disadvantages of CASE tools?
4. What do you mean by CASE? What are the different categories of CASE tools?
5. Explain how CASE support in software life cycle?
6. Explain the Building Blocks of CASE?
7. What are the important characteristics of CASE tools?
8. What is CASE? What are the main advantages of using CASE tools? Describe some of the important features that a future generation CASE tool should support?
9. Write a short note on CASE tools? Also state its benefits?
10. What do you understand by the terms: a CASE tool and a CASE environment? Why do integration tools increase the power of tools? Explain using some examples.
11. Give the main advantages of using CASE tools?
12. What is CASE? In case a large complex software solution is required, why is the 'Case' approach recommended? Comment how 'Case' approach affects the following:
(i) Requirement (ii) Risk (iii) Report Generation
(iv) Documentation (v) Programming Effort
13. Draw the schematic diagram of the architecture of a CASE environment and explain how the different tools are integrated?
14. Suggest potential benefits and practical problems of integrating CASE tools?
15. Give architecture of a CASE environment. How does CASE assist in software life cycle?
16. Define Re-engineering?



Unified Modeling Language

12.1 INTRODUCTION

In the year of 1997 to be exact — the Object Management Group (OMG) released the Unified Modeling Language (UML). One of the purposes of UML was to provide the development community with a stable and common design language that could be used to develop and build computer applications. UML brought forth a unified standard modeling notation that IT professionals had been wanting for years. Using UML, IT professionals could now read and disseminate system structure and design plans — just as construction workers have been doing for years with blueprints of buildings.

It is now the twenty-first century and UML has gained traction in software profession. On 75 percent of the resumes it is observed there is a bullet point claiming knowledge of UML. However, it becomes clear that they do not truly know UML. Typically, they are either using it as a buzz word, or they have had a sliver of exposure to UML. This lack of understanding inspired me to write this quick introduction to UML, focused on the basic diagrams used in visual modeling. When you are finished reading you will not have enough knowledge to put UML on your resume, but you will have a starting point for digging more deeply into the language.

UML was meant to be a unifying language enabling IT professionals to model computer applications. The primary authors were Jim Rumbaugh, Ivar Jacobson, and Grady-Booch, who originally had their own competing methods (OMT, OOSE, and Booch). Eventually, they joined forces and brought about an open standard. (Sound familiar? A similar phenomenon spawned J2EE, SOAP, and Linux.) One reason UML has become a standard modeling language is that it is programming-language independent. (UML modeling tools from IBM Rational are used extensively in J2EE shops as well in .NET shops.) Also, the UML notation set is a language and not a methodology. This is important, because a language, as opposed to a methodology, can easily fit into any company's way of conducting business without requiring change.

Since UML is not a methodology, it does not require any formal work products. Yet it does provide several types of diagrams that, when used within a given methodology, increase the ease of understanding an application under development. There is more to

UML than these diagrams, but the diagrams offer a good introduction to the language and the principles behind its use. By placing standard UML diagrams in our methodology's work products, it is easier for UML-proficient people to join any project and quickly become productive. The most useful, standard UML diagrams are: use case diagram, class diagram, sequence diagram, state-chart diagram, activity diagram, component diagram, and deployment diagram.

It is beyond the scope of this introductory article to go into great detail about each type of diagram. Instead.

12.2 MODEL

A model provides blueprint of the system. It is an abstract representation of a system. This model representation promotes better understanding. The advantages of modeling are summarized by Turban.

- i) Helps in visualising the system to be developed.
- ii) Permits to specify the structure and behaviour of the system.
- iii) Used as template to construct a proposed system.
- iv) Helps to master the complex system.
- v) Easy to capture and precise the requirements by all stakeholders.
- vi) Used to document the decision made.
- vii) Provides guidelines to explore multiple solutions economically.
- viii) Makes easy to enhance and manipulation of existing system.

It takes different form for various purposes and emerges at different levels of abstraction.

- i) Strategic / High-level focus on thought process and highlight the options.
- ii) Implementation level includes data structures, mechanisms, algorithms etc.,
- iii) Preliminary design stages focus on key concepts and mechanisms.

A model is good when it is possible to communicate, and when it fits its purpose. A model either can be simple mathematical, theoretical or visual. Visual modeling is defined as mapping the real world process of a computer system with a graphical representation. It captures essential parts of the system. Also a model can be classified into two : Static and dynamic.

Static model is viewed as an image of systems parameters at rest or a specific point in time. But dynamic model is viewed as a collection of procedures or behaviors that, taken together, reflect the behaviour of a system over a period of time.

12.3 THE UML

Unified Modeling language (UML) is a generic syntax for creating a logical model of a system. This is the unified model evolved by Graddy Booch, Rumbaugh and Jacobson. It combines all the best features of OOD, OMT and OOSE methodology and provides a set of diagrams. It is a complete solution provider of the business to answer all the queries (see Table 12.1). It insists the standardization of the notations used in various model.

Table -12.1 : *UML is a complete solution provider of the Business.*

	OOSE	OMT	BOOCH	UML TOOL
User and user actions ?	Yes	---	---	Use cases
Real world objects ?	--	Yes	---	High level class Diagrams
Objects for each use case ?	Yes	---	---	Robustness Analysis
Object Interactions ?	Yes	---	Yes	Sequence and collaboration Diagrams
Real time control Issues ?	Yes	Yes	Yes	State Diagrams
How to build ?	---	---	Yes	Low level class Diagrams.

UML is a graphical language for visualizing, specifying, constructing and documenting the artifacts of a software intensive system. UML combines the best of the best from -

- i) Data modeling concepts (E – R diagrams)
- ii) Business Modeling (work flow)
- iii) Object Modeling
- iv) Component Modeling

UML facilitates all the necessary elements to represent the system in whole. UML is viewed in several forms. It can be viewed as language, visual tool, specification, construction tool, and documentation:

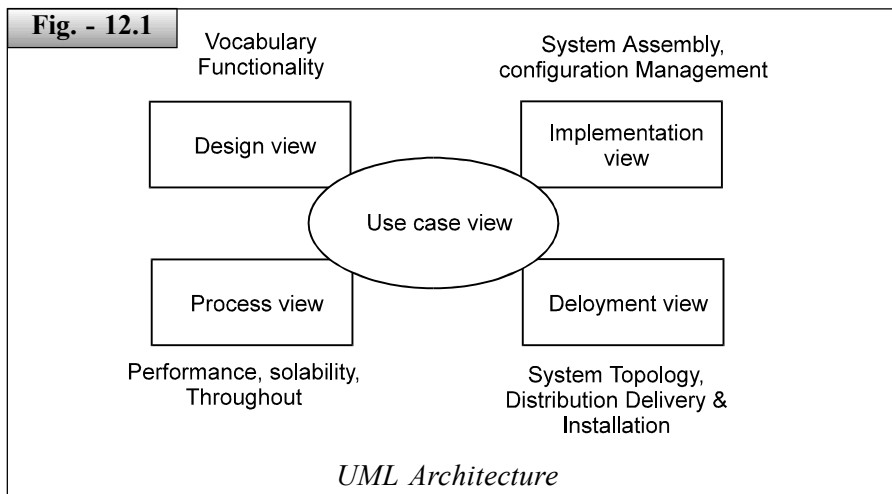
- i) UML is a language: UML provides a vocabulary and the rules for communication and focus on conceptual and physical representation of the system. So it is a Modeling Language. .
- ii) UML Visualizes: The UML includes both graphical and textual representation. It makes easy to visualize the system and for better understanding.
- iii) UML Specifies: UML addresses the specification of all the important analysis, design, and implementation decisions to develop and deploy a software-intensive system.
- (iv) UML Constructs: UML models can be directly connected to a variety of programming languages. It is sufficiently expressive and free from any ambiguity to permit the direct execution of models, the simulation of systems, and the instrumentation of running systems.
- (v) UML Documents: UML produces variety of documents, in addition to raw executable code; the artifacts include Requirements, Architecture, Design, Source code, Project Plans, Test, Prototypes, and Releases.

UML has been effectively used in several domains such as: EIS [*Enterprise Information Systems*], Banking and financial services, Telecommunications, Transportation, Defense/

Aerospace, Retail, Medical Electronics, Scientific and Distributed Web-based services. It can be used with all processes, throughout the development life cycle, and across different implementation technologies. Some of the known users of UML are: MCI systems, Republic Bank, Oracle, HP, Microsoft etc.,. The primary goals of UML are to:

- (i) Be an expressive, visual modeling language that is relatively simple and extensible.
- (ii) Have extensibility and specialization mechanisms for extending the core concepts
- (iii) Be independent of any programming language and development processes
- (iv) Provide the required formal basics for understanding the modeling Language
- (v) Support high-level concepts (frameworks, patterns and components)
- (vi) Address recurring architectural complex issues using high-level concepts
- (vii) Be scalable and widely applicable over many domains
- (viii) Encourage the growth of the OO tools market
- (ix) Integrate best practices and methodologies.

12.4 UML ARCHITECTURE



UML architecture is used to manage different viewpoints and hence control the iterative and incremental development of systems through out its life cycle. Architecture is the set of significant decisions about :

- i) The organization of a software system.
- ii) Selection of structural elements and their interfaces.
- iii) Behaviour in the collaborations.

Each of the view of UML can stand-alone so that different stakeholders can focus on the issues of the systems architecture that most concerns them.

Table 12.2 : Views and Responsibilities

Views	Responsibilities
Use case view	- Describe behaviour of the system by its and users, analysts and testers. - Shapes the system architecture - Static aspects are captured in use case diagram. - Dynamic aspects are captures in interaction and activity diagrams.
Design view	- Describe classes, interfaces and collaborations that form vocabulary. - Supports functional requirements. - Static aspects are captured in class and object diagrams. - Dynamic aspects are captured in interaction, statechart and activity diagrams.
Process View	- Describe threads and processes that form system's concurrency and synchronization mechanisms. - Static and dynamic aspects are captured as same as design view with active classes.
Implementation view	- Describes the components the files that are used to assemble. - Static aspects are captured in component diagrams - Dynamic aspects are captured in interaction, state-chart and activity diagrams.
Deployment view	- Encompasses the nodes that form the system's hardware topology on which system executes. - Static aspects are captured in deployment diagrams. - Dynamic aspects are captured in interaction, statechart & activity diagrams.

12.5 UML FOUNDATIONS

The vocabulary of the UML encompasses three kinds of building blocks such as Things, Relationships and diagrams.

(A) Things :

There are four kinds of things in the UML. They are structural things. Behavioral things, grouping things and Annotational things.

- i) *Structural things* are nouns and static parts of Model. The seven structural things are class, Interface, Collaboration, use case, Active class, Component and Node.
- ii) *Behavioral things* are verbs and dynamic parts of UML, representing behaviour over time and space. The two behavioral things are interaction and static machine.
- iii) *Grouping things* are the organisational parts of UML. It is a package where structural, behavioural and even other grouping things are grouped.
- iv) *Annotational things* are explanatory parts of models. These are the comments applied to describe, illuminate and remark about any element in a model.

(B) Relationship :

Relationship is a semantic connection among elements. The different kinds relationships in the UML are Dependency, Association (Aggregation), Generalisation and Realization.

(C) Diagram :

Every complex system is best approached through a small set of nearly independent views of a model no single view is sufficient.

Table 12.3 : The UML Diagrams

Static / Structural Diagrams	Dynamic / Behavior Diagrams
- Class Diagram - Object Diagram - Implementation Diagrams - o Component Diagram - o Deployment Diagram	• Use Case Diagram • Interaction Diagrams - o Sequence Diagram - o Collaboration Diagram • State chart Diagram • Activity Diagram

Table 12.4 : Diagrams and their common uses

DIAGRAMS	COMMON USES
Class diagrams	- Model the vocabulary of the system - Model simple collaborations - Model a logical database schema
Object diagrams	• Model Object Structures • Primary support of the functional requirement the system
Use Case diagrams	• Model the context of the system • Model requirements of the system
Interaction diagrams	
Sequence diagrams	• Model interaction among object in time sequence
Collaboration diagrams	• Model flow of control by organization
Statechart diagrams	• Model dynamic aspect event-ordered behavior single Use case • Model reactive objects
Activity diagrams	• Model a work flow of a single use case or for the entire software • Model an operation
Implementation diagrams	
Component diagrams	• Model static implementation view • Model source code • Model executable releases • Model physical databases • Model adaptable systems
Deployment diagrams	• Model embedded system • Model client server system • Model fully distributed system

12.6 RULES OF THE UML

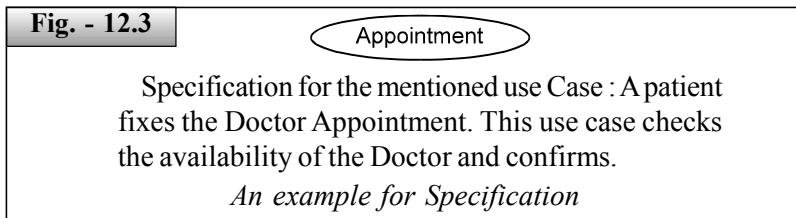
Like any language, UML has a set of rules that specify a well - formed model. The UML has sematic rules for.

- i) Names : Given for UML Foundations (Things, Relationships and Diagrams)
- ii) Scope : The context that gives specific meaning to a name.
- iii) Visibility : Now those names can be send and used by others.
- iv) Integrity : How things properly and consistently relate to one another.
- v) Execution : What it means to run or simulate a dynamic model ?

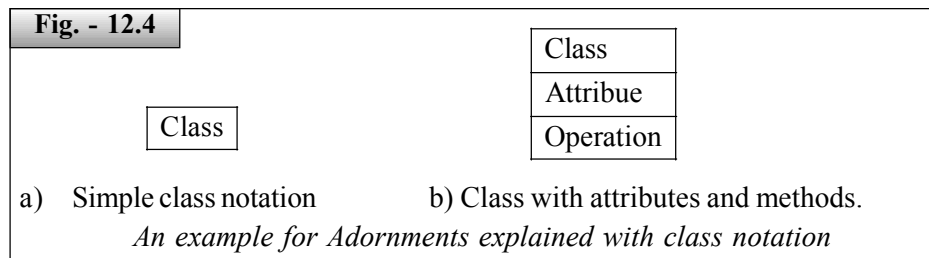
12.7 COMMON MECHANISMS IN UML

The UML is made simple by the presences of four common mechanisms that apply consistently throughout the language. These mechanisms define styles, certain architectural patterns and more harmonious by the conformance to a common features.

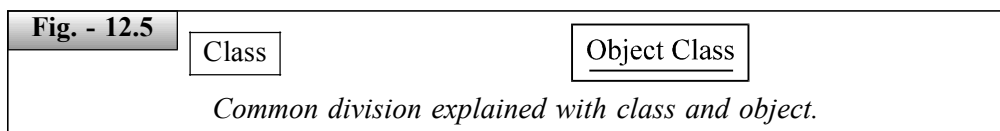
i) Specification : It provides the statement of the syntax and semantics of building blocks. It is used to provide the detailed description for the models and their notations in the system.



ii) Adornments : Almost all the elements in UML have a unique, direct graphical notation. They are intentionally designed to be easy to draw. The other details of the particular element can be added easily.



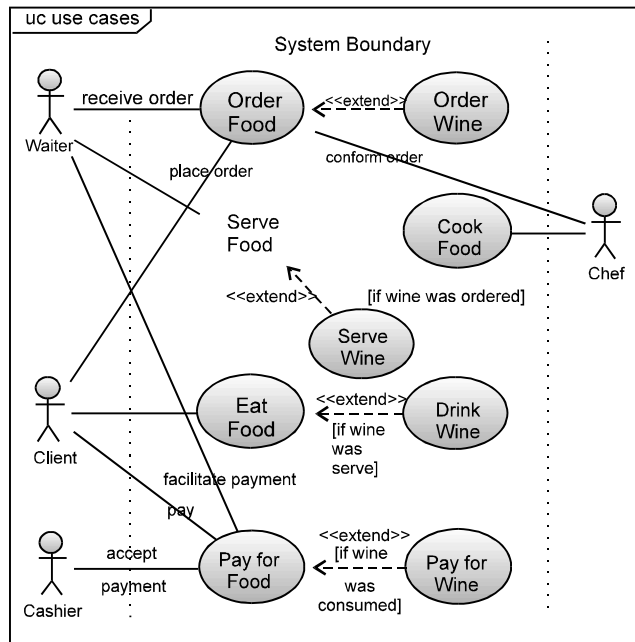
iii) Common Divisions : Object oriented systems after get divided in several ways. The same notations can be used in other, which is coupled with the first one. For example, an object is instance of class.



iv) Extensibility Mechanisms : The UML is open-ended, making it possible to extend the language in controlled ways to express the nuances of all models across all domains.

12.8 USE CASE DIAGRAM

Fig. - 12.6

*UML Use Cases for a Simple Restaurant Model*

A **use case diagram** in the **Unified Modeling Language (UML)** is a type of behavioral diagram defined by and created from a **Use-case analysis**. Its purpose is to present a graphical overview of the functionality provided by a system in terms of **actors** (human beings who will interact with the system), their goals (represented as **use cases**), and any dependencies (relationship) between those use cases.

Use-case diagrams generally highlights, groups of use-case-either all use cases for a complete system, or a breakout of a particular group of use cases with related functionality.

The main purpose of a use case diagram is to show what system functions are performed for which actor. Roles of the actors in the system can be depicted.

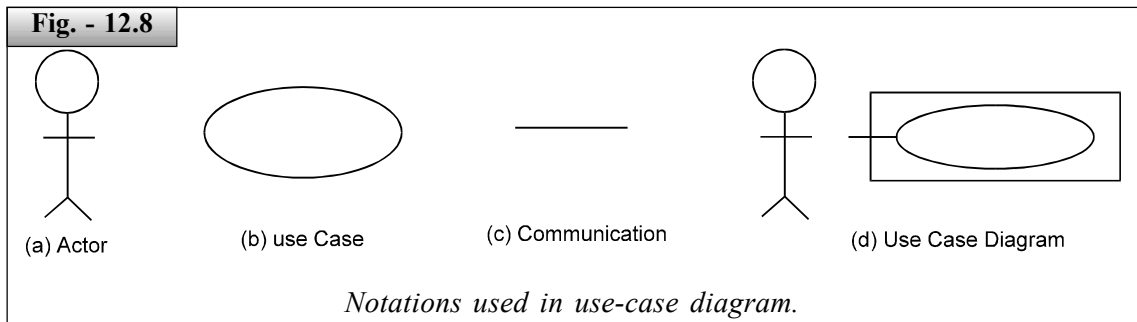
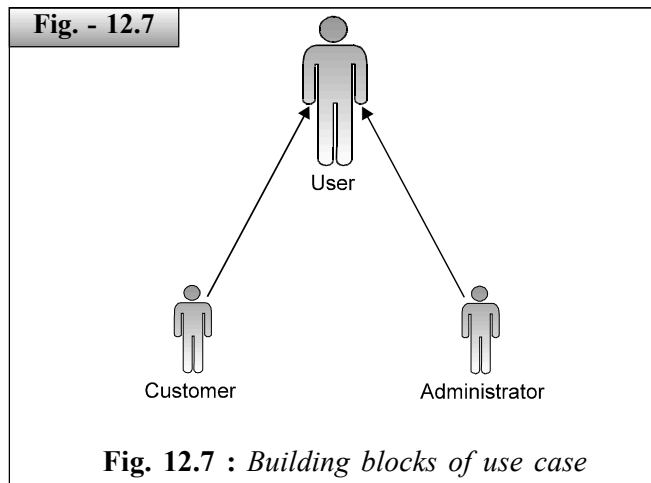
Overview

Use Case diagrams are formally included in two modeling languages defined by the **OMG**. Both the **UML** and **SysML** standards define a graphical notation for modeling **use cases** with diagrams. One complaint about the standards has been that they do not define a format for describing these use cases. Generally, both graphical notation and descriptions are important as they document the use case, showing the purpose for which an **actor** uses a system.

The use case diagram shows the position or context of the use case among other use cases. As an organizing mechanism, a set of consistent, coherent use cases promotes a useful picture of system behavior, a common understanding between the customer/owner/user and the development team.

This diagram describes the functionality of a simple *Restaurant System*. **Use cases** are represented by ovals and the **actors** are represented by **stick figures**. The Patron actor can Order Food (and optionally Order Wine), Eat Food (and optionally Drink Wine), Pay for Food. Multiple actors participate in some of the use cases. In such cases, the label on the association between the actor and the use case throws some light on the actor's contribution to the use case. The box defines the boundaries of the Restaurant System, i.e., the use cases shown are part of the system being modeled, the actors are not.

Diagram building blocks



Use case relationships

Interaction among actors is not shown on the use case diagram. If this interaction is essential to a coherent description of the desired behavior, perhaps the system or use case boundaries should be re-examined. Alternatively, interaction among actors can be part of the assumptions used in the use case.

● Actor Generalization

One popular relationship between Actors is *Generalization/Specialization*. This is useful in defining overlapping roles between actors. The notation is a solid line ending in a hollow triangle drawn from the specialized to the more general actor. Use Case Relationships

Three relationships among use cases are used often in practice. Include

In one form of interaction, a given use case may *include* another. “Include is a Directed Relationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case”

The first use case often depends on the outcome of the included use case. This is useful for extracting truly common behaviors from multiple use cases into a single description. The notation is a dashed arrow from the including to the included use case, with the label “«include»”. This usage resembles a macro expansion where the included use case behavior is placed inline in the base use case behavior. There are no parameters or return values. To specify the location in a flow of events in which the base use case includes the behavior of another, you simply write *include* followed by the name of use case you want to include, as in the following flow for *track order*.

- **Extend**

In another form of interaction, a given use case (the extension) may *extend* another. This relationship indicates that the behavior of the extension use case may be inserted in the extended use case under some conditions. The notation is a dashed arrow from the extension to the extended use case, with the label “«extend»”. Notes or constraints may be associated with this relationship to illustrate the conditions under which this behaviour will be executed.

Modelers use the «extend» relationship to indicate use cases that are “optional” to the base use case. Depending on the modeler’s approach “optional” may mean “potentially not executed with the base use case” or it may mean “not required to achieve the base use case goal.”

- **Generalization**

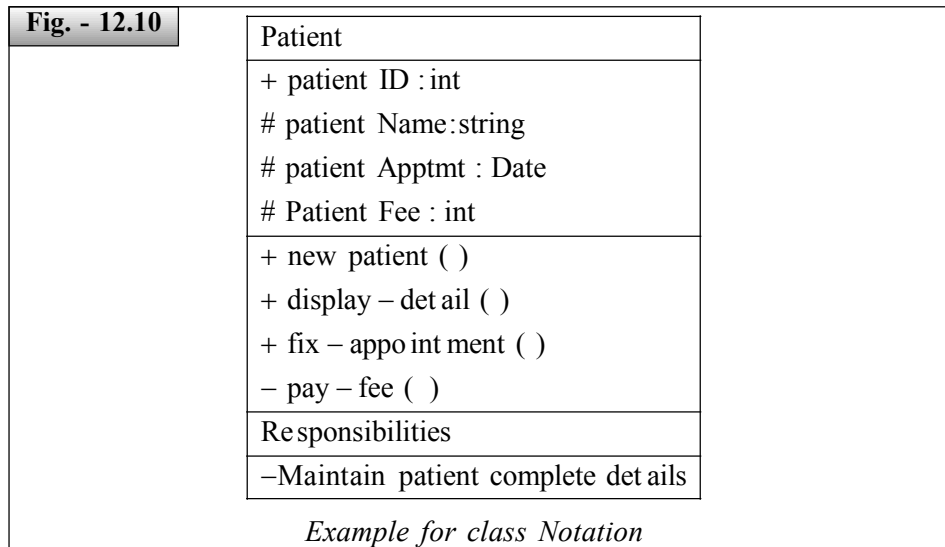
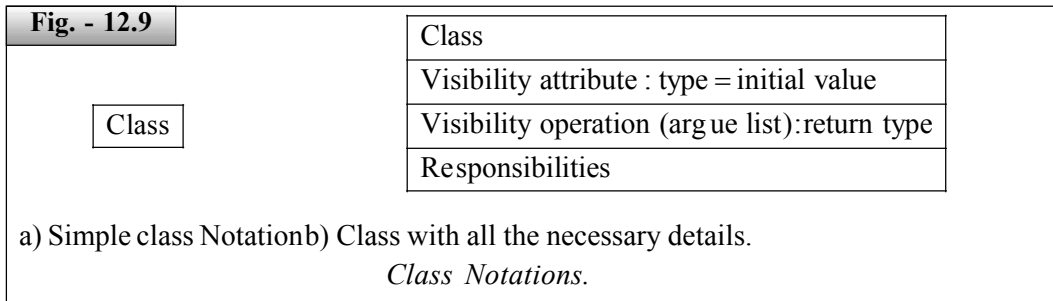
In the third form of relationship among use cases, a *generalization/specialization* relationship exists. A given use case may have common behaviors, constraints and assumptions to the general use case, describe them once, and deal with it in the same way, except for the details in the specialised cases. The notation is a solid line ending in a hollow triangle drawn from the specialized to the more general use case.

12.9 CLASS DIAGRAM

A class is a description of a set of objects. Default values can be assigned. A responsibility is a contract or an obligation of a class. Visibility specifies whether other classifiers can use attributes and operations defined for a particular class. The characters represent the visibility level symbolically +, –, and # which correspond respectively to the levels public : visible to all, private : only to the classifier, and protected : visible to descendent of the classifier. Naming of the classes to be strictly followed :

- i) The class name must be singular.
- ii) The class should describe the singular object and hence it should be a singular form of nouns.
- iii) The users / clients must feel comfort with the name.
- iv) The standard and existing vocabulary must be followed.
- v) The name of the class should reflect its inherent nature.

- vi) Capitalize the first letter of the class.
- vii) Avoid using prefix and suffix. If necessary, make consistent and easily readable.

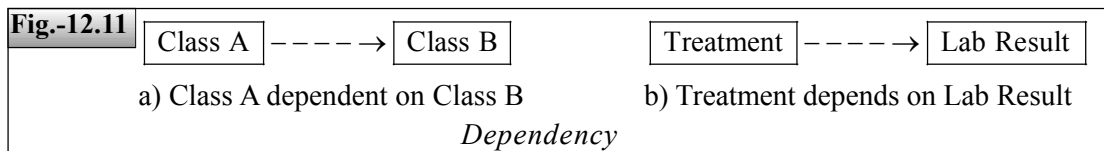


12.9.1 Relationship in class Diagram

Relationship provides a pathway for communication among objects. There are four kinds of relationships depending (uses), Generalization, Aggregation and association.

(A) Dependency :

Dependency is a unidirectional usage relationship between elements. Therefore change in specification of one class affect another class that uses it. The ‘treatment’ class depends on the ‘Lab Result’ class obtained.



(B) Class Hierarchies

Class Hierarchies are used to manage complexity by ordering of objects within trees of classes, with increasing levels of abstraction. The main advantage of class Hierarchy is that new classes can be built on the already existing ones, and the same can be reused.

Generalization is described as aon “is-a-kind-of” relationship. Specialization allows the capture of the specific features of a set of objects that have not been distinguished by the classes already defined. Generalization and specialization are two opposite viewpoints of concept of classification.

Fig. - 12.12

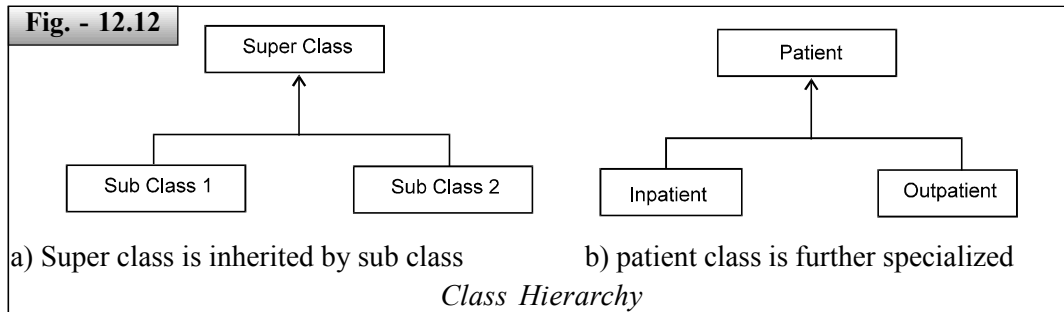
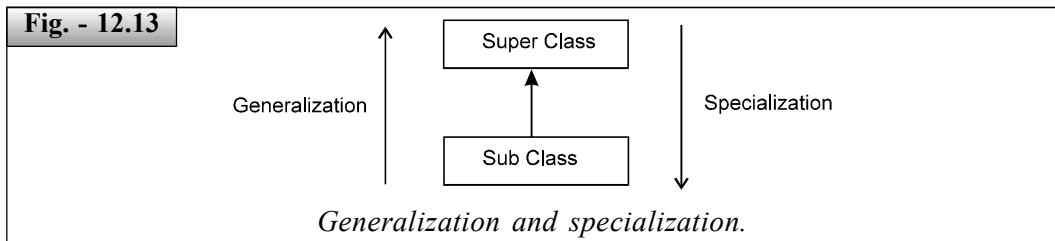
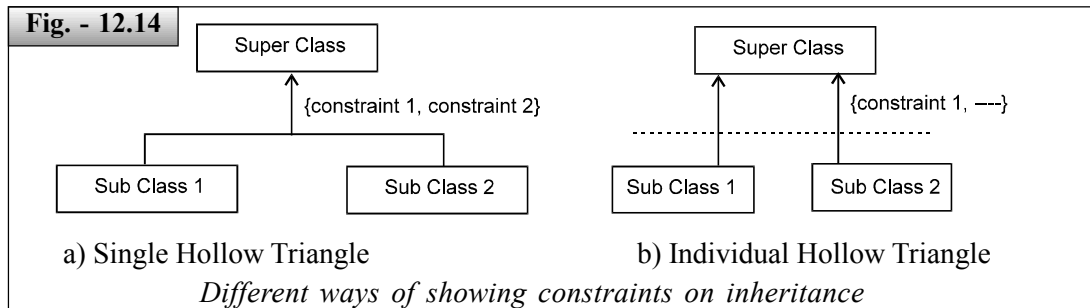


Fig. - 12.13



A constraint given on generalization are overlapping, Disjoint, Complete and incomplete. These are semantic constraints, and may be shown within { }, near the shared hollow triangle if several paths share a single hollow triangle. When more than one hollow triangle connected with super class, a dashed line should cross through all inheritant lines, and the constraints should be inserted in { } near the dashed line.

Fig. - 12.14



C) Association :

Association represents the structural relationships between class. They are bi-directional. They can be traversed in both directions, with different conditions. The direction implied by the name is the forward direction; the opposite direction is the inverse direction. There are several kinds of association exist : Qualified Association, OR Association, N-Ary Association, Association class.

Fig. - 12.15

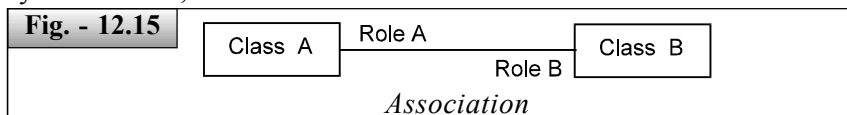
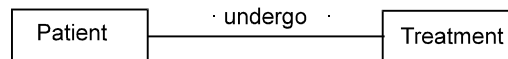


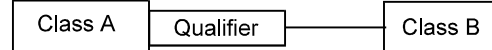
Table-12.5 : Cardinality

Symbol	Meaning
1	One & only one
0 .. 1	Zero or one
M .. N	Numerically specified
*	From zero to any positive integer
0.. *	From zero to any positive integer
1 .. *	From one to any positive integer

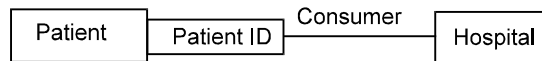
Fig. - 12.16

Cardinality : More than one patient undergo treatment

ii) Qualified Association : A qualifier is an association attribute, whose values partition the set of objects related to an object across an association. It specifies how a specific object at the many end of the association is identified. The qualifier rectangle is part of the association path, not part of the class. Here the patient ID uniquely qualifies the patient class associated with Hospital.

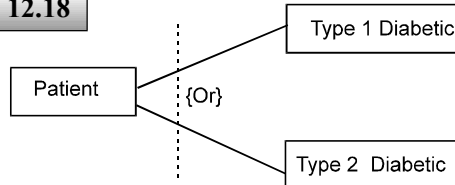
Fig. - 12.17

a) Qualifier in Class



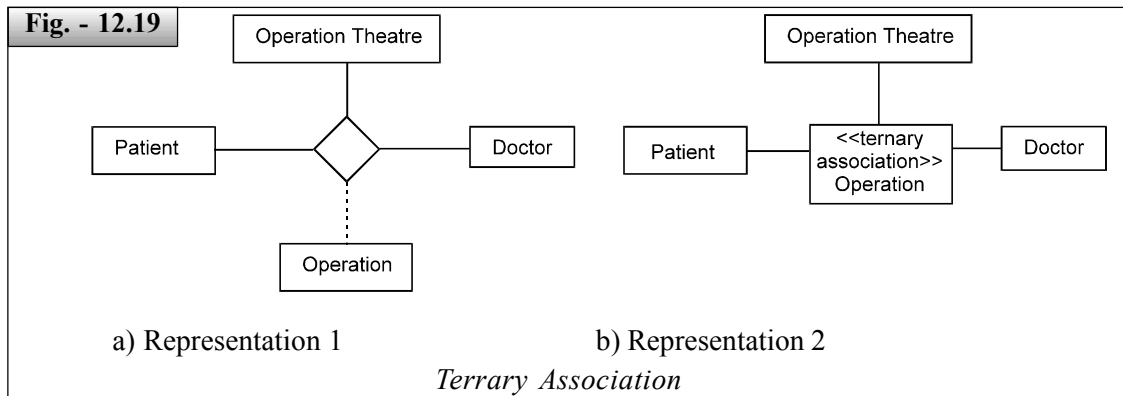
b) Example of Qualifier
Qualified Association

ii) OR Association indicates a situation in which only one of several potential associations may be instantiated at one time for any single object. Here a patient may either type 1 Diabetic or Type 2 Diabetic.

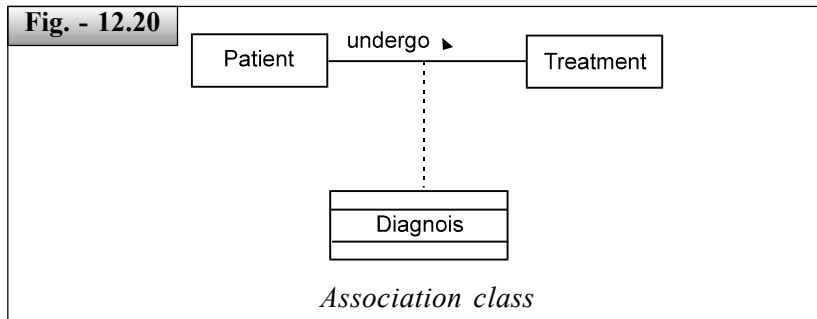
Fig. - 12.18

OR Association

iii) N-any association is an association among more than two classes. Most of the associations are referred to as 'binary associations', since they link two classes. A class maybe linked to more than one class, by means of an n-ary association. A dashed line may attach an association class symbol to the diamond, indicating an N-Ary association that has attributes, operations or associations. A simple ternary association is given in figure below.

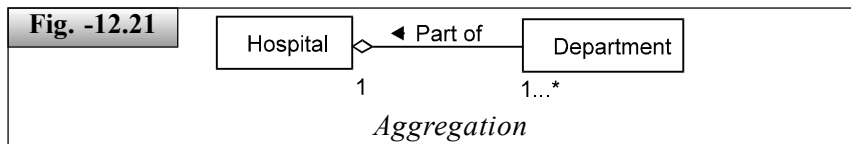


iv) Association Class : An association may be represented by a class to add attributes and operations to that association. A notation uses a dashed line to attach a class to an association. An association that contains attribute but does not directly participate in relationships with other classes is called an attribute association.

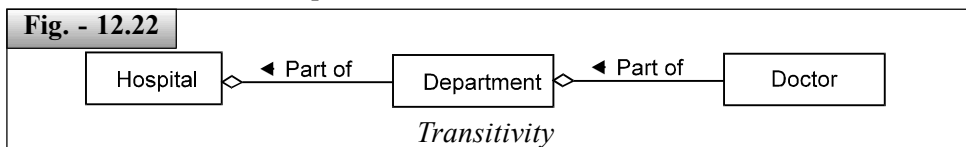


D) Aggregation :

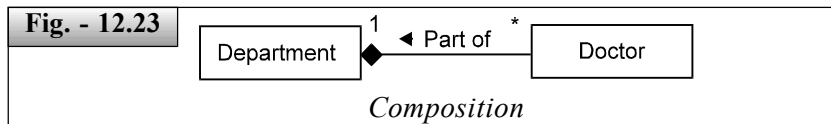
Aggregation is a form of association. It indicates a whole-part relationship. A hollow diamond is attached at the end of the path near whole. The aggregation only applies to one of the roles of an association, regardless of the number of classes involved.



There are two major properties of aggregation : Transitivity and Anti-symmetry. Transitivity is defined as If A is a part of B, and B is a part of C, then A is a part of C. Anti-symmetry is defined as the property of a part-of relation where, if A is a part of B, then the reverse is not true, i.e. B is not a part of A.



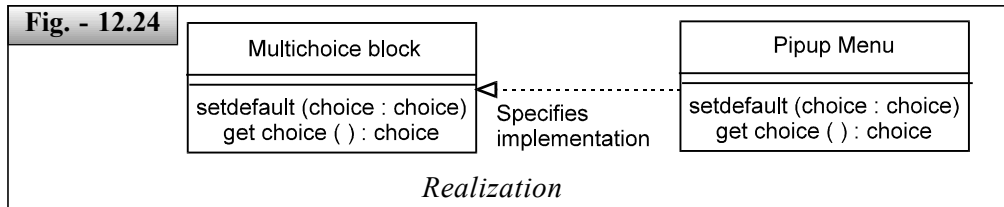
Composite aggregation or composition means that the part is a member of only one composite object, and that there is an existence and disposition dependency of that part on the composite.



Some of the aggregation patterns are assembly container and collection manner.

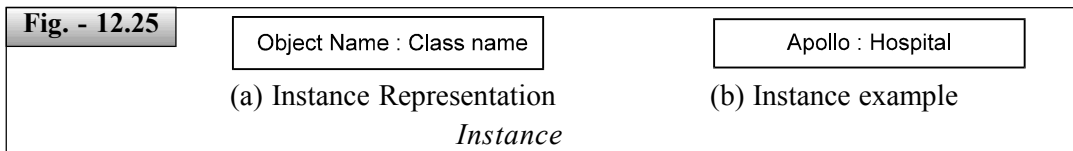
(E) Realization :

The realization relationship connects a model element, a class to another model element, such as an interface, that supplies its behavioral specifications but not its structure or implementation. Realization relates two elements at different semantic levels.



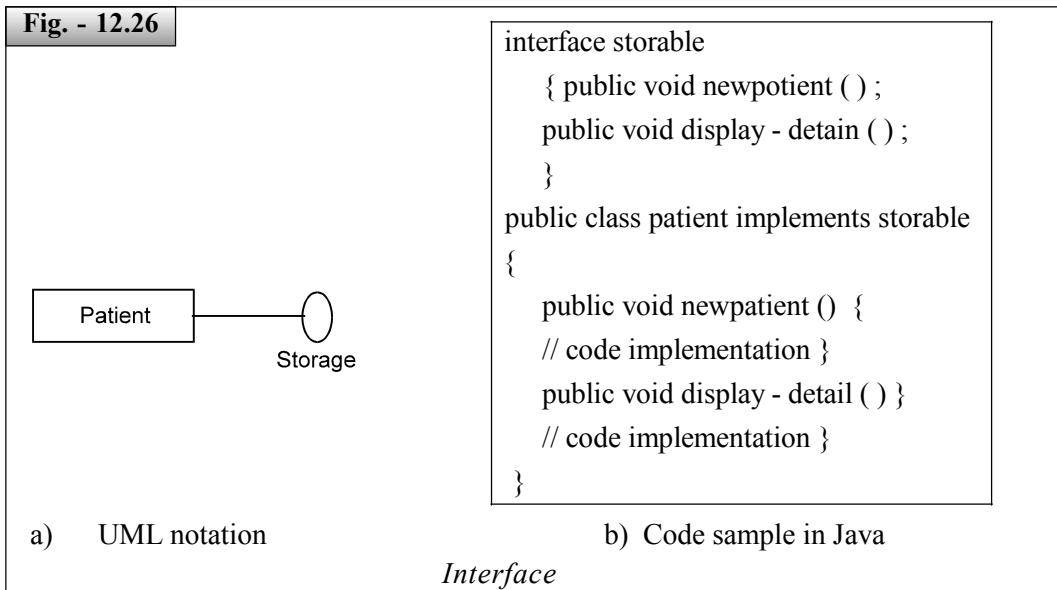
(F) Instance :

An instance is a run time entity with identity, to distinguish from other run time entities of the same class. Instance has a value at any time.



(G) Interface :

Interface is used to describe the visible behaviour of the class / component / package. A class provides services to other classes only through its interface. The public operation of class is called as interface.



(H) Template Class :

Template class / parameterized class is the generic class, and cannot be used as it is. During instantiation, actual parameters customize the real class based on the template class. Template class is used in OOD.

Fig. - 12.27

```
< template class T >
```

```
class vector
```

```
{
    T * V ;
    int size ;
    public :
        vector (const int m&)
        { v = new T [size = m]; } ;
};
```

Vector

- size : int

<< size : int >>

+ vector (in m : int)

v

a) code sample in c++ for template class

b) Template class in UML.

Template Class

I) Abstract & Concrete Classes :

Abstract classes cannot be instantiated directly. They do not give birth to objects, but may be used as a more general specification - of type - in order to manipulate objects that are instances of one or more of their subclasses. By convention the names of abstract classes are italicized.

Fig. - 12.28

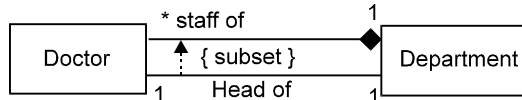
Class
{anstract}

Abstract Class : Italicized

Concrete class is a generalized class, which can be directly instantiated of necessity its implementation must be fully specified. Concrete class provides implementation of all its operations.

12.9.2 Extensibility mechanisms**(A) Constraints :**

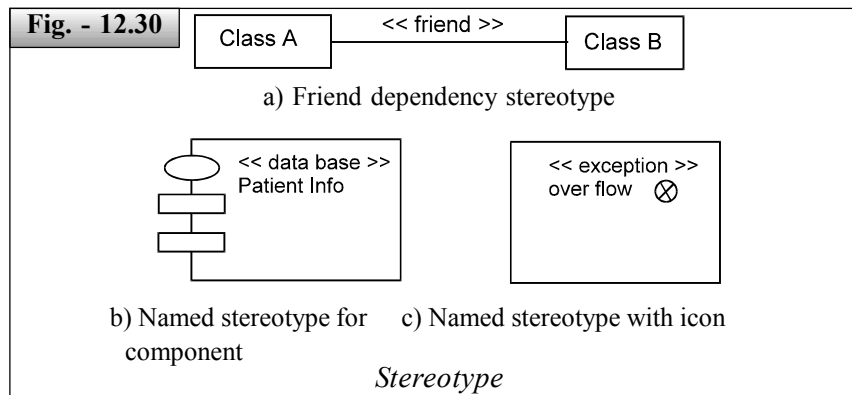
A constraint is a Boolean expression represented as a string to be interpreted in a designated language. It may be denoted on paths of association or as a note. Here, the one of the Doctor belongs to the Department is Head of the Department.

Fig. - 12.29

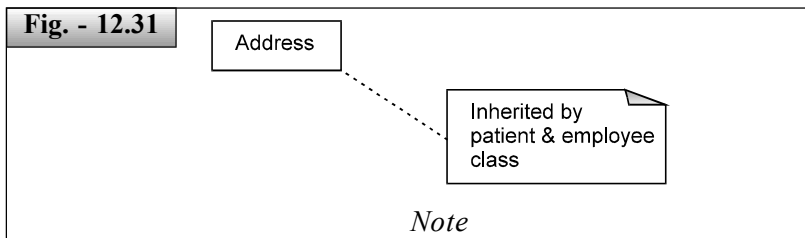
Constraint

(B) Stereotype :

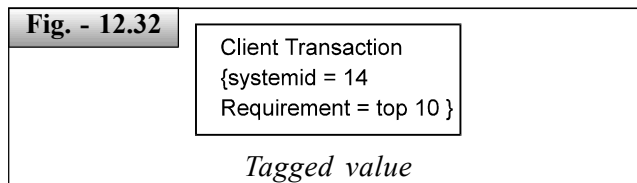
Stereotype is a kind of model element defined itself. It is a part of range of extensibility mechanisms. The kind of stereotype is mentioned within the notation <<>>. The stereotyped element may have its own distinct icon. It may have constraints that apply to its usage.

**(C) Note :**

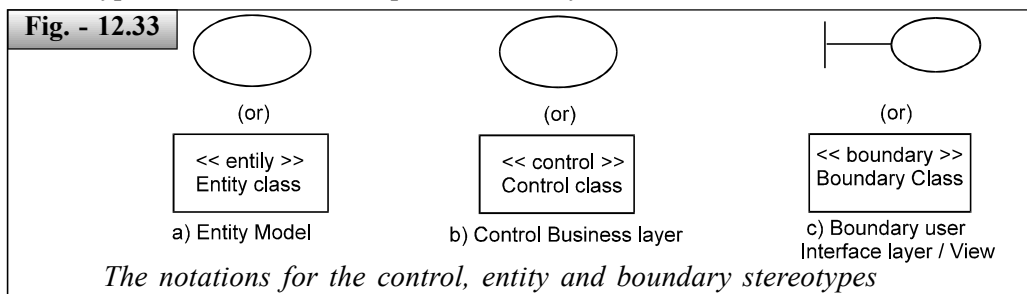
Note is a symbol for displaying a comment or other textual information. A note may be transformed into a constraint by using a stereotype and, in that case it modified the semantics of the model elements to which it is attached.

**D) Tagged Value :**

A tagged value is a (name, value) pair that describes a property of a model element. The tagged value modifies the semantics of the elements to which it relates.

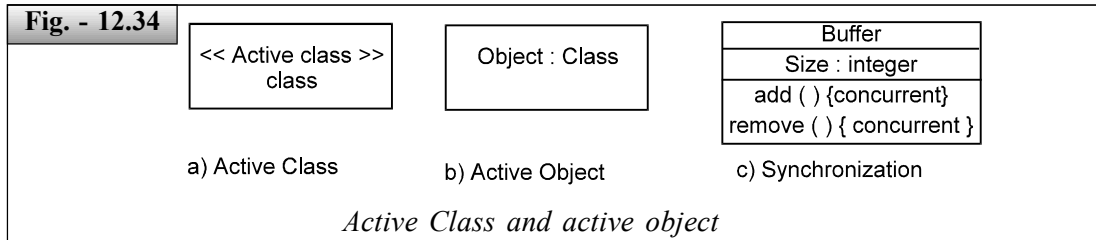
**E) Entity, Control and Boundary Types :**

Classes can be stereotyped to control, boundary and entity. These stereotypes are used to increase the semantic meaning of the classes and their usage in modeling situations. These stereotypes are used in OOAD process to analyse and realise use cases in more detail.



F) Active Classes :

Active classes are used to model the concurrent behaviour. They represent an independent flow of control. Active class executes in parallel with other active objects and initiates actions by itself. The class is stereotyped with << Active class >>.



An active class is implemented by process or thread of control.

i) **Process** : It specifies a heavyweight flow that can execute concurrently with other processes.

ii) **Thread** : It specifies a lightweight flow can execute concurrently with other threads within the same process.

There are four possible combination of communication, which includes, a message from passive to another passive, active to another active, passive to active and active to passive. Various mechanisms are used to enable active object communication. They are :

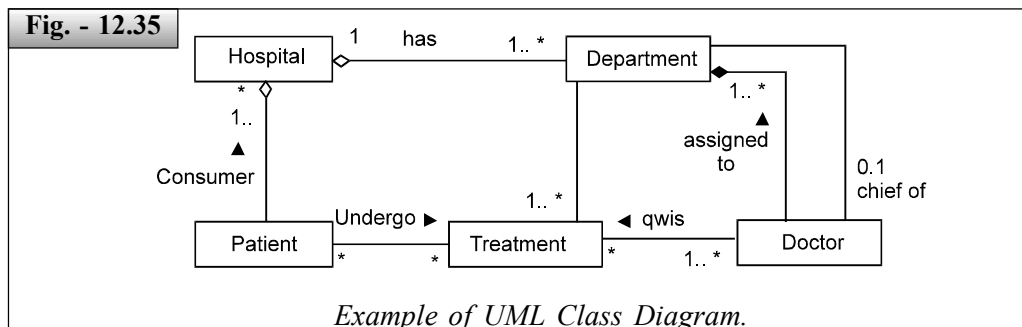
i) **Operation Calls** : Sending a synchronous message to object and waits for the operation to finish and return.

ii) Mailboxes are used to place a message as asynchronous message.

iii) Shared memory : A block of memory is reserved for communication of active objects to perform read / write information.

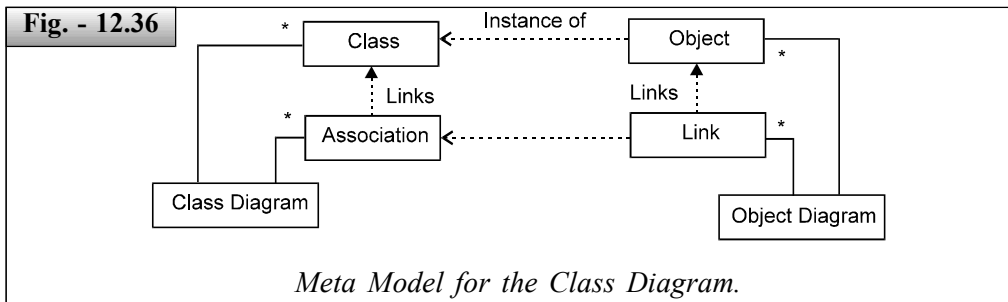
12.9.3 Example of UML Class Diagram

The example of UML class diagram is given for the DIET system. The roles and cardinality of the class are mentioned.



12.9.4 Meta Model

The UML Metamodel is a model of the UML itself. A model is an abstract description of a system or a process. Metamodel provides a single, common and definitive statement of the syntax and semantics of the elements of the UML. A simple Metamodel of UML class Diagram is given in figure below. Comparison of Model and Metamodel is given in Table below.



12.10 INTERACTION DIAGRAMS

Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are **sequence** and **collaboration** diagrams. This example is only meant as an introduction to the UML and interaction diagrams. If you would like to learn more see the **Resources** page for a list of more detailed resources on UML.

When to Use: Interaction Diagrams

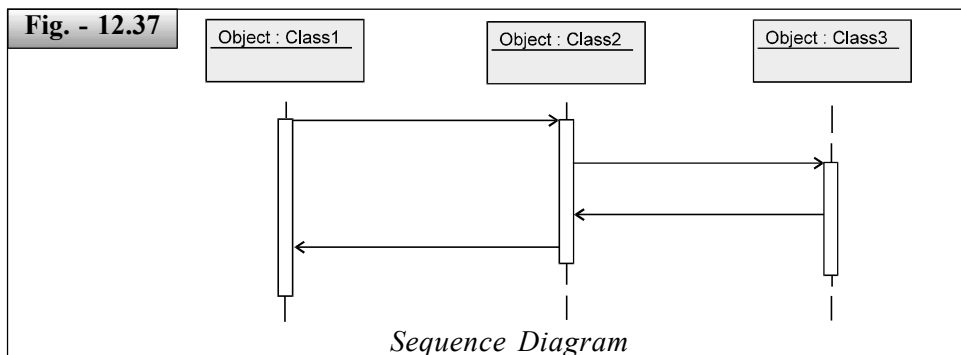
Interaction diagrams are used when you want to model the behavior of several objects in a use case. They demonstrate how the objects collaborate for the behavior. Interaction diagrams do not give a in depth representation of the behavior. If you want to see what a specific object is doing for several use cases use a **state diagram**. To see a particular behavior over many use cases or threads use an **activity diagrams**.

How to Draw: Interaction Diagrams

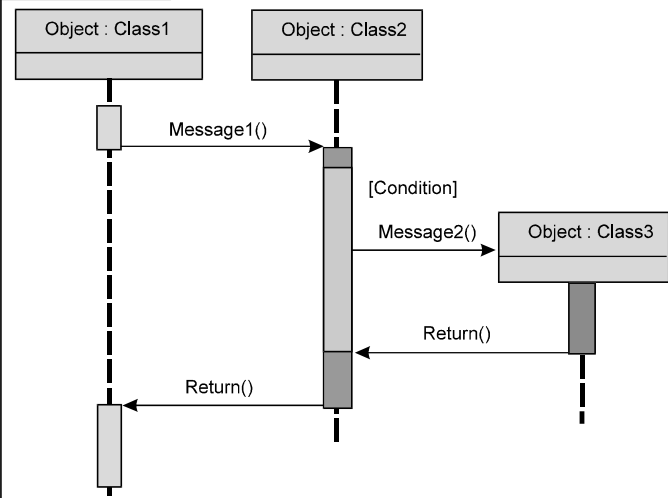
Sequence diagrams, collaboration diagrams, or both diagrams can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur. Collaboration diagrams demonstrate how objects are statically connected. Both diagrams are relatively simple to draw and contain similar elements.

12.10.1 Sequence diagrams

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. the diagrams are read left to right and descending. The example below shows an object of class 1 start the behavior by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message.

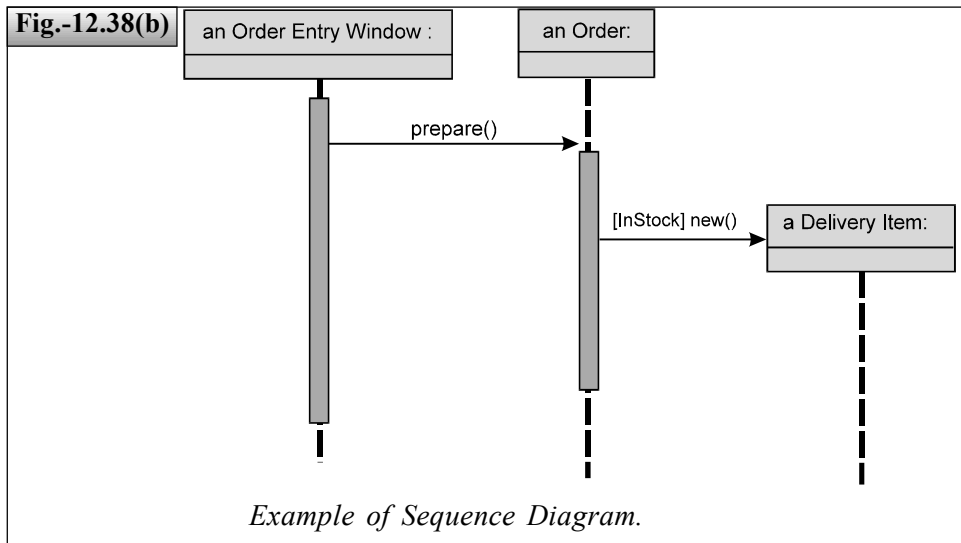


Below is a slightly more complex example. The light blue vertical rectangles represent the objects activation while the green vertical dashed lines represent the life of the object. The green vertical rectangles represent when a particular object has control. The represents when the object is destroyed. This diagrams also shows conditions for messages to be sent to other object. The condition is listed between brackets next to the message. For example, a [condition] has to be met before the object of class 2 can send a message() to the object of class 3.

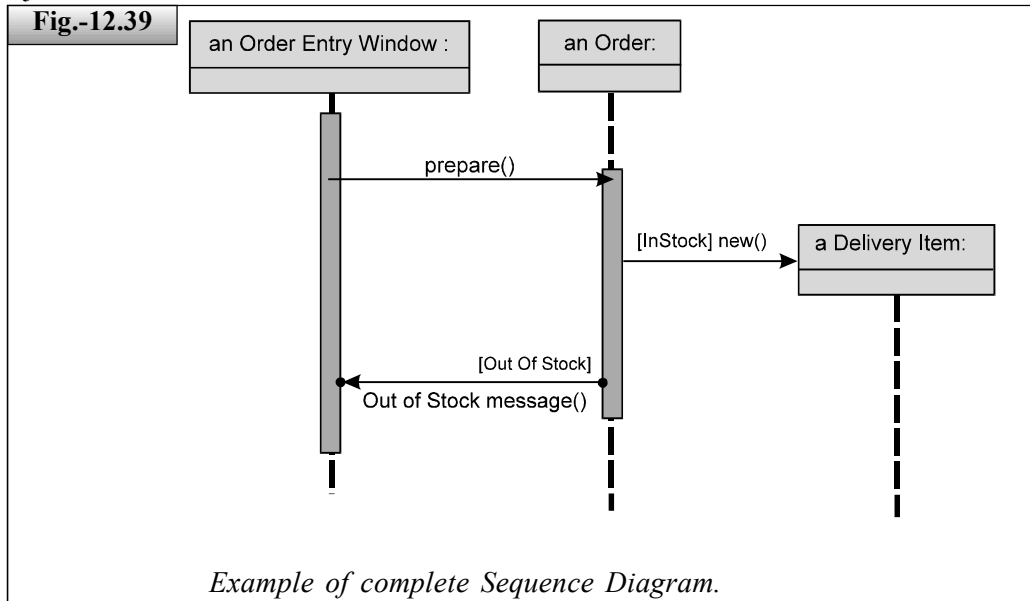
Fig.-12.38(a)*Example of Sequence Diagram.*

The next diagram shows the beginning of a sequence diagram for placing an order. The object an Order Entry Window is created and sends a message to an Order object to prepare the order. Notice the the names of the objects are followed by a colon. The names of the classes the objects belong to do not have to be listed. However the colon is required to denote that it is the name of an object following the objectName:className naming system.

Next the Order object checks to see if the item is in stock and if the [InStock] condition is met it sends a message to create a new Delivery Item object.

Fig.-12.38(b)*Example of Sequence Diagram.*

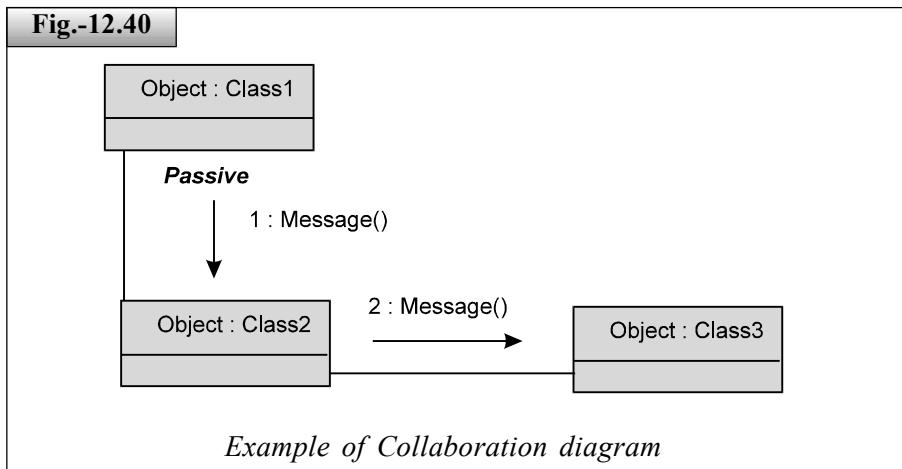
The next diagrams adds another conditional message to the Order object. If the item is [OutOfStock] it sends a message back to the Order Entry Window object stating that the object is out of stock.



This simple diagram shows the sequence that messages are passed between objects to complete a use case for ordering an item.

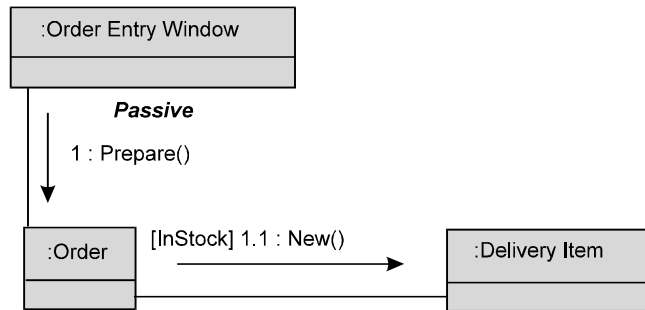
12.10.2 Collaboration diagrams

Collaboration diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in UML.



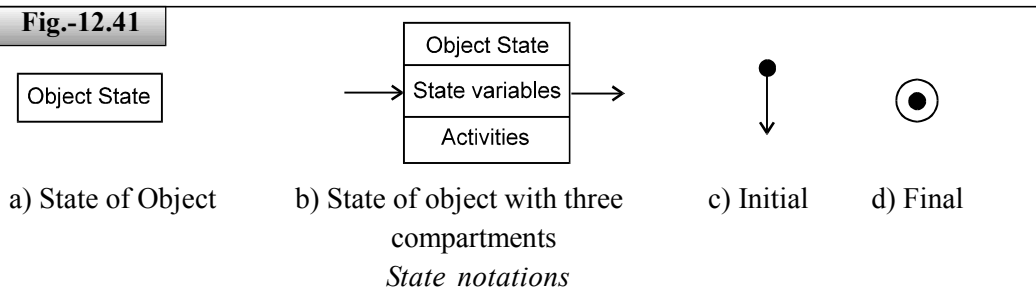
Example-12.1

The example below shows a simple collaboration diagram for the placing an order use case. This time the names of the objects appear after the colon, such as :Order Entry Window following the objectName:className naming convention. This time the class name is shown to demonstrate that all of objects of that class will behave the same way.

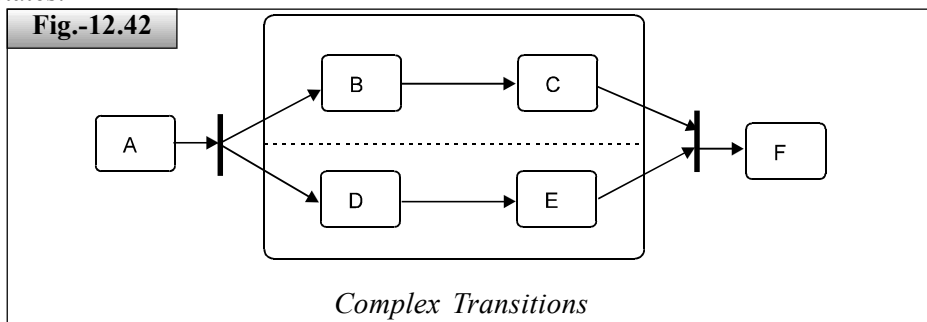
**12.11 STATE-CHART DIAGRAM**

Sometimes, it is desirable to model the behavior of a single object class, especially if the class illustrates significant dynamic behaviour. State-chart transition diagram may be created for these types of classes. It shows a life history of a given class, the events that causes a transition from one state to another, and the action that results from a state change.

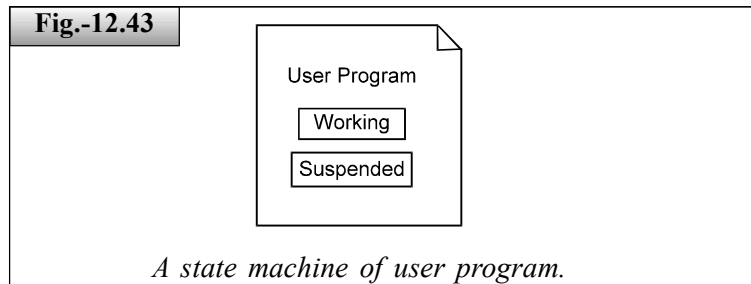
States are represented a rounded rectangles; each state has a unique identifying name. A state may contain three kinds of compartments as shown in figure 12.40. An object is not always in the same state at all times, and an object cannot be in on unknown or undefined state.



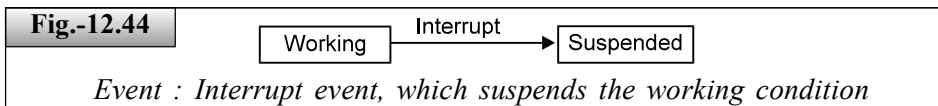
States are linked via unidirectional connections called transitions. A complex transition may include multiple source and target states. It represents synchronization or a splitting of control into concurrent threads. After a complex transition, it fires all its destination states.



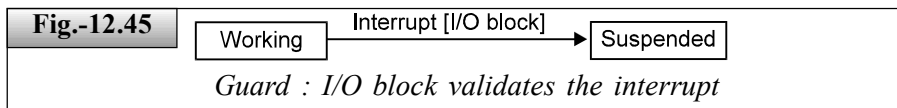
A State machine is an abstraction of all possible behaviours. It is similar to class diagrams, which are abstraction of the static structure. A state-chart diagram represents state machines from the perspective of states and transitions.



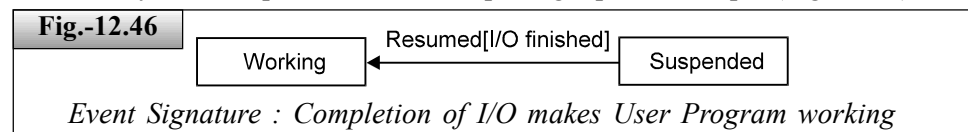
An event corresponds to the occurrence of a given situation in the problem domain. An event is used as a trigger to go from one state to another. There are four types of events such as guard-condition, receipt of an explicit signal from another object, receipt of a call or on operation by another object and passage of a designated period of time.



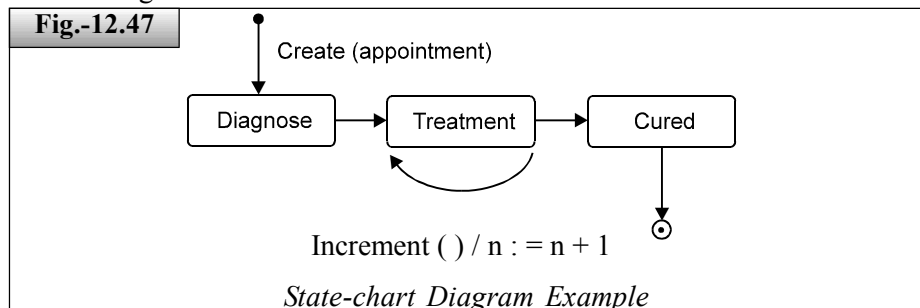
Guard is a Boolean condition, which, validate the triggering of an event. The transition will fire when the condition becomes true.



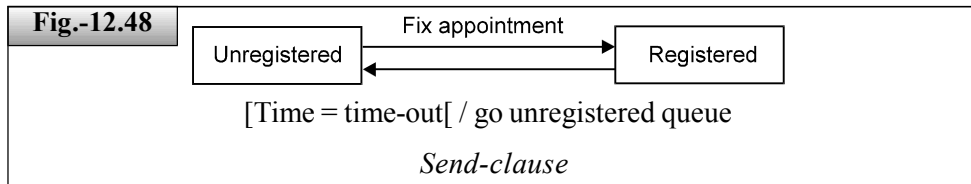
Event signature consists of an event-name and parameters, specifying the event that triggers a transition, along with additional connected to the event. A User Program becomes working condition by resume operation while completing Input and Output (Fig. 12.46)



Action - expression is a procedural expression executed when the transition fires. Nested action expressions or recursive action expressions are not allowed. The notation $:=$ is used for assignment.

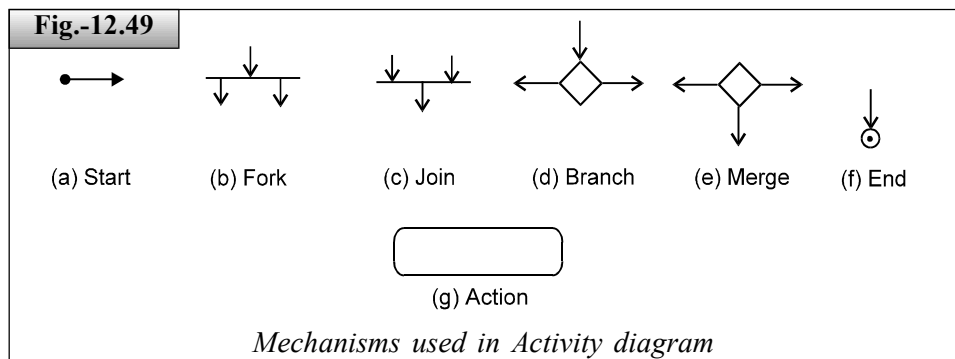


The Send-clause is a special case of action. It is an explicit syntax for sending a message during the transition between two states. The syntax consists of a destination-expression and an event-name. For example, consider the registered patient (Figure 12.48) must confirm and should be physically present 15 minutes before the time allotted. Else he is considered to be unregistered patient and he has to obtain fresh appointment. It is represented using send-clause.

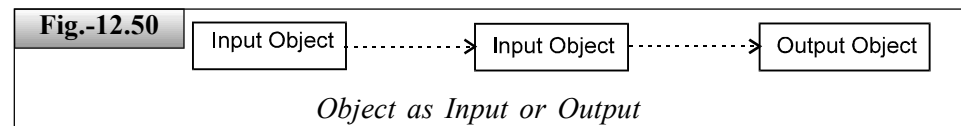


12.12 ACTIVITY DIAGRAM

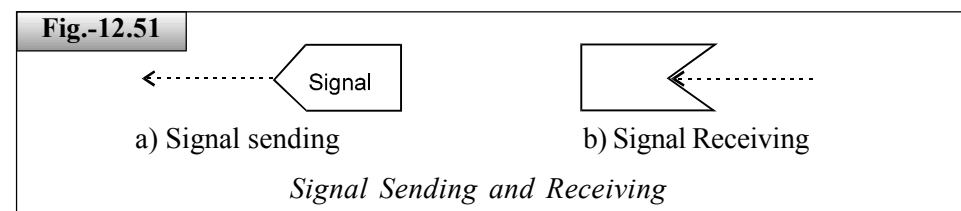
An activity diagram is used to model the flow of a particular use case or the entire business process, to provide the flow of program. A branch has a single incoming transition and several guarded outgoing transitions. Only one of the outgoing transitions can be taken, so the guards should be mutually exclusive. A merge has multiple input transitions and a single output. The activity diagram describes the sequencing of activities, with support for both conditional and parallel behaviour.



Objects can be shown in Activity diagrams. They are either input or output from the actions.

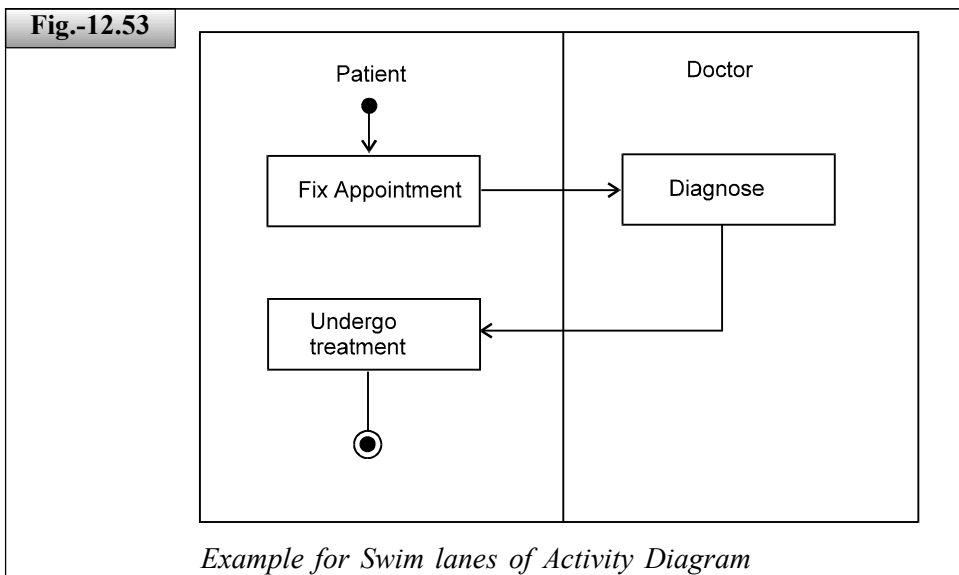
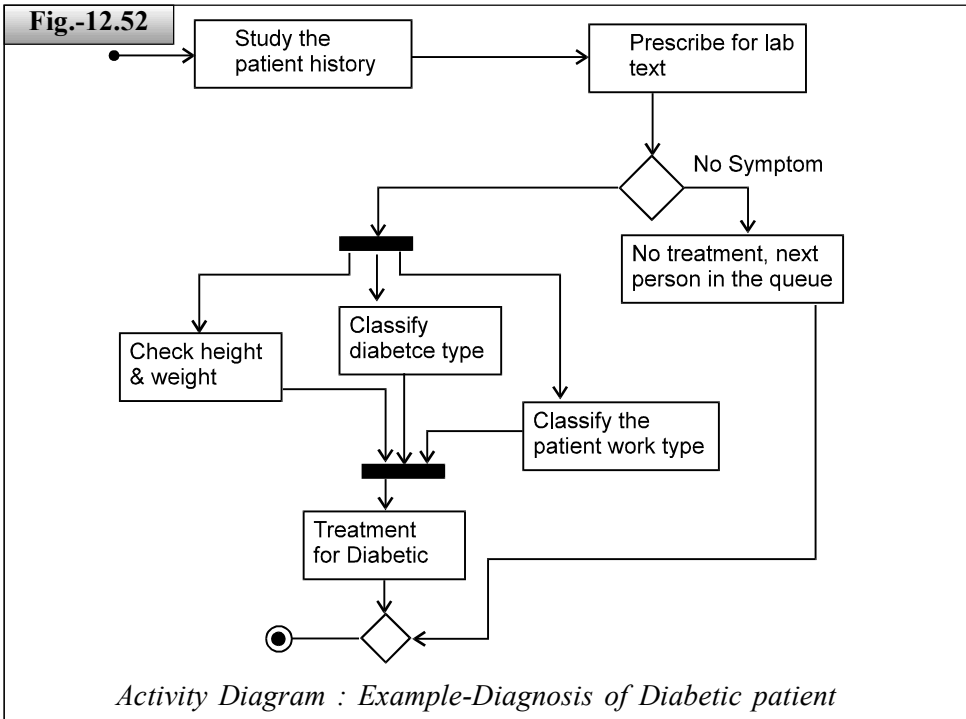


Signals may be sent or received in activity diagram. There are two symbols, one for sending and one for receiving a signal.



Activity diagram is variation of State-chart Diagram. The state-chart diagram is used to define the various states of objects. Interaction diagram deals with the message communication among objects.

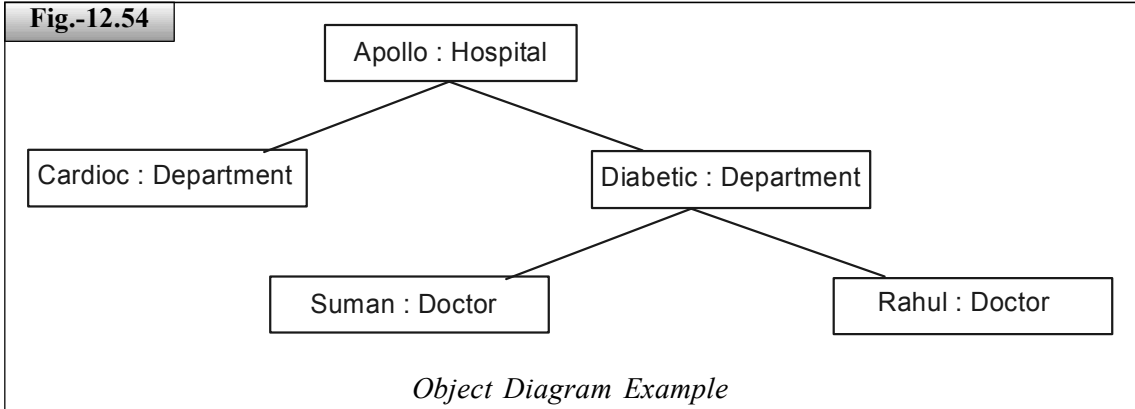
Activity diagram varies from the conventional flowchart. Flowcharts are limited to sequential processes, whereas activity diagrams can handle both sequential and parallel processes.



12.13 OBJECT DIAGRAM

Object diagram model the instance of classes. It shows a set of objects and their relationships at a point in time. Object model is used to design the static processes view of the system. Object diagram commonly contain objects and links. The use of object diagram is limited but they can be used to demonstrate a complex class diagram.

Fig.-12.54



12.14 IMPLEMENTATION DIAGRAMS

Implementation diagrams show the implementation phase of system development, and its architecture. Implementation is described by physical and logical structure. Physical architecture deals with a detailed description of the system and decomposition, in terms of hardware and software. It defines the physical location of classes and objects in which processes, programs and computers. Also defines the dependency between different code files and connection of hardware devices. Logical architecture deals with functionality of the system. It defines the functionality of system deliver, relationship among classes, class and object collaboration to deliver the functionality.

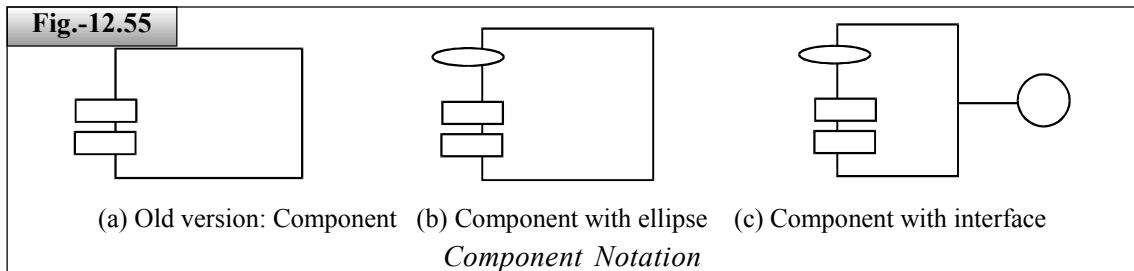
Implementation deals both source code structure and the run-time implementation structure. There are two different implementation diagrams: Component diagrams (structure of code) and Deployment diagrams (structure of run time system). They are simple and high level diagrams.

12.14.1 Component Diagram

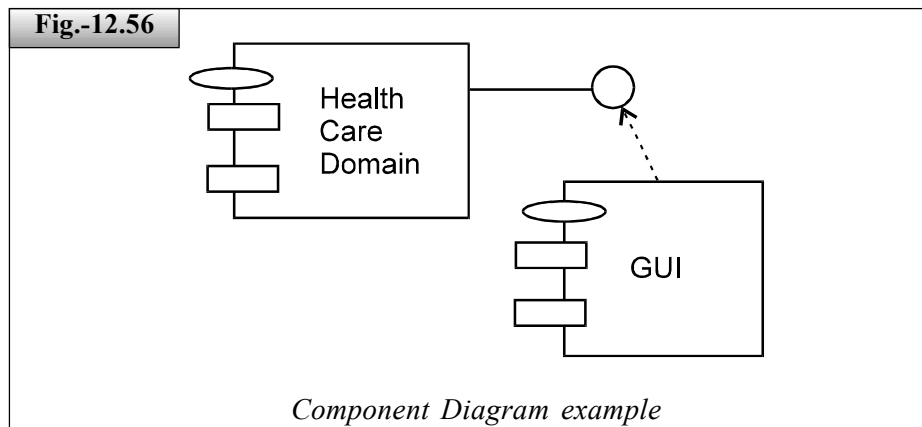
Component diagram specifies the software components and their relationships within the implementation environment. It represents the structure of the code. Components represents the implementation of physical architecture of concepts and the functionality defined in the logical architecture. A component is shown as a rectangle with an ellipse and two smaller rectangles to the left (Figure 12.55). The two small rectangles are left over from an old notation that used to put the component interfaces in the rectangles. The name of the component is written below the symbol or inside the large rectangle. Though components are types, only executable components can have instances. The software

component can be:

- (i) Source component / Compile-Time component (contains source code implementing one or more classes) use stereotypes such as << file >>, << page >>, << document >> and << database >>.
- (ii) Binary component / Link-time Component (Object code resulting of compiling source component such as object code file, static library file / dynamic library file), use stereotype <<library>>.
- (iii) Executable component / Run-time Component (executable program that is the result of linking all binary components) use stereotype <<application>>.



The dependency among the components indicates that a component refers to services offered by other components. A dashed arrow drawn from the client to supplier represents a dependency relationship (Figure 12.56).

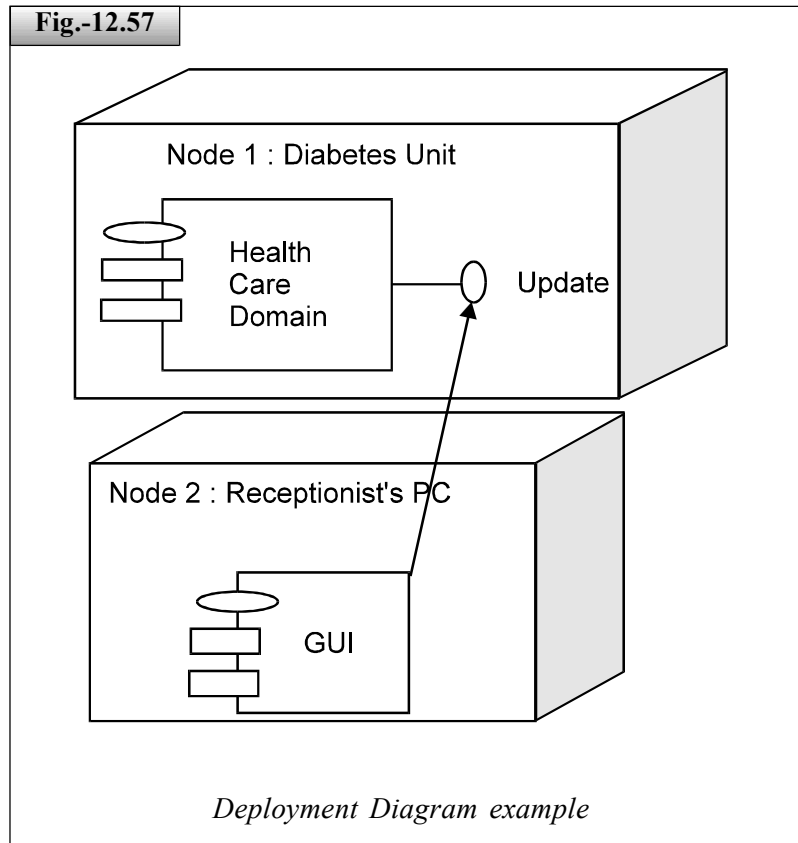


12.14.2 Deployment Diagram

The deployment diagram describes the run-time architecture of processors, devices and the software components. It describes the physical topology of the system and its structure. It specifies which components and logical elements (class, objects and collaborations) are executed in that node (system). Nodes are physical objects or devices such as computer, printers, card readers and communication devices and so on. A node is drawn as a three-dimensional cube with names inside it. Communication association connects nodes to each other. Executable component instances may be contained within node, to show that they reside and execute on the node instance. Components are connected

to each other via dashed-arrow line dependency. Like components, objects can also be placed inside the nodes directly or components within nodes. In summary, the deployment diagram

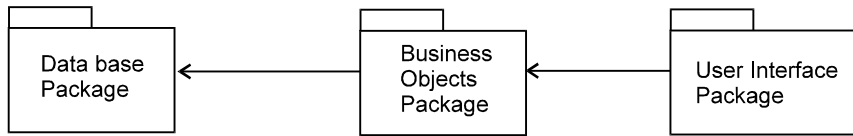
- (i) Shows the physical relationship among software and hardware components in the delivered system (Figure 12.57)
- (ii) is a good model to show how components and objects are routed and move around in a distributed system
- (iii) Shows the configuration of run-time processing elements and the software components, processes and objects that live on them.



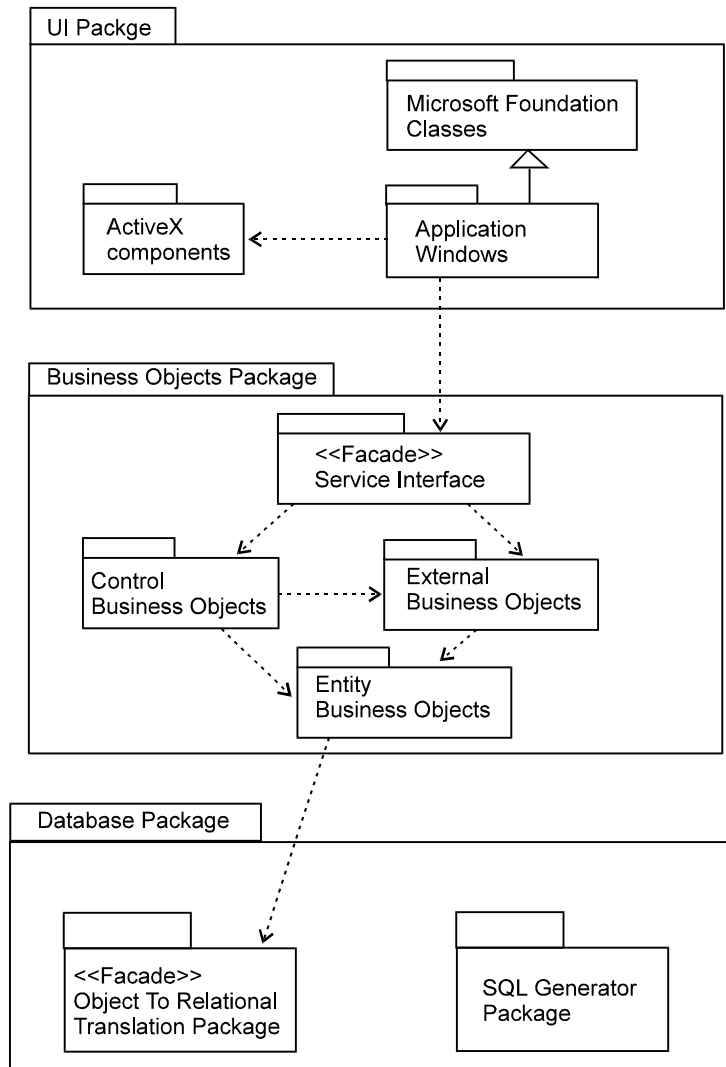
12.15 PACKAGES AND MODEL MANAGEMENT

A package is a grouping mechanism, where by all kinds of model elements are linked. It is a general-purpose mechanism for organising elements into semantically related groups. Packages can be nested within other packages. The package is shown as a large rectangle with a smaller rectangle attached on the top left corner of large rectangle. Component represents the physical package of code and a single class may be reused in several components. But the class can be defined in only one package.

Fig.-12.58



(a) Simple Package



(b) Dependencies between Layered Architecture packages

Package Example

12.16 OBJECT CONSTRAINT LANGUAGE

Object Constraint Language (OCL) is a formal language, similar to structured English to express side-effect-free constraints within UML models. OCL statements are depicted on UML diagrams in the format “{Constraint description}”. The OCL is a text language for writing navigation expressions, Boolean expression, and other queries. It is used to construct expressions for constraints, guard conditions, action, precondition and post conditions, assertions and other kinds of UML expression.

Table-12.6 : OCL Description

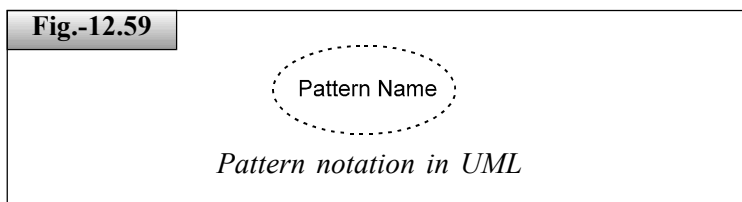
Syntax	Semantics	Example
Item. Selector	Name of an attribute of the item / object	Perry. PatientID
Item. selector (argument _{list})	Name of an operation of the object with its parameter	Perry. fixAppointment ("12/05/2005")
Item.Selector [qualifier - value]	Selector designates a qualified association. Used for array indexing.	Hospital. Department [1..5]
Self	Denotes the current object	
Set- > set-property	Name of built-in OCL function on sets.	Patient->size =10
Set->select(Boolean expression)	Result is subject of objects in set for which the Boolean expression is true	Department. Doctor ->select(experience > 5)
Operators: =, <, <=, >, >=, <>, +, -, *, /, and, Not	Usual arithmetic and Boolean expression	Patient->size >= 10

12.17 MODELING PATTERNS & FRAMEWORKS IN UML

12.17.1 Patterns

Patterns are reusable concepts. There are two kinds of patterns analysis and design. These patterns can be modeled using UML diagrams for reuse in OOAD process. The context & interaction of pattern in UML is described with the example of Facade and proxy Design pattern is dealt in this section. A pattern is documented as collaboration in UML. The symbol of collaboration is dashed ellipse with pattern name inside.

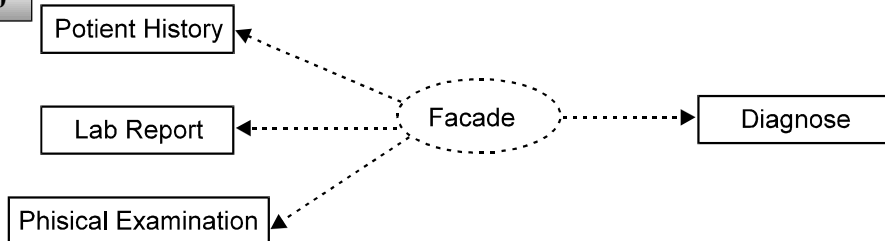
Fig.-12.59



i) Modeling Facade Design Pattern

Facade Defines a higher-level interface that makes the subsystem easier to use. Consider in Health Domain, Database must be connected with user interface. A single user interface is defined of obtain information from these classes for diagnosis to provide proper treatment.

Fig.-12.60

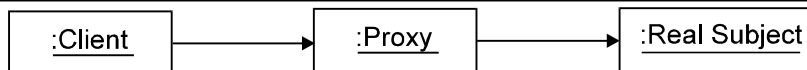


Facade Design Pattern : Documented as Collaboration

ii) Modeling Proxy Design Pattern

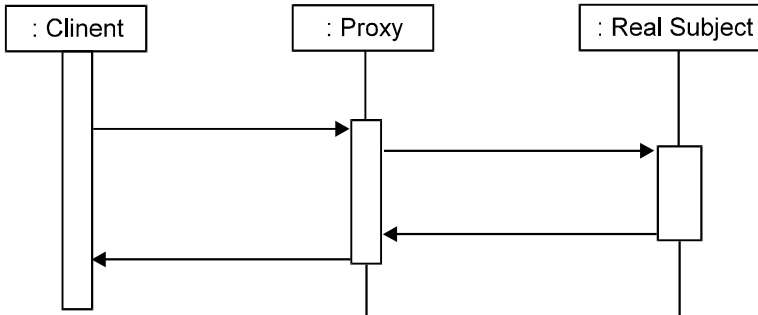
Proxy pattern separates an interface from an implementation into different classes. The context of proxy pattern is described as object diagram.

Fig.-12.61



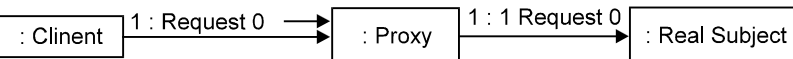
Context of Proxy Pattern : Object Diagram

Fig.-12.62



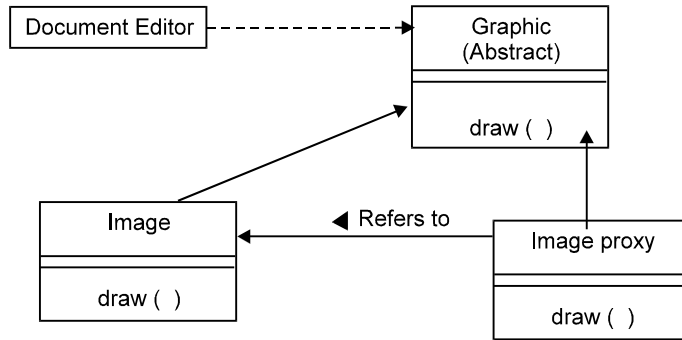
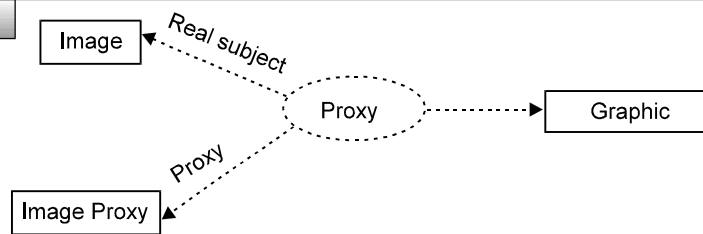
Interaction in the proxy pattern : Sequence Diagram

Fig.-12.63



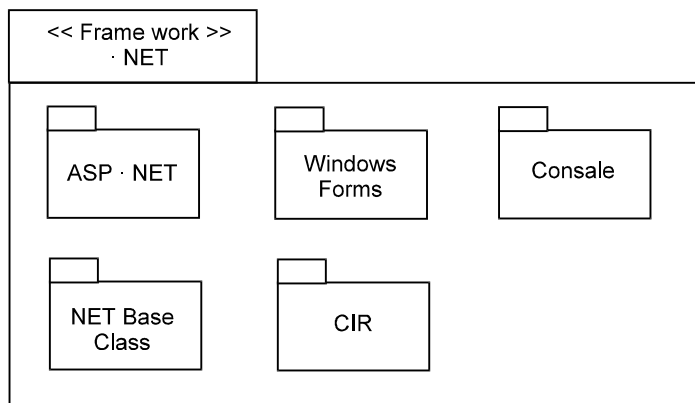
Interaction in proxy pattern : Collaboration Diagram

Consider a client interface is Document Editor of word, which wants to draw a graph. The system is represented using proxy pattern as shown in figure below. Document Editor (Client) accesses embedded images through the interface defined in abstract Graph (subject) class. Image proxy maintains the file name as a reference to the image on disk.

Fig.-12.64**Fig. 12.64 : Proxy Pattern to draw graph in Document editor.****Fig.-12.65****Fig. 12.65 : Proxy Pattern Documented as collaboration**

12.17.2 Frameworks

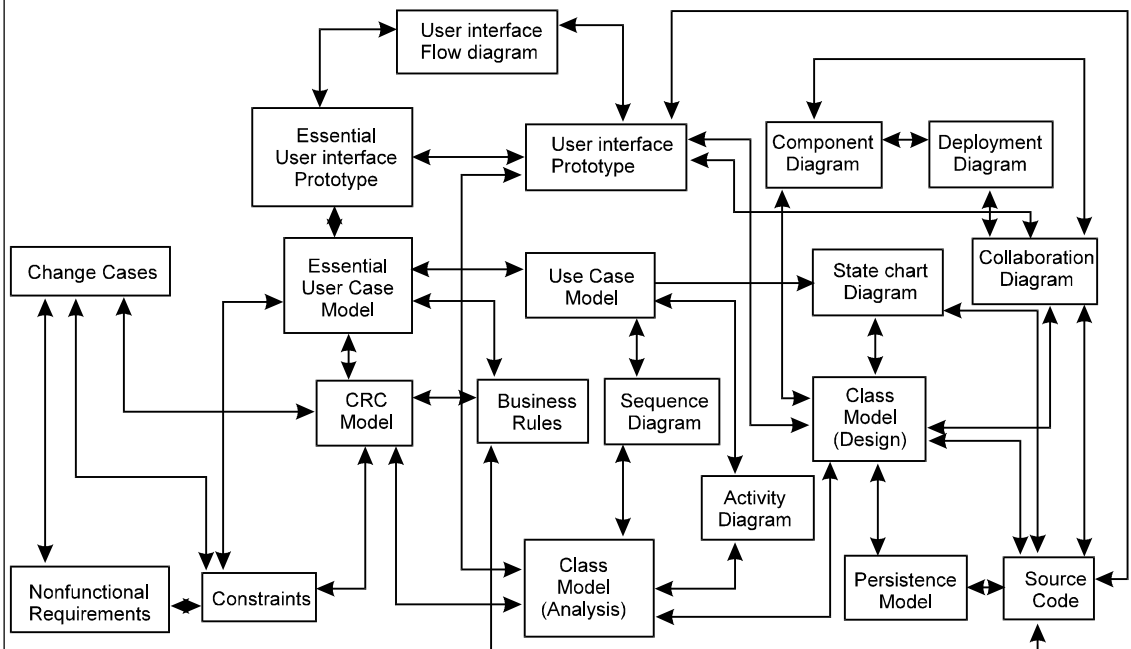
The representation of large systems calls for many classes and its complexity limits human understanding. The macrostructure must be introduced to facilitate model understanding. A Simple .NET Framework and its components are shown in the figure below. The package notation is used to represent the collection of components and it is stereotyped.

Fig.-12.66*NET Framework : Package Representation*

12.18 UML COMPATIBILITY

In OO approach, the focus of high quality product is at the time of production itself, prior to delivery rather than correcting them after delivery. Use cases are used for determining objects. UML diagrams are used throughout the software development life cycle from Requirement specification and Analysis to Design, code and Implementation. UML permits reverse engineering. The product development becomes more iterative and non-linear approach, which facilitates the development easier.

Fig.-12.67



UML Model used throughout the product life suggested by Scott Ambler.

SUMMARY

The Unified Modeling language, its architecture, various diagrams used in UML and their notations are dealt. The UML diagrams are explained with simple example for better understanding. The benefits and applicability of each diagram are highlighted. Also this chapter is dealt with OCL, Modeling patterns and frameworks in UML.

EXERCISE

1. Define Model. What are the advantages of modeling the system?
2. Explain UML and its importance.
3. Describe various views in UML architecture
4. What are the UML foundations?
5. Can UML be a 'Complete Solution Provider'? Support your answer with proper justification.
6. Summarize the benefits of UML diagrams.
7. Describe the consistency of common mechanism used in UML.
8. Define the following terms a) Use Case b) Actor c) Use Case Diagram. Give example for Use Case Diagram.
9. What are the various notations used to represent Actor?
10. Give example for relationships used in Use Case diagram.
11. Define class. Describe Class Notation with example
12. Describe the given terms used in Class Diagram with example
 - a) Dependency
 - b) Class Hierarchy
 - c) Association
 - d) Cardinality
 - e) Qualified Association
 - f) Recursive Association
 - g) N-Ary Association
 - h) Association Class
 - i) Recursive Association
 - j) Ordered Association
 - k) Aggregation
 - l) Composition
 - m) Realization
 - n) Instance
 - o) Interface
 - p) Abstract Class
 - q) template class
 - r) Active class
 - s) Thread and Process
13. What is the extensibility mechanism used in UML?
14. Describe MetaModel and its usage.
15. Define time and space representation to represent real and distributed system.
16. Demonstrate sequence diagram with suitable example
17. When will you use collaboration diagram to show interaction of classes?
18. How the activity diagram is used to represent complex functionality?
19. What is swim lane?
20. Give example for Object Diagram.
21. Describe Component Diagram
22. How deployment diagram is useful to locate software components?
23. What are the benefits of packaging and model management?
24. Describe OCL with suitable example.
25. How patterns and frameworks are modeled in UML?
26. How UML diagrams are applied to various phases of software development phases?

Object Oriented Software Engineering

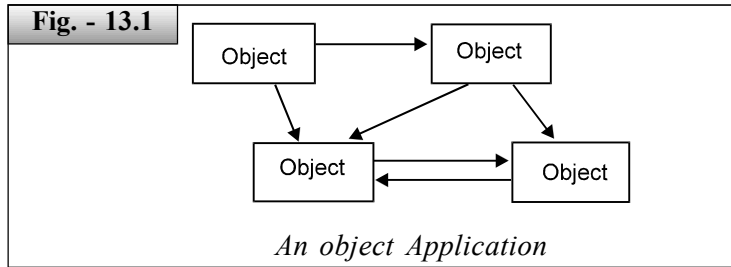
13.1 INTRODUCTION

Over the last two decades, the function/procedure oriented structured methodology in software engineering being supplanted by a more sophisticated, flexible, reusable object oriented approach. This approach address rising demands in highly sophisticated software, technical complexity and continuous evolution. It allows to create systems that are easy to develop, maintain and also easy to extend and reuse. Using Object orientation, it is possible to model systems in terms that match human thinking. The fundamental concepts of Object oriented (OO) approach are mature and established in practice. So it is easy to interpret and understand the system and its development. Also it reduces the semantic gap between system and its reality. This chapter introduces the basics of object oriented software development.

One of the most important features of object orientation is that the whole development process is based around a single, central concept - that of object (Figure 13.1). Object orientation promises a new approach to system development, which will produce software that is more maintainable, testable, reusable and able to cope with large complex systems. It treats data as a critical element in the program development and does not allow it to flow freely around the system. Some of the characteristics of object-oriented approach are:

- (i) Data is more emphasized.
- (ii) Data and functions that operate on data are bind together as object.
- (iii) Real world things are directly mapped into objects in the model, which minimizes the semantic gap.
- (iv) Data structures are designed to characterize the objects.
- (v) Data is hidden and cannot be accessed by external functions.
- (vi) Follows bottom-up goal driven approach (Requirements are analyzed in terms of users who use and affect the system).
- (vii) Seamless transition of software development is achieved by using same Meta language for various phases.

(viii) Communication of objects is achieved only through proper interfaces.



OO development requires working with more users, offers benefits throughout the entire development life cycle. It insists a new mind set on the part of developers. Object orientation is appropriate:

- (i) To develop complex systems
- (ii) Systems that are prone to change
- (iii) System with User Interfaces
- (iv) Systems that are based on client/server model
- (v) To build E-commerce/Web based applications
- (vi) For Enterprise Application Integration (EAI)

Though OO is versatile it is not advisable to use it, when systems for which structured techniques are ideal (for mainframe based) and which OO cannot be used throughout the development life cycle.

13.2 OBJECT ORIENTED TERMINOLOGIES

Object orientation has its own specialized vocabulary, which must be mastered if the technology is to be understood. The principle terms are introduced in the Table 13.1. The underlying concepts are explained in detail.

Table-13.1 : Principle terms of Object oriented Terminology

Term	Definition
Object	A person, place, thing, concept, event, screen, or report. Object is a software unit packaging together data and methods to manipulate the data.
Class	Defines the general structure of the similar object. Objects that share the same operations / method and common characteristics are grouped into a class. So it is template for creating objects.
Attribute	Data item defined as part of a class or object.
Method	Procedure or function defined as part of a class; refers the implementation.
Message	An instruction to invoke some action on object. It is a request sent to an object to execute one of its methods.
State	Represents a period of time during which the object satisfies some condition.

Encapsulation	Packaging data and operations into an object.
Data Abstraction	Allows essential operations and data to be public without including their background details.
Data hiding	Making the encapsulated data of an object inaccessible and invisible to other objects.
Inheritance	Mechanism for defining a new class in terms of an existing class.
Polymorphism	Capability of a method to be applied to objects of different types; Ability to hide different implementations behind a common interface.
Overloading	kind of polymorphism (compile time), which allows two, or more methods use the same name with different parameter list.
Strong and weak typing	In strong typing, exact data type of arguments must be defined in the program itself. Weak typing permits to mention the data type of arguments at run time.
Overriding	Kind of polymorphism (run time), capability of sub class to override the characteristics of the super class. It occurs when a sub and super class methods use the same name with identical signatures, such that subclass hides the super class method.
Object Persistence	The property of an object through which it extends its lifetime.
Meta Class	A class of classes is termed as Meta Class.

13.3 OBJECT ORIENTED SDLC (SOFTWARE DEVELOPMENT LIFE CYCLE)

Object-oriented programming became popular 1990s with the increased use of Smalltalk, C++, and other new object-oriented programming languages. To take full advantage of object-oriented features (i.e. classes, inheritance, methods), the Object-oriented SDLC model was developed.

An important feature of Object-oriented (OO) systems is that software objects represent real-world objects. Objects are derived from Classes, and a class hierarchy allows objects to inherit characteristics from parent classes. This allows software object reuse, less coding, encapsulation of functionality, and many other advantages. A major problem that arose with OO programming is that if the Class hierarchy is not properly designed, all the OO advantages disappear. The object-oriented model attempts to properly define and document the Class hierarchy from which all the system objects are created and object interactions are defined.

13.3.1 Objectives of Object Oriented SDLC

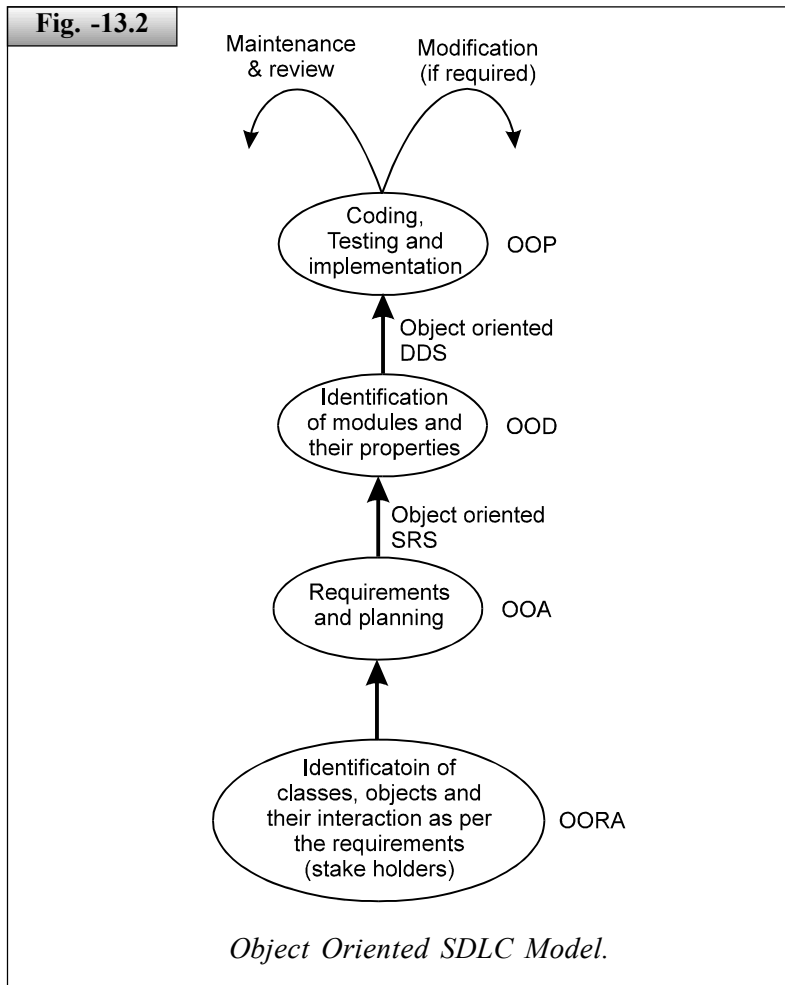
- Define and understand the software development process
- Describe the object-oriented systems development
- List and explain the object-oriented methodologies

The object-oriented SDLC model has these phases that arbitrarily correspond to the traditional SDLC phases noted in brackets:

1. **Object-Oriented Requirements Analysis (OORA)** [Design Analysis]: This is where classes of objects and the interaction between them are defined.
2. **Object-Oriented Analysis (OOA)** [Design Analysis]: In terms of object-oriented concepts, understanding, and modeling a particular problem within a problem domain.
3. **Object-Oriented Design (OOD)** [System Design Specification]: The object is the basic unit of modularity; objects are instantiations of a class.
4. **Object-Oriented Programming (OOP)** [Programming and Testing]: Emphasizes the employment of objects and methods rather than types or transformations, as in other programming approaches.

In the software development process the phases like OOA, OOD and OOP are transformed into the reality, such as analysis, design and implementation.

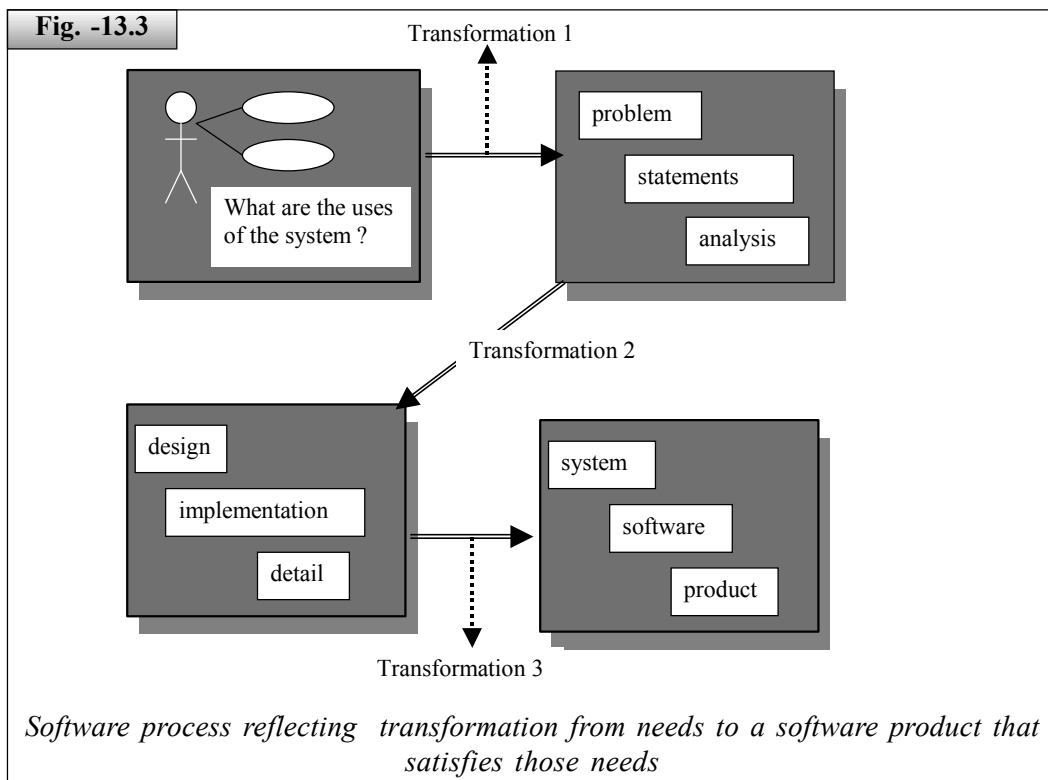
The object oriented SDLC can be viewed as given in the figure 13.2.



13.3.2 The Software Development Process



- System development can be viewed as a process
- Development is a process of change, refinement, transformation, or addition to the existing product
- Generally, the software development process can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation (figure 13.3) :



Transformation : 1 (*Analysis/OOA*) – translates the user’s needs into system requirements & responsibilities

Transformation : 2 (*Design/OOD*) – begins with a problem statement and ends with a detailed design that can be transformed into an operational system

Transformation : 3 (*Implementation/OOP*) – refines the detailed design into the system deployment that will satisfy the user’s needs

- The emphasis on the analysis & design aspects of the software development process, is intended to promote *building high-quality software* (meeting the specifications and being adaptable for change)

13.3.2.1 Object-Oriented Requirements Analysis (OORA)

To develop and object oriented software product, during the requirement analysis phase the following activities are systematically carried out.

- Analyzing from an object-oriented viewpoint
- Step-by-step approach to OORA
- Melding the OORA approach into the traditional analysis approach
- OORA deliverables
- OORA notations and diagrams
- Tools & techniques for OORA
- Guidelines for finding objects
- Developing use-case scenarios

13.3.2.2 Object-Oriented Analysis (OOA)

Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt during object-oriented design (OOD). Analysis is done before the Design

The sources for the analysis can be a written requirements statement, a formal vision document, interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.

It is concerned with

- determining the systems requirements and identifying classes and their relationship to other classes in the problem domain
- identify users/ actors
- use scenarios to understand requirements
- use case to describe the user-computer system interaction to captures user’s goals & needs
- use-case model represents the users’ view of the system or users’ needs
- validate & test the model
- make documentation along the analysis activity (good modeling implies good documentation)

The result of object-oriented analysis is a description of *what* the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up. The purpose of object oriented analysis is to develop a model that describes computer software as it works to satisfy a set of customer defined requirements.

13.3.2.3 Object-Oriented Design (OOD)

Object-oriented design (OOD) transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language.

The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of *how* the system is to be built. The different activities involved in this phase are ;

- ☛ OOD notations and diagrams
- ☛ Step-by-step approach to OOD
- ☛ Steps in preliminary design
- ☛ Steps in detailed design
- ☛ Translating analysis objects into design objects
- ☛ Using inheritance principles
- ☛ Traceability issues
- ☛ Integrating existing software
- ☛ Handling data bases
- ☛ Design matrices & principles

13.3.2.4 Object-Oriented Programming (OOP)

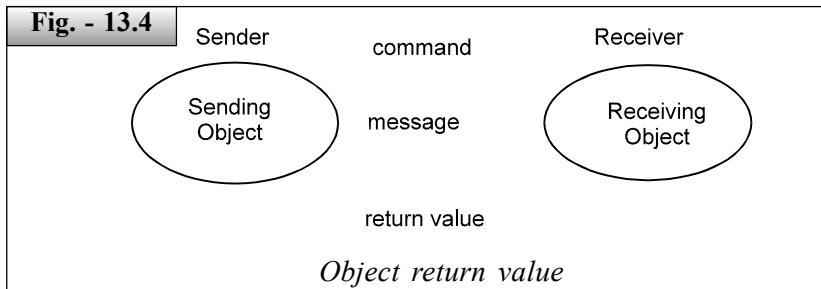
Object-Oriented Programming (OOP) is different from procedural programming languages (C, Pascal, etc.) in several ways. Everything in OOP is grouped as “objects” (see data abstraction). OOP, defined in the purest sense, is implemented by sending *messages* to *objects*. To understand this concept, we first need to know the following.

- Objects and classes
- Instantiation
- Polymorphism
- Inheritance
- Sending messages to objects
- Information hiding

What is An Object?

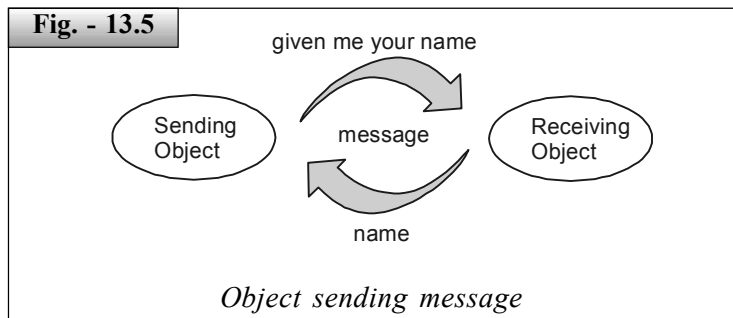
An object can be considered a “thing” that can perform a set of activities. The set of activities that the object performs defines the object’s behavior. For example, a “**StudentStatus**” object can tell you its grade point average, year in school, or can add a list of courses taken. A “**Student**” object can tell you its name or its address.

The object’s interface consists of a set of commands, each command performing a specific action. An object asks another object to perform an action by sending it a *message*. The requesting (sending) object is referred to as *sender* and the receiving object is referred to as *receiver*.



Control is given to the receiving object until it completes the command; control then returns to the sending object.

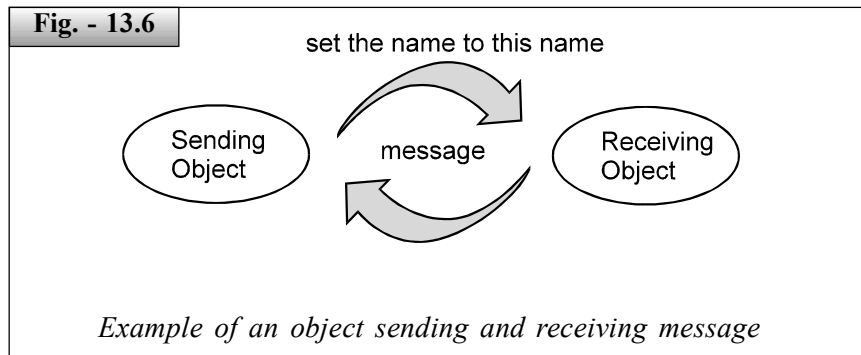
For example, a **School** object asks the **Student** object for its *name* by sending it a *message* asking for its name. The receiving **Student** object returns the *name* back to the sending object.



A message can also contain information the sending objects needs to pass to the receiving object, called the *argument* in the message. A receiving object always returns a *value* back to the sending object. This returned value may or may not be useful to the sending object.

For example, the **School** object now wants to change the student’s name. It does this by sending the **Student** object a message to set its name to a new name. The new address is passed as an argument in the message. In this case, the **School** object does not care about the return value from the message.

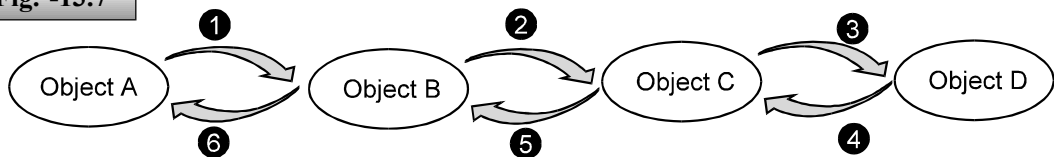
Fig. - 13.6



SEQUENTIAL OPERATION

It is very common that a message will cause other messages to be sent, either to itself or to other objects, in order to complete its task. This is called *sequential operation*. Control will not return to the original sending object until all other messages have been completed. For example, in the following diagram, Object A sends a message to Object B. For Object B to process that message it sends a message to Object C. Likewise, Object C sends a message to Object D. Object D returns to Object C who then returns to Object B who returns to Object A. Control does not return to Object A until all the other messages have completed.

Fig. -13.7



Sequential operation among the objects

Method

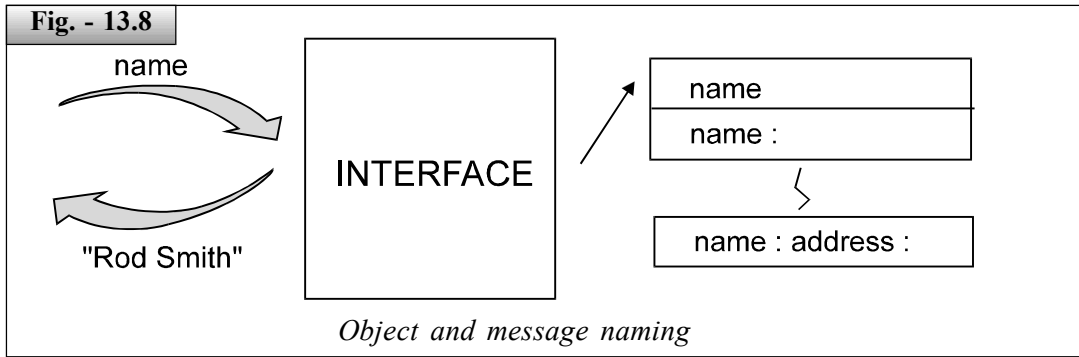
How do receiving objects interpret messages from the senders? How are the messages processed?

Each message has code that associated with it. When an object receives a message, code is executed. In other words, these messages determine an object's behavior and the code determines how the object carries out each message. The code that is associated with each message is called a *method*. The message name is also called the *method name* due to its close association with the method.

When an object receives a message, it determines what method is being requested and passes control to the *method*. An object has as many methods as it takes to perform its designed actions.

Refer to the following diagram, **name**, **name:**, **address** and **name:address** are method names for the **Student** object. When the **Student** object receive the **name** message, the **name** message passes control to the *name* method defined in **Student**.

Fig. - 13.8



Methods that operate on specific objects are *instance methods* and messages that invoke instance methods are called *instance message*. Methods that operate on specific classes are *class methods*. This will be discussed in more details in later chapter.

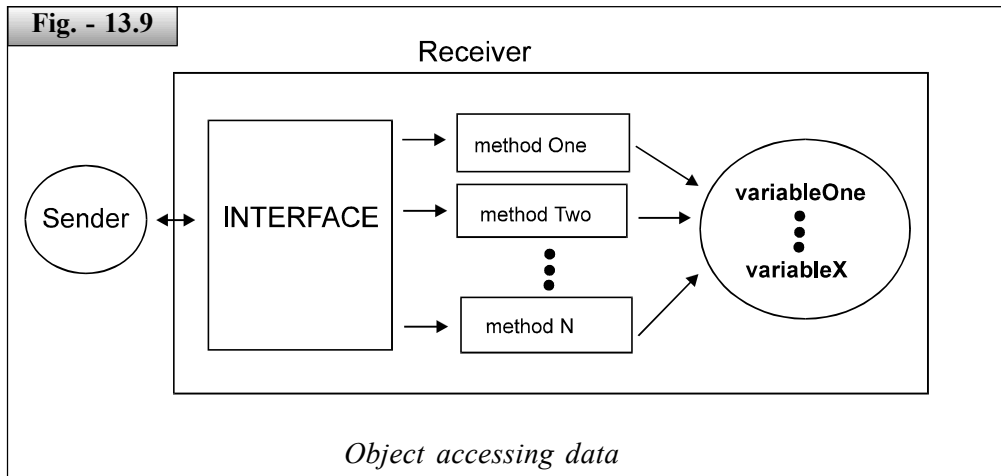
Methods are similar to subroutines, procedures, or functions found in procedural languages (C, Pascal). For example, a method name equates to a subroutine name, and the code for the method equates to the code found in a subroutine. Sending a message to an object is similar to calling a subroutine.

Object's Data

Each object need to keep the information on how to perform its defined behavior. Some objects also contain variables that support their behavior. These variables are called *instance variables*. Only the *instance method* for an object can refer to and change the values stored in the instance variables. The instance methods for other objects cannot refer to this object's data. An object may only access another object's data by sending it messages. This is called *encapsulation* and assures that there is a secure process for getting to an object's data.

Refer to the following diagram, the instance variables *variableOne* through *variableX* can only be accessed by the sender via instance methods *methodOne* through *methodX*. The sender can not refer to the variables directly by itself.

Fig. - 13.9



Unlike procedural programming where common data areas are often used for sharing information, object-oriented programming discourages direct access to common data (other than the use of *global variables*) by other programs. Only the object that “owns” the data can change its constant. Other objects can view or change this data by sending message to the “owner.”

The instance variable names can be identical to the method names that associate with them. For example, the **Student** object has methods of **name**, **address**, and **major** as well as instance variables of *name*, *address*, and *major*. Smalltalk distinguishes variable identifier and a method identifier by the identifier’s position in the expression.

Object-Oriented Problem Solving Approach

Object-oriented problem solving approach is very similar to the way a human solves daily problems. It consists of identifying *objects* and how to use these *objects* in the correct sequence to solve the problem. In other words, object-oriented problem solving can consist of designing objects whose behavior solves a specific problem. A message to an object causes it to perform its operations and solve its part of the problem.

The object-oriented problem solving approach, in general, can be divided into four steps. They are: (1) Identify the problem, (2) Identify the objects needed for the solution, (3) Identify messages to be sent to the objects, and (4) Create a sequence of messages to the objects that solve the problem.

Following is an example of solution to a simple problem using the four steps.

Example 13.1

Object-oriented Problem Solving Approach

Step 1: *Problem Identification* - Compute the sum of two numbers and print out the result.

Step 2: *Object Identification* - Identify objects needed to solve the problem.

Num1 - first number (object).

Num2 - second number (object).

Sum - result of the addition of the two numbers (object).

Step 3: *Message Identification* - Messages needed to send to objects.

+ aNum - This is the message sent to the receiver object with an parameter *aNum*. The result of this message is the value (a numeric object) of the total sum of the receiver object and *aNum*.

print - a message that displays the value of the receiver object.

Step 4: *Object-message sequences* - Following is the object-message sequence to solve the problem.

```
(Num1 + Num2) print
```

The *message* **+** with a parameter **Num2** (an object), is sent to the object **Num1**. The result of this is an object (Num1 + Num2), which is sent the message **print**. The parentheses are included to avoid any ambiguity as to which message should be activated first.

Note: The sequence of Smalltalk evaluation is from left to right unless is specified by parentheses (). For example:

```

           2 + 3 * 4           is 20
while     2 +(3 * 4)         is 14

```

Object-Oriented Paradigm

A computer language is object-oriented if they support the four specific object properties called *abstraction*, *polymorphism*, *inheritance*, and *encapsulation*.

Inheritance??!

Data Abstraction???

Encapsulation?!?!?

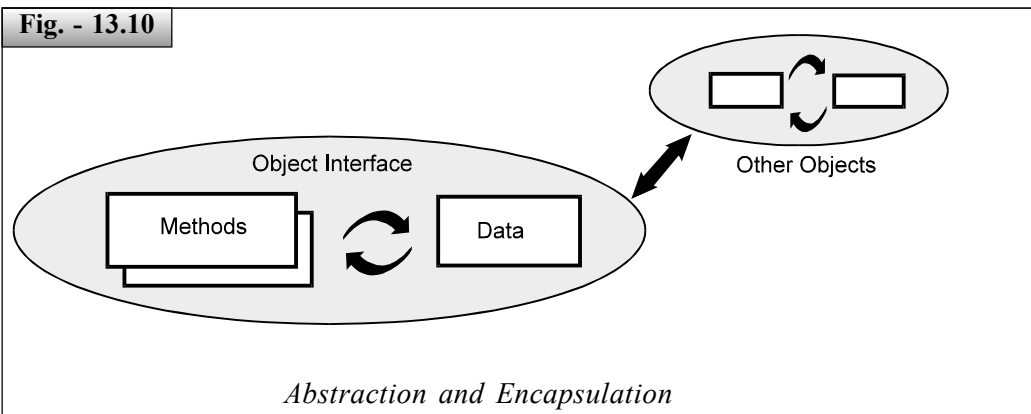
Reuse???

Polymorphism??

Smalltalk??!

The objective of this section is to provide a thorough understanding of the principles of object-oriented paradigm.

- **Data Abstraction**
- **Encapsulation**



In object-oriented programming, objects interact with each other by messages. The only thing that an object knows about another object is the object's interface. Each object's data and logic is hidden from other objects. In other words, the interface *encapsulates* the object's code and data.

This allows the developer to separate an object's implementation from its behavior. This separation creates a "black-box" effect where the user is isolated from implementation changes. As long as the interface remains the same, any changes to the internal implementation is transparent to the user. For example, if the **name** message is sent to the **Student** object, it does not matter to the user how the developer implemented the code to handle this message. All the sending object needs is the correct protocol for interacting with the **Student** object. The developer can change the implementation at any time, but the **name** message would still work because the interface is the same.

● Polymorphism

Another benefit of separating implementation from behavior is *polymorphism*. Polymorphism allows two or more objects respond to the same message. A method called **name** could also be implemented for an object of the class **Course**. Even though the implementation of this name message may return a course number and a course title, its protocol is the same as the **name** message to the **Student** object. Polymorphism allows a sending object to communicate with different objects in a consistent manner without worrying about how many different implementations of a message.

An analogy of polymorphism to daily life is how students response to a school bell. Every student knows the significant of the bell. When the bell (message) rings, however, it has its own meaning to different students (objects). Some students go home, some go to the library, and some go to other classes. Every student responds to the bell, but how they response to it might be different.

Another example of polymorphism is the function of printing. Every printable object must know how to print itself. The message is the same to all the different objects: **print**, but the actual implementation of what they must do to print themselves varies.

The sending object does not have to know how the receiving object implement the message. Only the receiving objects worries about that. Assume that there is a **printPage** method in a **Document** object that has the responsibility of printing a page. To print the page, the **printPage** method sends the **print** message to each object on the page. The **Document** does not need to know what types of objects are on the page, only that each object supports the behavior of printing.

New objects can be added to the page without affecting the **printPage** method. This method still sends the **print** message and the new object provides its own **print** method in response to that message.

Polymorphism allows the sending object to communicate with receiving objects without having to understand what type of object it is, as long as the receiving objects support the messages.

● Inheritance

Another important concept of object-oriented programming is *inheritance*. Inheritance allows a class to have the same behavior as another class and extend or tailor that behavior to provide special action for specific needs.

Let's use the following application as an example. Both **Graduate** class and **Undergraduate** class have similar bahavior such as managing a name, an address, a major, and a GPA. Rather than put this bahavior in both of these classes, the bahavior is placed in a new class called **Student**. Both **Graduate** and **Undergraduate** become *subclass* of the **Student** class, and both *inherit* the **Student** behavior.

Both **Graduate** and **Undergraduate** classes can then add additional behavior that is unique to them. For example, **Graduate** can be either Master's program or PhD program. On the other hand, **Undergraduate** class might want to keep track of either the student is Fresh man, Sophomore, Junior or Senior.

Classes that inherit from a class are called *subclasses*. The class a subclass inherits from are called *super class*. In the example, **Student** is a super class for **Graduate** and **Undergraduate**. **Graduate** and **Undergraduate** are subclasses of **Student**.

The Power of Reuse

One of the most important features of object-oriented programming is the ability to modify existing solution to solve new problems. If a particular kind of problem has been solved using the OOP approach, a similar but slightly different problem can usually be solved by making some changes in the object-message protocol that already exist. Most of the time, this requires adding new *messages*. Other cases may require adding new *objects* and new *messages* to which those objects respond.

Reusability is probably the most important and the strongest feature of Smalltalk. Although *procedural languages* can be reuse too, Smalltalk makes programming for reuse much easier.

13.4 MERITS OF OBJECT ORIENTED SOFTWARE

The object oriented software products has the following potential benefits over the others.

- ☛ Managed Complexity
- ☛ Improved quality
- ☛ Increased Reusability
- ☛ Increased Extensibility
- ☛ Financial benefits
- ☛ Increased chance of project success
- ☛ Reduced maintenance burden
- ☛ Reduced application backlog

13.5 DEMERITS OF OBJECT ORIENTED SOFTWARE

Though the object oriented software products have many more advantages, but these also have the following drawbacks.

- ☛ Requires greater concentration on requirements, analysis and design Developers must work closely with users
- ☛ Requires a complete change in mindset on the part of individuals
- ☛ Provide only long term benefits
- ☛ Demands up-front investments in training and tools Insists more on testing
- ☛ Still the success is greatly depend on people involved.

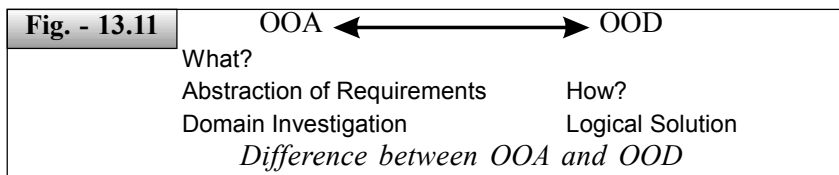
13.6 DIFFERENCES BETWEEN OOA AND OOD

OOA and OOD are simply at opposite ends of the same activity (see Figure 13.11). OOA identifies the requirements; investigate the domain and ends up with identifying set of objects. OOA is independent of implementation. OOD provides how those identified objects are implemented. OOD is closely associated with the implementation phase.” OOA exposes the components of the required behaviors, for implementation. On the other

hand, OOD hides the details of the implementation so that those details do not corrupt the rest of the design.

There is no good dividing line, which separates them, either in time, or in function. OOA and OOD can be synonymously related to the different colors in a spectrum. They are certainly different, but it is difficult to tell when, one starts and the other leaves off. OOAD is a continuum of activity, which incorporates both analysis and design.

OOA and OOD are the best done concurrently. Analysis and Design should not be separated into broad phases. The analysis model should not be frozen before the design modeling begins. Both are closely related, and that they cooperate synergistically. The analysis model cannot be completed in the absence of the design model. Also it may be noted that the design model cannot be completed in the absence of the analysis model. One of the most important aspects of OOAD is the synergy between the two concepts.



13.7 OOPS PROGRAMMING LANGUAGES

There are almost two dozen major object-oriented programming languages in use today. But leading commercial languages are fewer in number. This page covers just a few of the most popular. These are:

C++ , Smalltalk, Java,

C++

C++ is an object-oriented version of C. It is compatible with C (it is actually a superset), so that existing C code can be incorporated into C++ programs. C++ programs are fast and efficient, qualities which helped make C an extremely popular programming language. It sacrifices some flexibility in order to remain efficient, however. C++ uses compile-time binding, which means that the programmer must specify the specific class of an object, or at the very least, the most general class that an object can belong to. This makes for high run-time efficiency and small code size, but it trades off some of the power to reuse classes.

C++ has become so popular that most new C compilers are actually C/C++ compilers. However, to take full advantage of object-oriented programming, one must program (and think!) in C++, not C. This can often be a major problem for experienced C programmers. Many programmers think they are coding in C++, but instead are only using a small part of the language's object-oriented power.

Smalltalk

Smalltalk is a pure object-oriented language. While C++ makes some practical compromises to ensure fast execution and small code size, Smalltalk makes none. It uses run-time binding, which means that nothing about the type of an object need be known before a Smalltalk program is run.

Smalltalk programs are considered by most to be significantly faster to develop than C++ programs. A rich class library that can be easily reused via inheritance is one reason for this. Another reason is Smalltalk's dynamic development environment. It is not explicitly compiled, like C++. This makes the development process more fluid, so that "what if" scenarios can be easily tried out, and classes definitions easily refined. But being purely object-oriented, programmers cannot simply put their toes in the o-o waters, as with C++. For this reason, Smalltalk generally takes longer to master than C++. But most of this time is actually spent learning object-oriented methodology and techniques, rather than details of a particular programming language. In fact, Smalltalk is syntactically very simple, much more so than either C or C++.

Unlike C++, which has become standardised, The Smalltalk language differs somewhat from one implementation to another. The most popular commercial "dialects" of Smalltalk are:

- VisualWorks from ParcPlace-Digitalk, Inc.
- Smalltalk/V and Visual Smalltalk from ParcPlace-Digitalk Inc.
- VisualAge from IBM
- VisualWorks

VisualWorks is arguably the most powerful of Smalltalk's. VisualWorks was developed by ParcPlace, which grew out of the original Xerox PARC project that invented the Smalltalk language. VisualWorks is platform-independent, so that an application written under one operating system, say, Microsoft Windows, can work without any modification on any of a wide range of platform supported by ParcPlace, from Sun Solaris to Macintosh. VisualWorks also features a GUI (Graphic User Interface) builder that is well-integrated into the product.

Smalltalk/V and Visual Smalltalk

Digitalk's versions of Smalltalk are somewhat smaller and simpler, and are specifically tailored to IBM compatible PCs. A Macintosh version was available, but support has since been abandoned. This does not bode well for Digitalk cross-platform efforts. Digitalk has a separate GUI builder, called PARTS Workbench (bundled with Visual Smalltalk), which allows quick construct of an application.

ParcPlace and Digitalk were merged into a single company, ParcPlace-Digitalk, Inc. The future of the Digitalk product line is uncertain, and it may just be spun off back into a separate company.

VisualAge

IBM's version of Smalltalk, VisualAge, is comparable to Smalltalk/V with PARTS. Both of these Smalltalk's allow programmers to readily exploit machine-specific features, at the expense of some portability. IBM has adapted existing industry standards for such things as file management and screen graphics. When IBM talks, people listen, and IBM has made a substantial commitment to Smalltalk.

Java

Java is the latest, flashiest object-oriented language. It has taken the software world by storm due to its close ties with the Internet and Web browsers. It is designed as a portable language that can run on any web-enabled computer via that computer's Web browser. As such, it offers great promise as the standard Internet and Intranet programming language.

Java is a curious mixture of C++ and Smalltalk. It has the syntax of C++, making it easy (or difficult) to learn, depending on your experience. But it has improved on C++ in some important areas. For one thing, it has no pointers, low-level programming constructs that make for error-prone programs. Like Smalltalk, it has garbage collection, a feature that frees the programmer from explicitly allocating and de-allocating memory. And it runs on a Smalltalk-style virtual machine, software built into your web browser which executes the same standard compiled Java byte-codes no matter what type of computer you have.

Java development tools are being rapidly deployed, and are available from such major software companies as IBM, Microsoft, and Symantec.

SUMMARY

The need of object oriented software engineering, and the OO concepts are well explained. Potential benefits and drawbacks of using OO technology are highlighted. The object oriented system development life-cycle and OO languages are also dealt.

EXERCISE

1. In what way does the object oriented approach claim to improve the system development process?
2. Define the principle terminologies used in object-oriented approach.
3. Demonstrate object oriented concepts for Banking System.
4. Briefly explain the difference between the following pairs of terms with suitable Examples
 - a) Class / Object
 - b) Encapsulation / Data hiding
 - c) Super class / sub class
5. Describe OOSDLC
6. What are the benefits and drawbacks of using OO in software development?
7. List few object languages and compare them.



Software & Tools

14.1 INTRODUCTION

The history of software tools began with the first computers in the early 1950s that used linkers, loaders, and control programs. Tools became famous with Unix in the early 1970s with tools like `grep`, `awk` and `make` that were meant to be combined flexibly with pipes. The term “software tools” came from the book of the same name by Brian Kernighan and P. J. Plauger.

Tools were originally simple and light weight. As some tools have been maintained, they have been integrated into more powerful integrated development environments (*IDEs*). These environments consolidate functionality into one place, sometimes increasing simplicity and productivity, other times sacrificing flexibility and extensibility. The workflow of IDEs is routinely contrasted with alternative approaches, such as the use of Unix shell tools with text editors like Vim and Emacs.

The distinction between tools and applications is murky. For example, developers use simple databases (*such as a file containing a list of important values*) all the time as tools. However a full-blown database is usually thought of as an application in its own right.

For many years, computer-assisted software engineering (*CASE*) tools were sought after. Successful tools have proven elusive. In one sense, CASE tools emphasized design and architecture support, such as for UML. But the most successful of these tools are IDEs.

The ability to use a variety of tools productively is one hallmark of a skilled software engineer.

Broadly, a tool is any device that, when used properly, improves the performance of the task.

In general the tools are grouped in to the categories of analysis, design and development tools.

14.2 ANALYSIS TOOLS

Analysis tools assist systems specialist in documenting an existing system, whether manual or automated, and determining the requirements for a new application activities. These tools include :

i) Data Collection Tools

It captures the details describing current systems and procedures. It also makes documentation of the processes and decision activities. These tools are used in requirements identification phase.

ii) Charting Tools

These tools create graphic representations of systems and activities. These also assist in the *drawing* and *revision* of Data Flow Diagrams (*DFD*) and *icons* associated with structured analysis. Also include flow charting programs.

iii) Dictionary Tools

These tools help to record and to maintain descriptions of system elements, such as *data items*, *processes*, and *data stores*. It also often provides the capability to examine system descriptions for inconsistent or incomplete descriptions. These also include the capability to report where items are used.

The most useful tools in each of these categories are becoming automated, both to improve the efficiency of the analyst and to make the results of the analysis effort more accurate and complete.

14.3 DESIGN TOOLS

Design tools assist in formulating the features of a system that will meet the requirements outlined during the analysis activities.

i) Specification Tools

These tools assist in commencing the features that should be included in an application, such as input, output, processing and control specifications. These also include tools for creating data specifications.

ii) Layout Tools

These tools are used to describe the position of data, message and headings on display screens, reports, and other input/output media.

System analysts have used tools for the design of systems since early days of computing. However, the recent infusion of computer assistance and powerful graphics is giving new meaning to system design.

14.4 DEVELOPMENT TOOLS

Development tools aid the analyst in translating design into functioning applications :

i) Software Engineering Tools :

These tools assist in formulating, software designs, including procedures and controls, as well as documentation for the design.

ii) Code generators :

Code generators provides or produce source code and working application from functional specifications that are well articulated.

14.5 TOOLS FOR SPECIAL PURPOSES

Apart from the above tools, so many tools are used for special purpose of use or application during the software development.

These are :

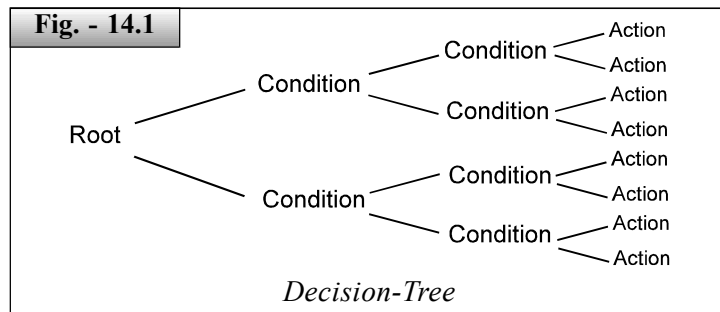
- 1) Tools for Documenting procedure and Decision making.
- 2) Tools for Data flow strategy.
- 3) Tools for proto-typing.
- 4) CASE Tools.(detail in Chapter-11)

14.5.1 Tools for Documenting procedure and Decision making

Two types of specialised tools are required for decision making process. These are decision tree and decision table.

a) Decision-Tree

A decision tree is a diagram that presents *conditions* and *actions* sequentially and thus shows which conditions to consider first, which second and so on. It is also a method on showing the relationship of each condition and its permissible actions.



The above diagram resembles branches on a tree.

The left of the diagram resembles the *root* of the tree i.e. the starting point of the decision sequence. The particular branch depends on the conditions that exist and the decision to be made. Progression from left to right along a particular branch is the root of making a series of decisions.

The next following each decision point is the next set of decisions to be considered. The nodes of the trees represents conditions and indicate determination must be made about which condition exists before the next path is chosen.

The right side of the tree resembles the actions to be taken, depending upon the sequence of conditions be followed.

c) Decision Table

Decision tables provides a mechanism for recording complex decision logic. Decision

tables are mostly used in data processing applications and also have an extensively developed literature.

Table-14.1 : Basic Elements of a Decision Table

	<i>Decision rules</i>			
	Rule 1	Rule 2	Rule 3	Rule 4
Condition statement		Condition entries		
Action statement		Action entries		

The above table shows the basic Elements of Decision Table and how Decision table is segmented into four quadrants i.e.

- Condition statement
- Condition entry
- Action statement
- Action entry.

The condition statement contains all the conditions which are examined. Condition entries are used to combine conditions into decision rules. The action statement describes the action to be taken in response to decision rules and the action entry quadrant relates decision Rules to actions.

Table-14.2 : Limited entry Decision Table

	1	2	3	4
Credit limit is satisfactory	Y	N	N	N
Pay experience a favorable	-	Y	N	N
Special clearance is obtained	-	-	Y	N
Perform approve order	X	X	X	
Go to reject order				X

The above table describes the format of limited entry decision table i.e. entries are limited to Y, N, – 2nd X.

In limited entry decision table	Y	denotes YES
	N	denotes No
	–	denote don't care
	X	denotes perform action.

According to above table orders are approval if the credit limit is not exceeded, or if the credit limit is exceeded but poor experience is good or if a special arrangement is been made. If none of the above conditions holds then the order is rejected.

The (Y, N, –) entries in each column of the condition entry quadrant from a decision rule, if more than one decision rule has identical (Y, N, –) entries the table is said to be ambiguous.

Ambiguous pairs of decision tree rules that specify identical actions are said to be redundant and those specifying different actions are contradictory. Contradictory Rules permit specification of non deterministic and concurrent Actions.

Table-14.3 : An Ambiguous Decisions table

	<i>Decision rules</i>			
	Rule 1	Rule 2	Rule3	Rule 4
C1	Y	Y	Y	Y
C2	Y	N	N	N
C3	N	N	N	N
A1	X			
A2		X		
A3			X	X

The above table describes the redundant rules (R3 and R4) and contradictory Rules (R2 and R3 and R2 and R4)

A decision table is said to be complete if every possible set of conditions has a corresponding prescribed action. There are 2^N combinations of conditions in a table that has N condition entries.

If any one of combinations results fails, it results in an incomplete decision table.

Table-14.4 : An incomplete and Over-specified Decision Table

C1	Y		N
C2		Y	N
C3			Y
A1		X	
A2	X		
A3			X

The above table shows an incomplete and over specified Decision table.

In the table the combination (N, N, N) for conditions C1, C2, C3 has no action specified. Also the condition (Y, Y, N) specifies both Actions A1 and A2. This multiply specified action may be desired or it may indicate a specification error.

The above table also illustrates the use of Karnaugh Map to check a decision table for completeness and multiply specified actions. The specification is said to be incomplete if there are any blank entries in the Karnaugh Map. The specification is multiply specified if there are any multiple entries in the Karnaugh Map.

Advantages of Decision table

The various advantages of decision table are :

- Decision Rules are clearly structured.
- Manager's can be relieved from Decision Making
- Consistency in decision making
- Communication is Easier between manager and analysis.
- Documentation is easily prepared, changed or updated.
- Easy to use.
- Easier to draw or modify in comparison to flow charts.
- Facilitate more compact documentation.
- Easier to follow a particular path down one column than through complex and lengthy flowcharts.

Disadvantages

- Impose an additional burden
- Does not depict the flow
- Not easy to translate it
- Can't list all the alternatives

14.5.2 Tools for Data Flow strategy or Data Flow Analysis

Data flow analysis examines the use of data for a specific business process. Data Flow strategy is applicable in both requirement determination and system design phase. Therefore several number of tools are taken into the help to improvise the task requirement and system designing as follows :

It also permits the system analysis to isolate areas of invert in the organization and study them by examining the data that enter the process and seeing how these are changed when they leave the process. The tools for data flow analysis are

- a. Data Flow Diagram (DFD)
- b. Data Dictionary
- c. Structure chart / program structure chart.
- d. Data Structured Diagram

a) Data Flow Diagram (DFD)

The DFD is a graphic tool which is used to describe and analyse the movement of data through a system (Manual or automated). It also includes the activities like data process, Data store and Data Flow.

This tool is an essential component for structured Analysis and design through which other components are developed.

The transformation of data from input section to output section through the processing section may be described logically and independently of the physical components associated with the system. These physical components are for e.g. computers, file cabinets, disk units and word processors.

The transformation is performed by using the logical DFD or physical DFD. These are discussed in structured Analysis and Design.

Structured Analysis and Design

Structured analysis referred as structured system analysis which is a methodology that consists of a brief survey of the problem domain or organization followed by structured analysis. This step helps in co-insisting the SRS completely. This step is usually followed by structured design.

It may be relevant to recall the distinction between process and methods. Software process is a framework for the tasks that are required to build systems of high quality. Software methods provide the technical ways (sort of “*how to*”) of developing (or constructing) the software.

- **Advantages of Structured Analysis and Design**

- ☛ It results in good Documentation
- ☛ it provides for predictability of design.
- ☛ It aids in Maintainability of the product / project and also making enhancements.
- ☛ It serves is a medium of communication among various stakeholders in the system.
- ☛ It makes it structured for an organizational approach to development.
- ☛ It provides for easy resting and debugging.
- ☛ It aids in reducing the complexity of the system through modularization.

- **Is structured Analysis a unique Approach ?**

There are many approaches suggested but all have some common sets of characteristics. A possible classification of approaches to structured analysis came from C.H.P. Brookes and others in 1982.

1. *Checklist Based Methods*

These methods are primarily used for data processing situations. A set of steps required to be carried out is check listed (hence the name). Also documented are a checklist of do's and don'ts.

2. *Form based methods :*

One of the earliers methods used is the form based approach which is required to strandardize procedures. e.g. Approaches are IBM's Study Organisation Plan (SOP) and Hierarchial Input Process Output (HIPO).

3. *Data structure Based Methods*

These methods are developed on the premise that the structure of a program ought to closely follow the structure of the data being dealt with. This method was very popular with approaches such as the Warnier or method and the Jackson System Development Method.

4. *Data Flow Based Methods :*

Since a computer based system has to handle both data and logic, data is studied in the form of I/O of the system or as files. The method requires the development of Data Flow Diagram (DFD). This approach was developed in 1970's by Coad and Constanvine, TOM De Marco, Page Jones and Myer's were involved for the development of this approach.

5. *Automated Methods :*

Automated methods attempt to provide assistance in getting much of the tedious amount of documentation done automatically. Tools such as Problem Statement Language (PSL) and the Problem Statement Analysis (PSA) took a decade to develop but are in promotive stages for supporting automation.

Table-14.5 : History of Structured Analysis

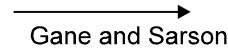
Year(s)	contribution to structural Analysis.
1974	Strevens, Myrs and constrantine on notations for structured design.
1978	Yourdon and constratine on structured design.
1979	Tom De Marco on structured System Analysis and design which introduced key graphical symbols.
1980	Page Jones on some variations of the Structured Analysis.
1982	Gane and sarson on refinement of notations.
1985	Ward and Mellor on extensions to handle red time systems.
1987	Hartly and pirbhai on extensions for handling real fime and control systems.
1988	Bruyn and others in an effort to develop one consistent notation for structured analysis.
1989	Coad and yourdon method which is a modern treatment of structured analysis with the use of case tools.

● **Notation for data flow oriented analysis**

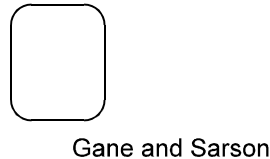
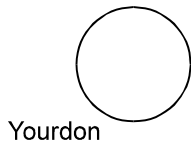
- Data flow analysis methods were developed and promoted by two oragnizations simultaneously :
- A consulting and professional development firm named as Yourdon, Inc., has publicly promoted the methods and also in books (Weinberg, Demarco, Page-Jones).
- Through the work and writing of Game and Sarson, McDonwell Douglas, has influenced the popularity of data flow analysis.

- By using only four simple notations logical data flow diagrams can be completed, i.e., special symbols or icons and the annotation that associates them with a specific system.
- Depending upon the Yourdon or Gane and Sarson approach, specific icons associated with each element are used :

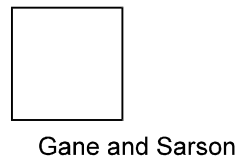
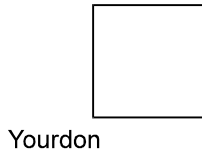
1. Data Flow : It is nothing but the Data move in a specific direction from an origin to a destination in the form of a document, letter, telephone call, or virtually any other medium. The data flow is a “packet” of data.



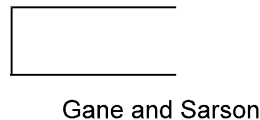
2. Processes : The physical component of the process is not identified. We can define the processes as the people, procedures, or devices that use or produce or transform data.



3. Source or destination of data : External sources or destinations of data, which may be people, programs, organizations, or other entities, interact with the system but are outside its boundary. The term source and sink are interchangeable with origin and destination.

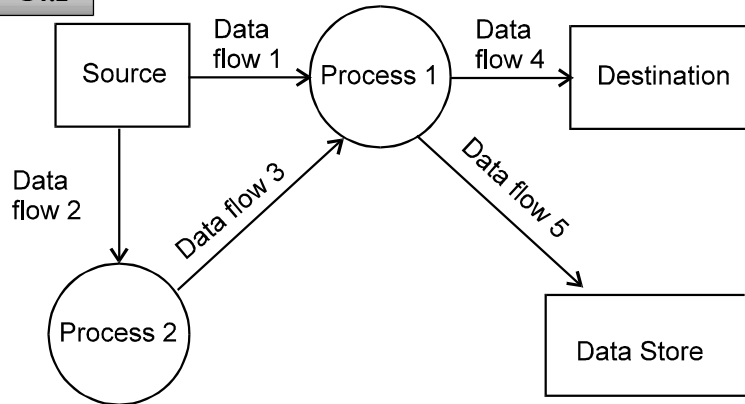


4. Data store : The data store may represent computerized or non-computerized devices. Here data are stored or referenced by a process in the system.

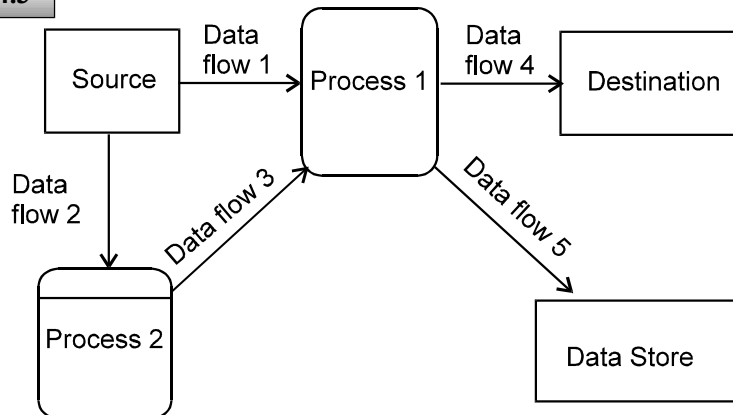


- In a data flow diagram, each component is labeled with a *descriptive name*.
- Process names are further identified with a *number* that will be used for identification purposes.

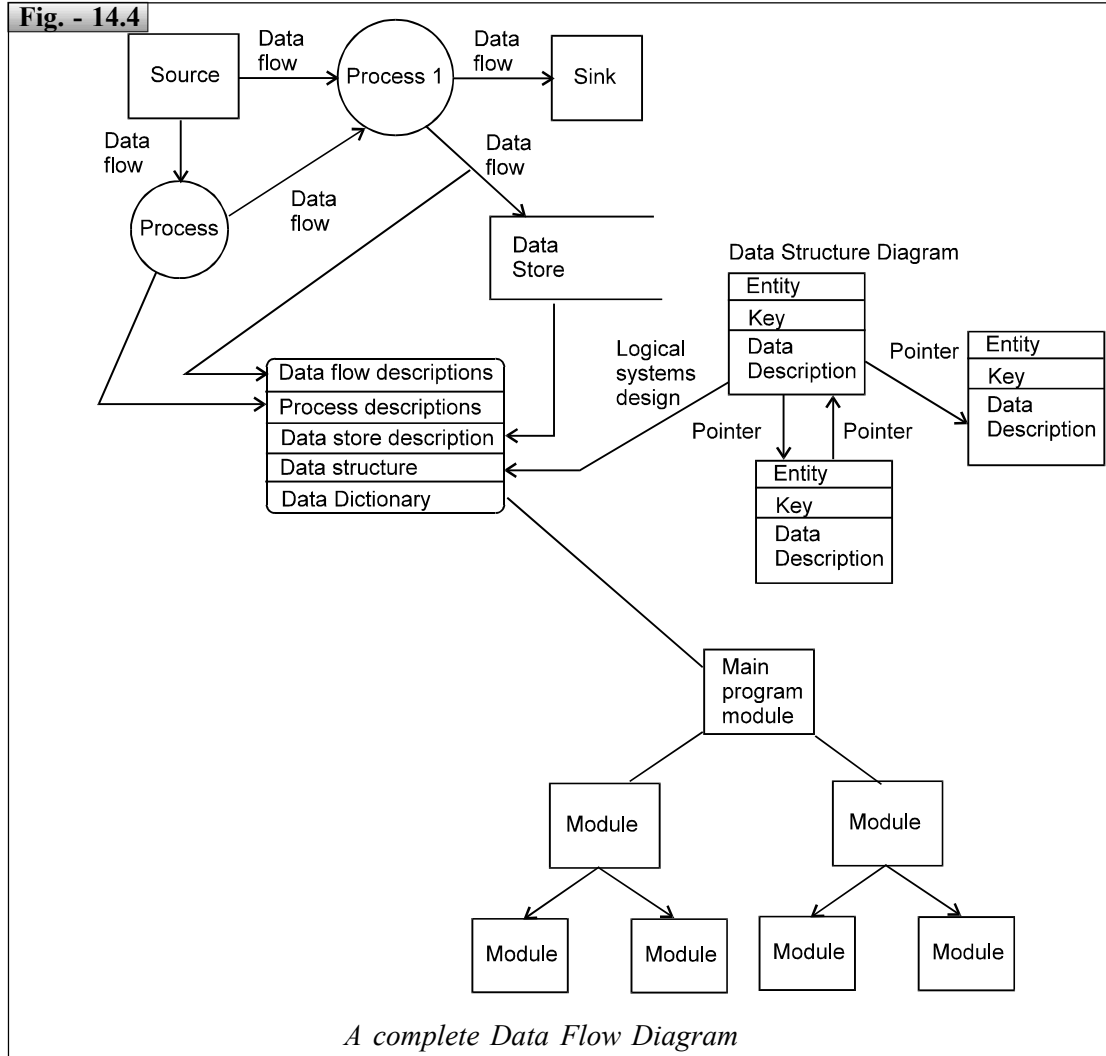
- The *number* assigned to a specific process doesn't represent the sequence of processes.
- It is strictly for identification and will take on added value when we study the components that make up a specific process.

Fig. - 14.2*Data flow diagram using Yourdon notation*

- The figure 14.2 shows the Yourdon notation in a simple data flow diagram consisting of five data flows, two processes, a data store, a source, and a sink.

Fig. - 14.3*Data flow diagram using Gane and sarson notation.*

The figure 14.3 shows the same elements using the Gane and Sarson notation.

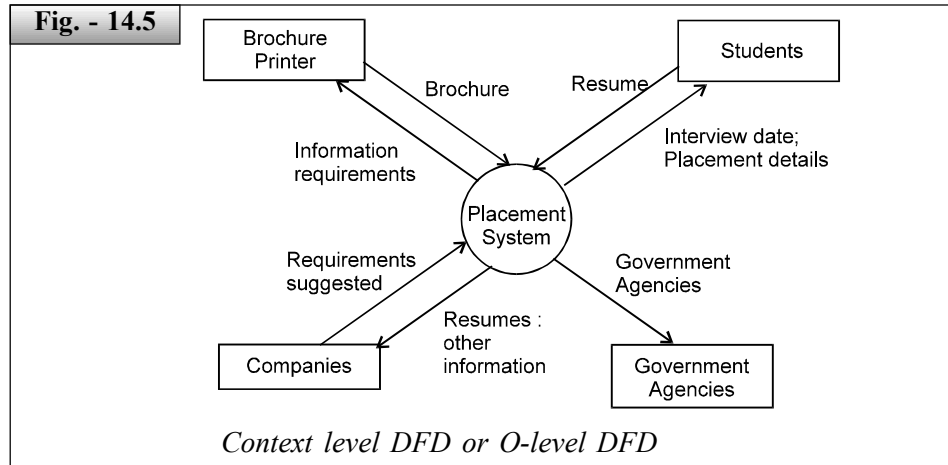
Examples of a data flow diagram

- **Leveling of DFD**

- We can use the *DFD*, when we want to represent a system or a piece of software at any level of abstraction.
- We can begin at the top-most level where only the bare minimal details of information flow and process details are found.
- The *DFD* at this level may be called *level 0 DFD* and is referred to as the '*Context Level DFD*'.
- In a *DFD*, processes are represented as bubbles (*circles with the names of the processes or transformations*).
- A line represents a data flow with names of data items, with an arrowhead indicating the direction of flow.

- The entire system is represented by just one process (*and so one bubble*) at *level O DFD* or the *context level DFD*.
- In this, all data flows from and to the external entities are depicted.
- External entities themselves are shown within rectangular boxes.

As an illustration, consider a placement system on campus for a university. The *level O DFD (context level DFD)* is shown in fig. 14.5.

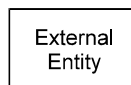


At this Level O, the DFD just places the system in context and depicts external entities and basic data flows.

- We can refine the level O DFD to a higher level, called the level 1 DFD.

Here, the single bubble representing the whole process is refined into further details involving other processes. Level 1 DFD may contain 3 to 8 bubbles representing processes, with interconnecting arrows as data flows. In addition, the level 1 DFD may also contain a “*data store*” which is a data repository i.e. to be stored for use by one or more processes. The data store can represent something as simple as a buffer, a queue, a flat file or even a sophisticated RDBMS. We now have nearly all the notations required for DFD. i.e.

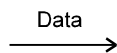
Fig. - 14.6



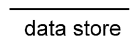
This represents a producer or consumer of information that is notionally outside the boundary of the system.



This is a transformation of information that is inside the system itself and must be modeled.



This represents a data flow, with the arrow head representing the direction of the data flow.



This represents a repository of data that allows processes to store data inside the system.

Notations for DFD

In order to illustrate the refinement of a context level DFD into level 1 DFD, we once again look at a university placement system.

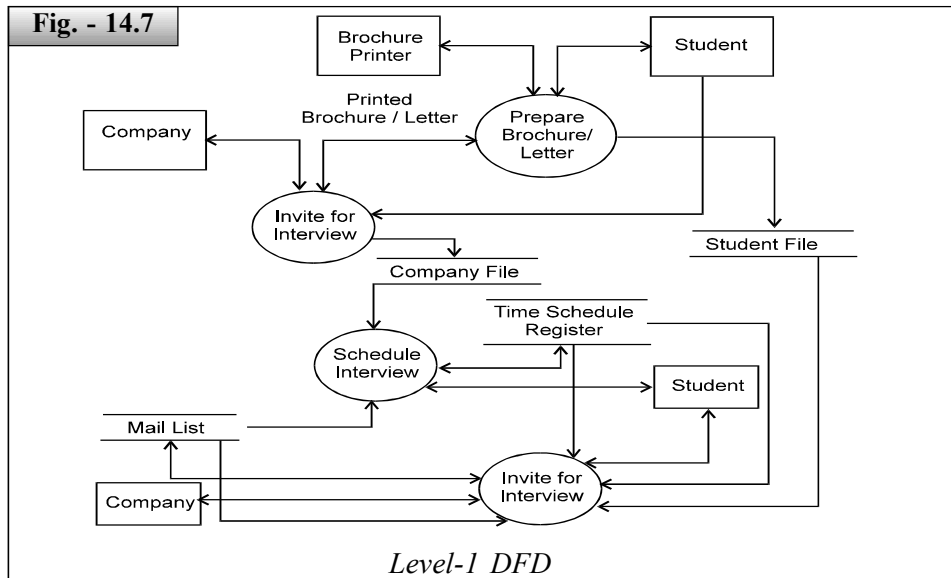
Universities arrange for companies to visit the campus, interview students and select them for employment. A placement officer prepares information material such as brochures, letters and profiles of potential candidates available for employment and sends them to companies. Some of these companies discuss with the placement office suitable dates for interviews, giving details about their activities, profiles of persons being arranged. Selected candidates are offered jobs. The company as well as the placement officer work with a mutually agreed upon set of policies.

Given figure 14.7 is the level 1 DFD of the placement system.

It should be noted that the level 1 DFD shows the important subsystems of the placement system. It shows various processes and data flows. The following are some of the main processes shown in this level 1 DFD.

- Process 1 : Preparation of brochures and letters.
- Process 2 : Invitation for interviews
- Process 3 : Interview schedule
- Process 4 ; Selection of students

Both the “*Invitation for Interviews*” and “*Interview Schedule*” processes access a data store called “Mail List”. In a similar manner, Processes “*Interview schedule*” and “*Selection of Students*” make use of the data store file-2 or the “*Time schedule Register*”. In level 1 DFD, we show all the external entities as shown in level DFD. This is required for consistency. We must ensure that no new external entities are introduced or existing ones dropped in any of the other levels of the DFDs. beyond level 1, it is conventional not to show external entities explicitly. A convention used in DFD is to show only those data stores that are accessed by at least two processes. However, if the analyst feels that a data store is very important, but is accessed only by one single process, it may be shown.



Now we move on to develop level 2 DFDs. In level 2 DFDs, we need to note a few important points.

- The external entities aren't shown explicitly. Sometimes they can be shown implicitly by qualification with respect to the entity using "from" and "by".
- There must be some consistency among the incoming and outgoing arrows with the previous level (level 1 in this case)
- The same processes in the previous level may appear, as they were, in this level too.
- processes in the previous level may be refined into more detailed processes and data flows.

Figure below shows the level 2 DFD for process 1, i.e. "*Prepare Brochures and Letters*".

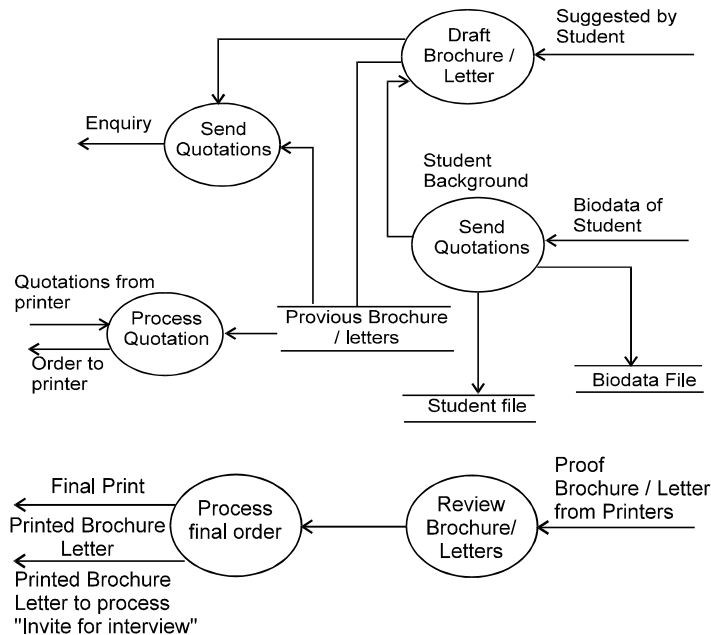
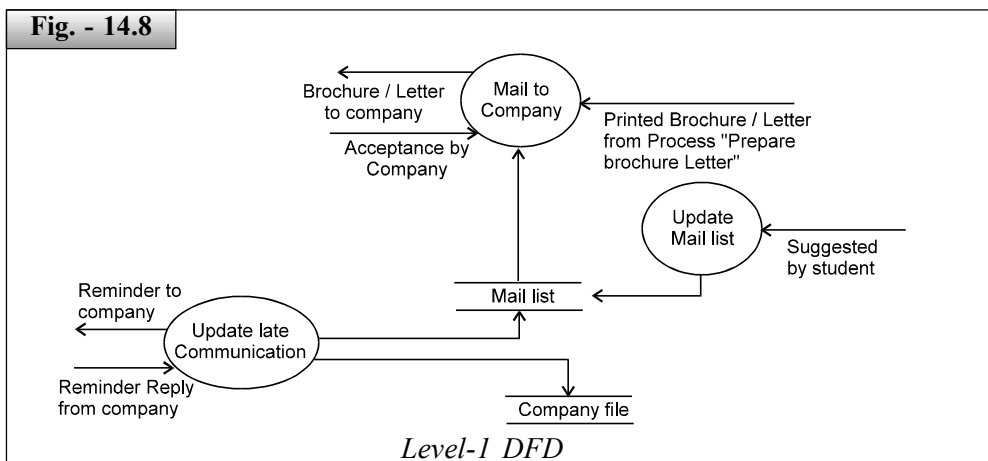


Figure 14.8 shows the level 2 DFD for the process 2, i.e. "*Invitation for Interview*".



The Processes that are not further refined in higher levels are also called *primitive processes*. Figure 14.9 shows the level 2 DFD for process 3, i.e. “*Interview Schedule*”.

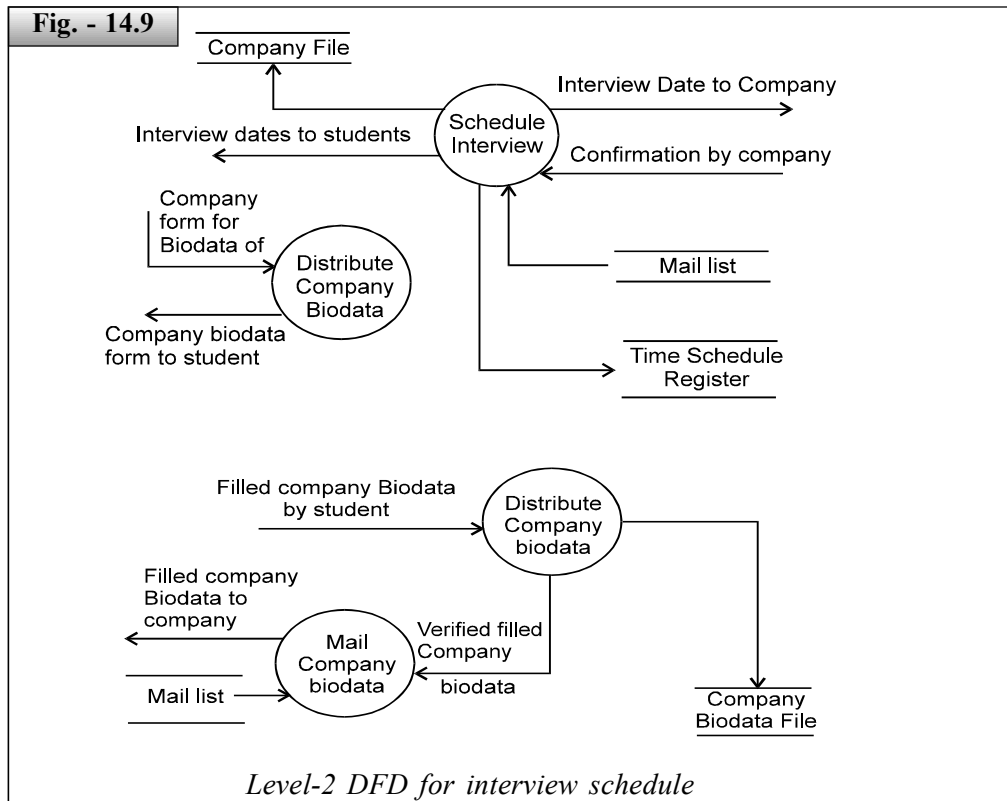
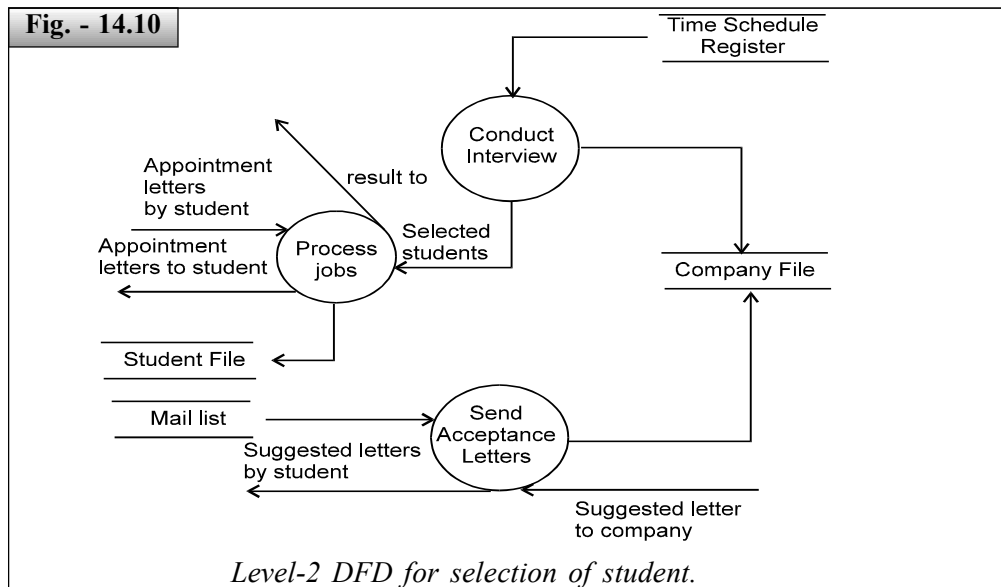


Figure 14.10 shows the level 2 DFD for process 4, i.e. “*Selection of Student*”.



No further explanations are given for these, as they are fairly self-explanatory. As we refine the DFDs across various levels, we must note that “*information flow continuity*” must be maintained, i.e. input and output at each level of refinement must remain the same. This is also called balancing which helps in developing consistent models.

Many beginners confuse a DFD with a flowchart. *A DFD depicts information flow without explicit representation of things like conditions, loops, etc. In other words, a DFD doesn't depict procedure logic.*

A basic DFD (even at multiple levels) is not sufficient to describe the requirements of software. What can a DFD describe? Arrows into and out of a process bubble represent data items that are input and output. A data store represents some organized collection of data. But what a DFD does not depict is the content of the data items. To get this, we need to introduce another component. This is called the *data dictionary*. Some authors also refer to this as the *requirements dictionary*.

b) Data Dictionary

- A data dictionary is a *catalog* or a *repository* of the elements in a system.
- As the name suggests, these elements center around data and the way they are structured to meet user requirements and organizations needs.
- In a data dictionary you will find a list of all the elements such as *data flows, data stores, and processes* etc. for composing the data flowing through a system.
- The data dictionary stores details and descriptions of these elements.
- If analyst want to know how many characters are in a data item, by what other names it is referenced in the system, or where it is used in the system, they should be able to find the answers in a properly developed data dictionary.
- The dictionary is developed during *data flow analysis* and assists the analysts involved in determining *systems requirements*.

● Why is a Data Dictionary Important ?

Analysts use data dictionaries for five important reasons :

- ☛ In order to manage the detail in large system.
- ☛ In order to communicate a common meaning for all system elements.
- ☛ In order to document the features of the system.
- ☛ In order to facilitate analysis of the details in order to evaluate characteristics and determine where system changes should be made and determine where system changes should be made.
- ☛ In order to locate errors and omissions in the system.

A data dictionary is nothing but the data about data which describes the entries in the DFD that are labeled as data items along the arrowed lines. The following are part of the information recorded in a data dictionary.

- Data flows and its components, i.e. input, output and intermediate data.
- Data stores, like files
- Processes

- Any other relevant information that helps in writing the requirements specifications better.

A data dictionary is a supplementary documents to be used with the DFD. There is one entry in the data dictionary corresponding to each of the data flows, files and processes.

It is important that we understand the structure of data entries in a DFD. Data entries in a data dictionary should be precise, easy to describe and easy to understand. We could adopt a natural language representation, say in ordinary English, for the data dictionary. However, a symbolic representation is better.

Studies have shown that structural programs have three basic structures.

- Sequence

A sequential set of instructions.

- Selection

Decision constructs such as `if-then`, `case`, etc.

- Iteration

Constructs to perform looping, like `for`, `do while`, etc.

Some authors like Bohm and Jacopini have mathematically shown that the above three constructs are sufficient to develop algorithms for certain classes of problems.

In the context of a data dictionary, we can use the following to represent data.

- Sequence - It is a concatenation of two or more components in some specific order.
- Selection - It represents a choice of one of two or more data items.
- Iteration - To represent a repetition of some designated components zero, one or more items.

To define data items, we can use the following symbols.

= This means "is equivalent to"

+ This means to add

[] This means "either or"

{ }_a^b This means "iteration of"

Here a and b are lower and upper limits, respectively.

() This means "optional"

Studies have shown that the above symbols are sufficient to describe data items in data dictionaries.

As a simple illustration, consider the following data entries in a data dictionary.

Invoice - Construct = {Invoice - Item}₁⁴

Invoice - Item = Item - ID + Unit Price + Quantity +

Item - Value + (User - Order)+ [Warehouse - ID Dealer ID]

This means that Invoice - construct contains multiple (up to 4) Invoice - Items. And Invoice - Item consists of Item - ID, Unit - Price, etc. The Invoice - Item would consist of either a Warehouse - ID or Dealer - ID.

The data dictionary entries for the university placement system are quite large in number, i.e.

Fig. - 14.11

Data Flow Name :	University Biodata From filled by student
Composition :	Unstructured
Note :	Student's own format
Data Flow Name :	Company Biodata From
Composition :	Unstructured
Note :	This data dictionary entry also defines Filled Company Biodata
File Name :	Mail List
Composition :	{Company – Code + Company – Name + Company – Address}
Note : This file is manually maintained in alphabetic order on index cards.	

The data dictionary must define data items unambiguously. It must be noted that the data dictionary tends to grow too rapidly in size and complexity for larger systems. Unless we have some sort of automated support, it would become tedious and complex. CASE tools provide such automated support.

Writing Specification of Processes

Usually, another document, called process specifications, is also provided to supplement the DFD. This describes the processes in detail. It is called either PSPEC or MiniSpec.

There would be one PSPEC entry for each primitive process. Since the processing logic of processes at higher levels is simply an aggregation of primitive processes at lower levels. What are the methods of writing PSPECs ?

- A student of English called “Structured English”
- Decision Trees
- Decision Tables

This vocabulary is too limited in the data dictionary. The syntax is also too limited. The following are some guidelines for writing PSPECs.

- Describe only policy issues and matters.
- Do not describe implementation issues.
- In describing, use only sequence, if, then and do... while.
- Do not use punctuations, compound sentences or adverbs.

The PSPECs for the university placement system for some typical processes are shown in the below figure.

Fig. - 14.12

Process No	:	<Give an appropriate ID>
Process	:	Update Mailing List
Description	:	Check what is suggested by student for suggested companies. Add new company or change required fields from existing company or delete existing company.

Process No	:	<Give an Appropriate ID>
Process	:	Mailing to Company
Description	:	Receive Printed Brochure / Letters from Process “Prepare Brochure / Letter” Prepare address-slips of companies Mail brochure / letters to companies.
Process No	:	<Give an appropriate ID>
Process	:	Update Late Communications
Description	:	Prepare Defaulter List from Mail List Prepare reminder to company mail reminders. If reply from such company received then record in company file.

It must be noted that we need not write a PSPEC for the process “Invitee for Interview”. This is because they can be understood as an aggregation of lower level process.

Physical and logical DFDS :

Physical DFDS are listed in DFDS, as they may not be structured well. Logical DFDS are constructed to enable proper structuring into logical files. The files are restructured based on the following criteria :

- Have minimum data items that are required for each process.
- Do a normalization of the files.
- Synthesize the files
- The derived fields are to be eliminated.
- Have the data structure diagram constructed.

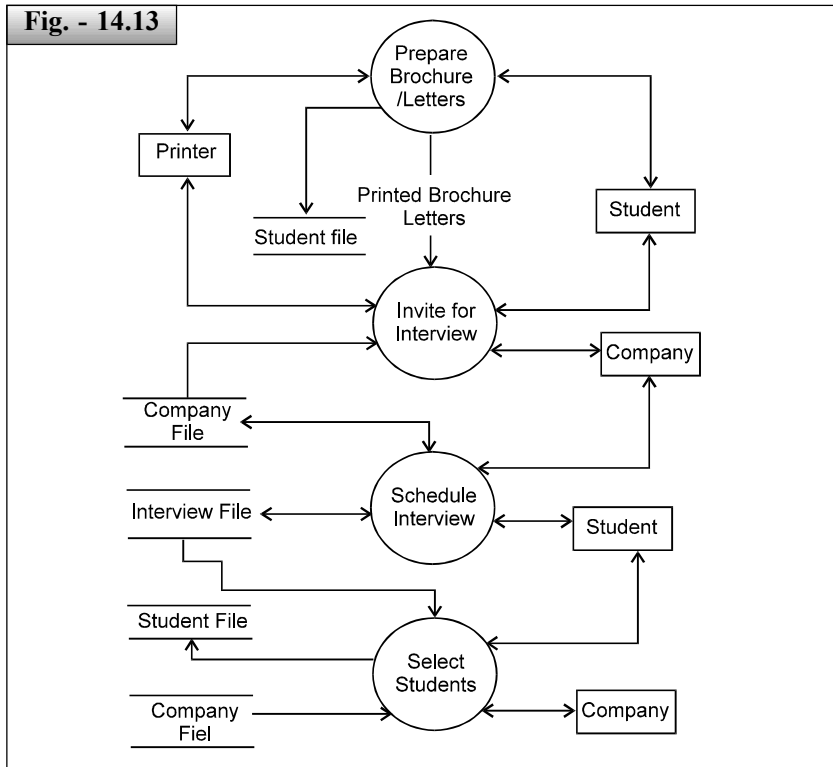
The following are the various files involved in the physical DFDS of the university placement system.

- Mail list
- Time Schedule Register
- Company File
- Student File
- Previous Brochures; Letters
- Biodata File

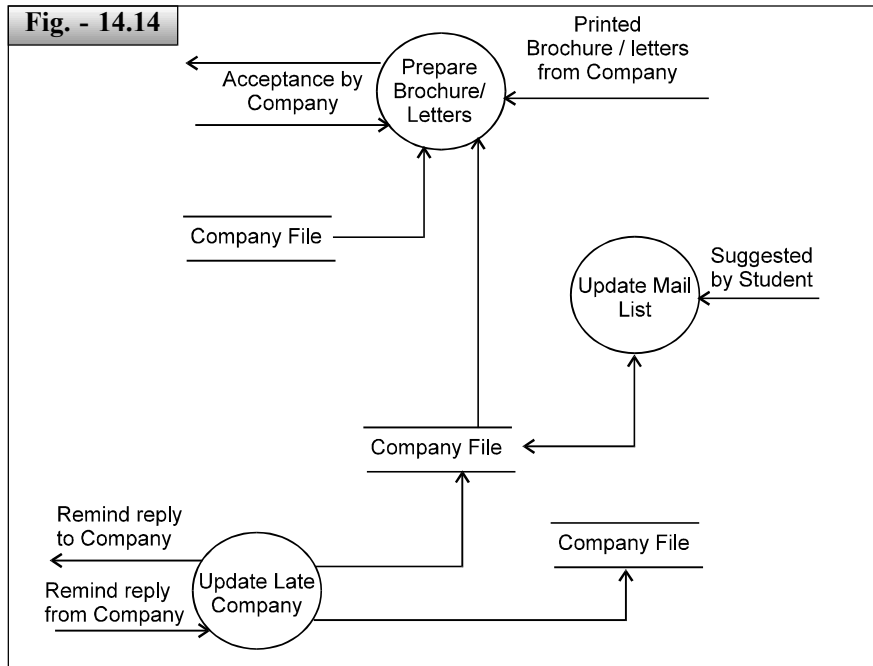
Unstructured files such as “Previous Brochures, Letters” and “Biodata File” need not be restructured. This is because the university placement system itself doesn’t determine their structure. Below figure shows the structure of some of the current files in the physical DFD. Note that this is only indicative and not complete.

Table-14.6

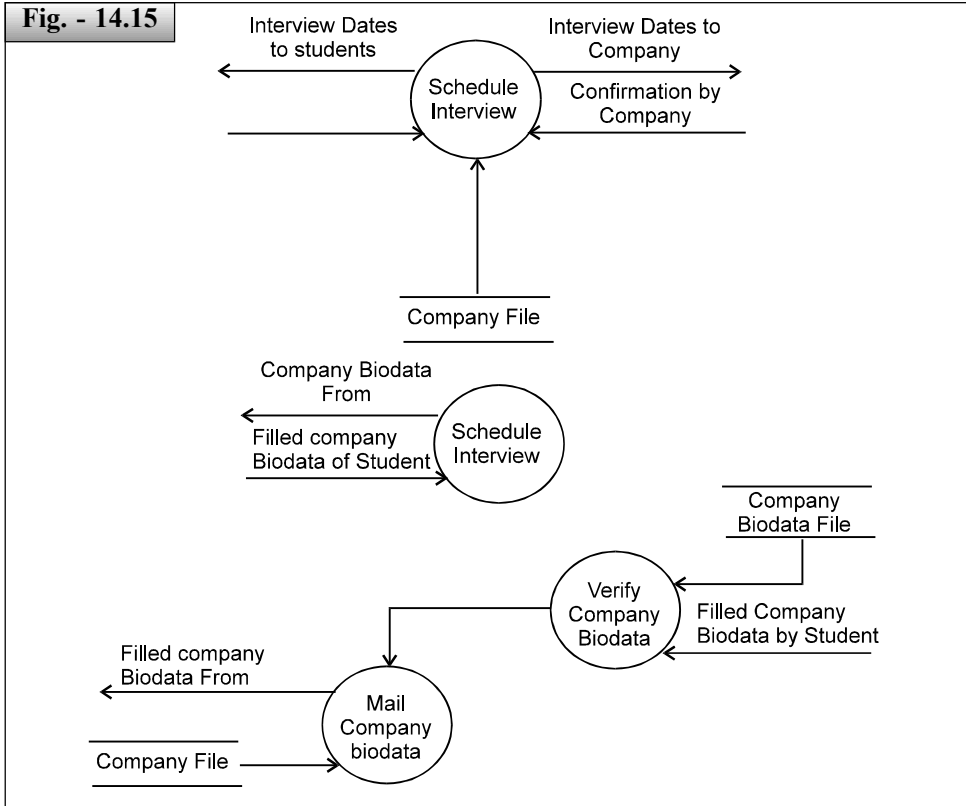
File	Group name	Fields	Remarks
Time Schedule		Company-Code Company-Name Interview – Date Interview –Details	
Mailing List		Company – Code Company – Name Company – Address	
Company File	Year-Information	Specialization Salary Location	Last 10 years data
	Interview-information	Interview-dates Interview-details	Maximum 2 dates are possible
	Student-Details	Student-Roll Student-Name Student-Address	
Student File	Student-Details	Student-Date of Birth Student-year	
	Job-Information	Student-Roll Student-Name Specialization Student-Salary Student-location	
	Student Information	Student-Roll Student-Name Student-Address Student-Date of Birth Student-years	
	Qualification	University Degree Year Months	Latest 3 educational qualification
Student File	Experience	Year-From Year-To Organisation Salary	Latest 3 experience
	Pending-Information	Company-Code Company-Name	Maximum 3 entries
	Order-Information	Company-Code Company-Name	Maximum 3 entries
	Job-Information	Company-Code Company-Name Designation Salary	

Fig. - 14.13

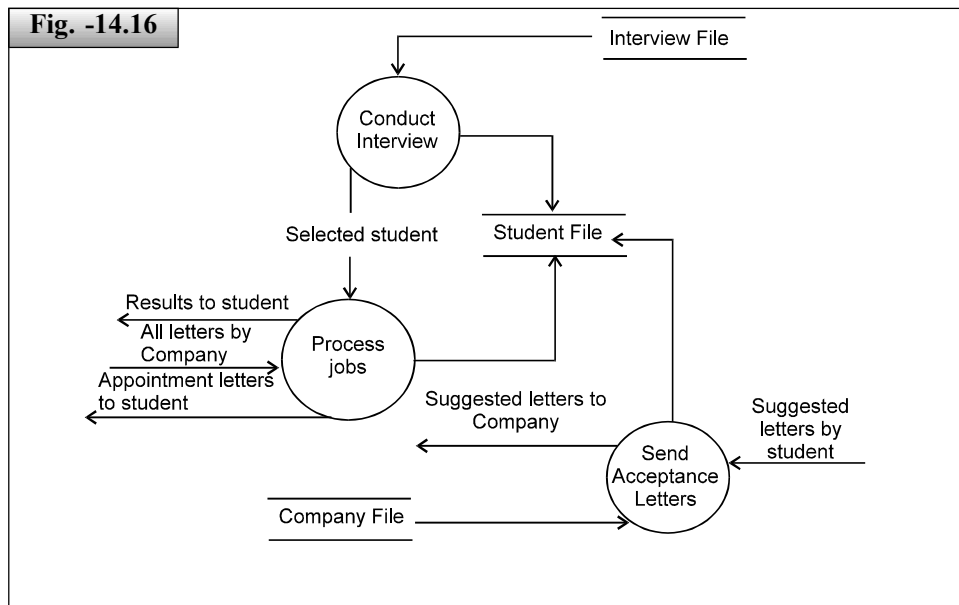
The level 2 DFD for the process “Invite for Interview” is shown in figure Fig. 14.14.

Fig. - 14.14

The level 2 DFD for the process “Schedule Interview” is given in the figure 14.15.



Below figure shows level 2 of the process “select students”.



It is important in any system for the current physical DFDs to be independent of implementation details. The data dictionary entries would be identical concerned with data flows. That is, there are no changes to them as we move from physical DFDs to logical DFDs. For the data stores, though, the entries in the data dictionary vary as we move from physical to logical DFDs. This is because the data stores are restructured in the logical files. The PSPECs must show access of processes to logical files instead of physical files.

Handling Real-Time Systems

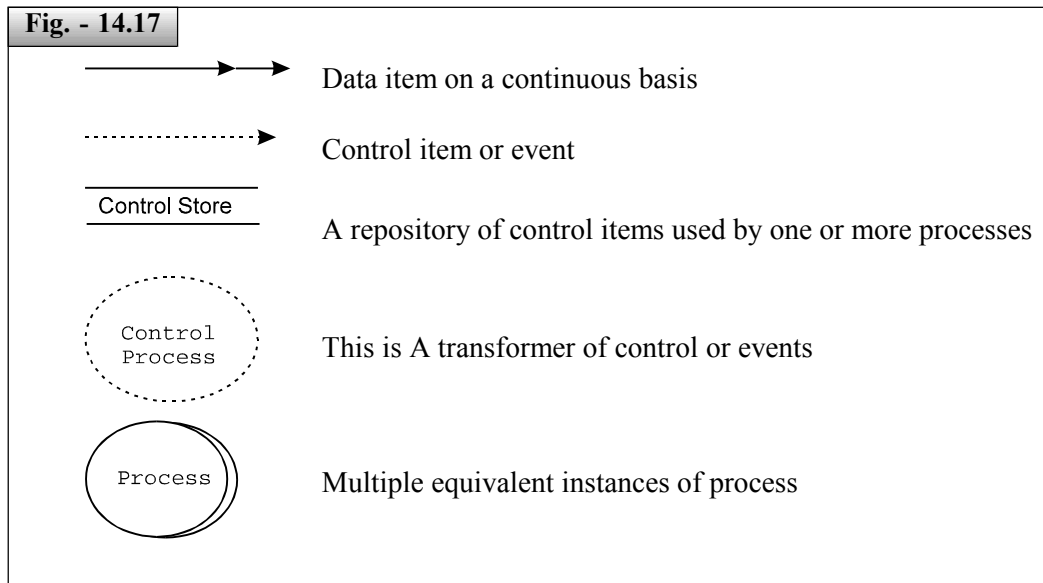
There are many applications that are time-dependent. These applications handle more control information than data. The normal notations of DFD enumerated for information system type of applications tend to be inadequate to describe real-time system requirements.

Ward and Mellor provided in 1985 some extensions to accommodate the needs of describing real-time systems. Some of the demands of real-time systems are given below:

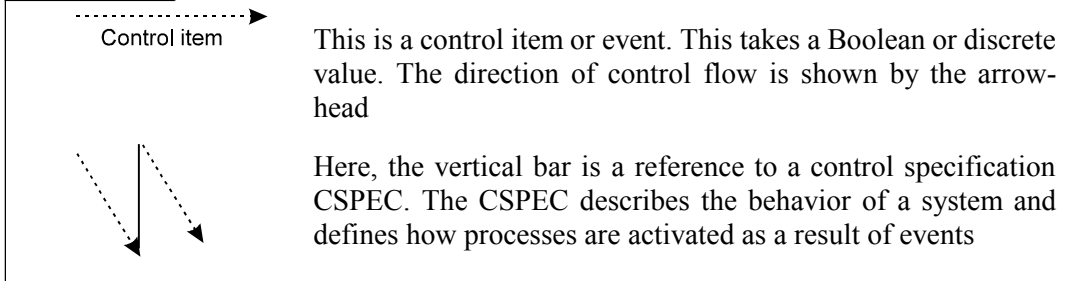
- Some information flow happens on a time-continuous basis
- Multitasking situations require multiple instances of the same transformations
- Need to handle system states and the mechanism of state transitions

The extensions suggested by Ward and Mellor are given in Figure 14.17.

Fig. - 14.17



In 1987, Hatley & Pirbhai made some more extensions to handle control oriented aspects in DFD. While Ward & Mellor provided graphical devices, Hatley and Pirbhai focus on representation and specification of control oriented aspects of the software. Their extensions are given below in Figure 1.16.

Fig. - 14.18

Behavioral modeling is a very significant aspect of requirements analysis. The notational extensions provided by Ward-Mellor and Hatley-Pirbhai strengthens the DFD's capability to do behavioral modeling. Systems require data flows, data modeling, control flow and behavioral modeling.

For highly data intensive applications, a strong modeling method called E-R Diagram (not discussed here) will be used.

- **Steps in Structured Analysis**

Structured analysis requires the carrying out of the following steps:

- **Create a Data Flow Model**

- Draw the Context Level DFD
 - Refine through to Level 1, Level 2, etc.
 - Stop when you begin to describe procedural / implementation details

- **Develop the data dictionary**

- **Write PSPECs**

- **Create a Control Flow Model**

- Develop CFD (Control Flow Diagram), that is, a DFD showing control flow
 - Write the CSPECs
 - Do data modeling, if required, through E-R Modeling
 - Integrate the analysis into the software requirements specifications

Structured System Design

A structured chart is the diagrammatic representation of a software structure and one of the most vital outputs of the structured design process. As a huge fraction of the expenditure of a standard system is connected to the maintenance of the software package during its lifetime, it has become a necessity to create a software package whose maintenance is easy. For this purpose, specific portions of the software program are to be altered using the least amount of time and some resources by those persons who are familiar with those specific portions.

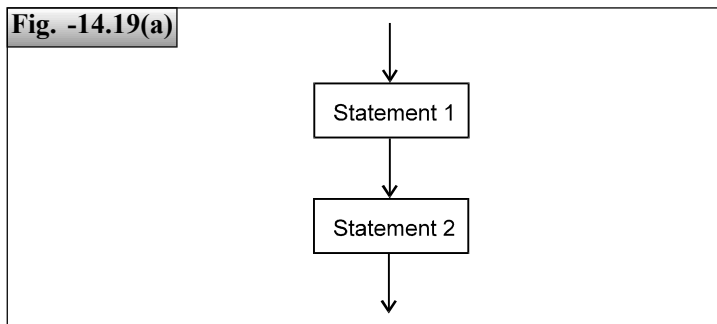
- **Structured programming concepts**

On the basis of limited resources, the software for 1st and 2nd generation computer systems were developed. Each program included a set of tricks. These tricks should be confirmed that the available memory and the disk space would not be exceeded. The program ensured that the input given was accepted and the necessary output was provided in the batch mode. The third generation computers consists of sophisticated computing resources at affordable prices. For these more versatile and easily maintained softwares are used. AS software applications were more dependent on the environment so, a good number of commercial applications had to be altered.

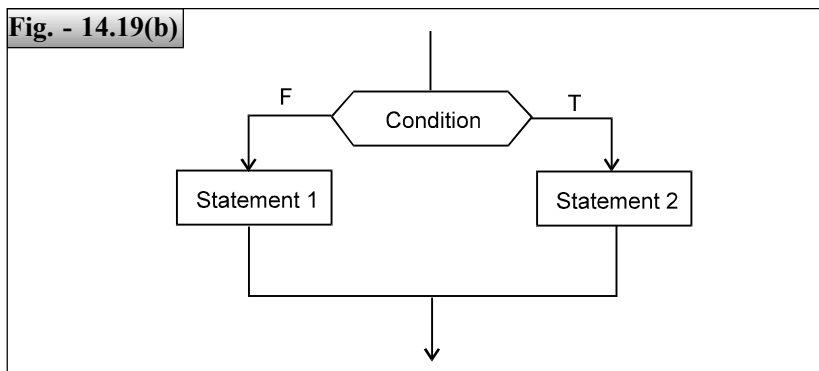
We need to know the fact that non-structured computer programs using “GOTO” instructions are complex to understand and alter. According to Dijkstra, the complexity of a computer program varies directly with the number of “GOTO” instructions. For the solution of some particular problems we are required to write a program that are apart from the sequential structure. This type of structured program is termed a “GOTO less” program, i.e. it does not use “GOTO”.

let us know about sequential, selection and iteration structures.

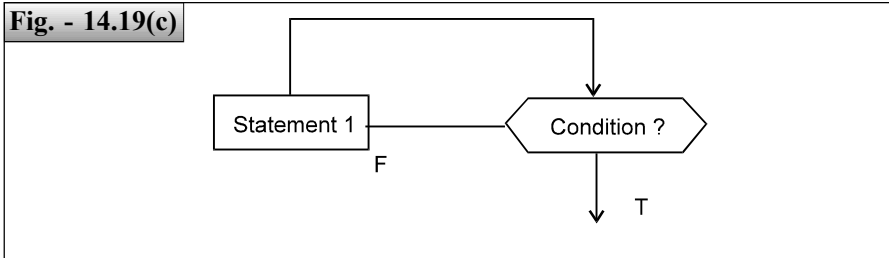
Sequential Structure is the simplest form of structures, in which statements are executed sequential manner and when encountered. See the figure (a).



Selection structure is also known as the ‘If-then-else’ structure. Also a condition is examined. If it is true, a set of commands is executed; or else another set of commands is executed. See the figure (b).



Iteration Structure is also known as the “Do-while” structure. It means one set of statements is executed repeatedly till a particular condition is satisfied. If the condition is not satisfied, then the control passes on to the next statement. See the figure (c).

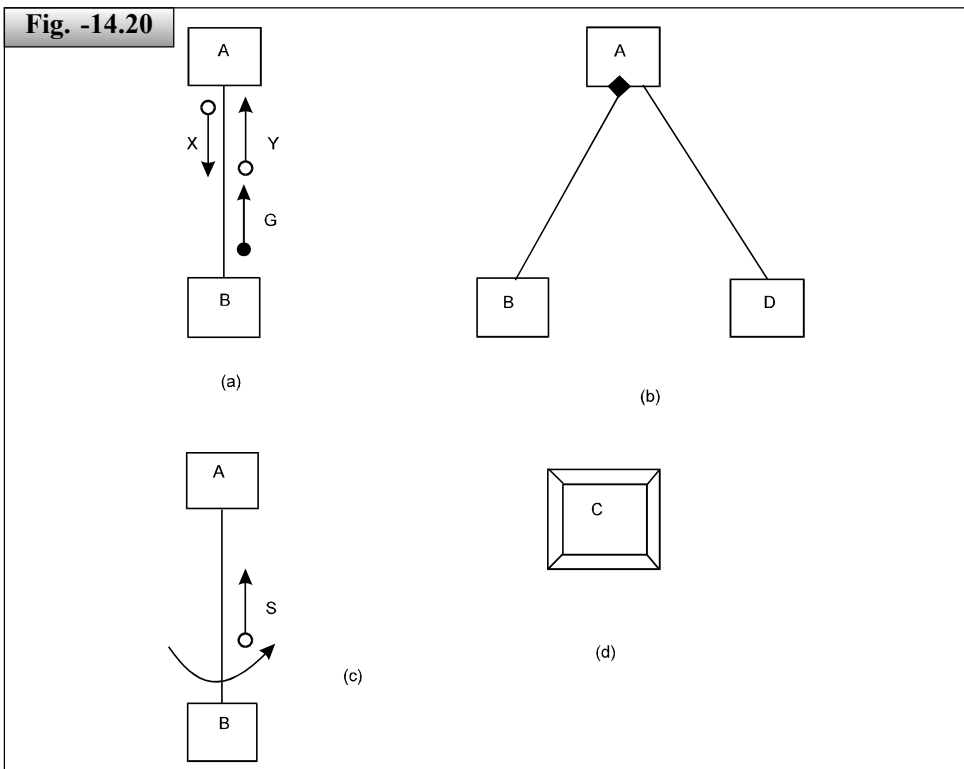


c) Structure Chart

A structure chart is a technique of documentation that depicts the hierarchy of and the interrelationships among modules in a computer program. Let us consider a simple payroll processing system. A record is examined, the gross salary is computed for temporary and permanent workers, deductions are calculated, a salary slip for each employee is printed and specified running totals in a ledger form are printed.

Symbols of structure chart :

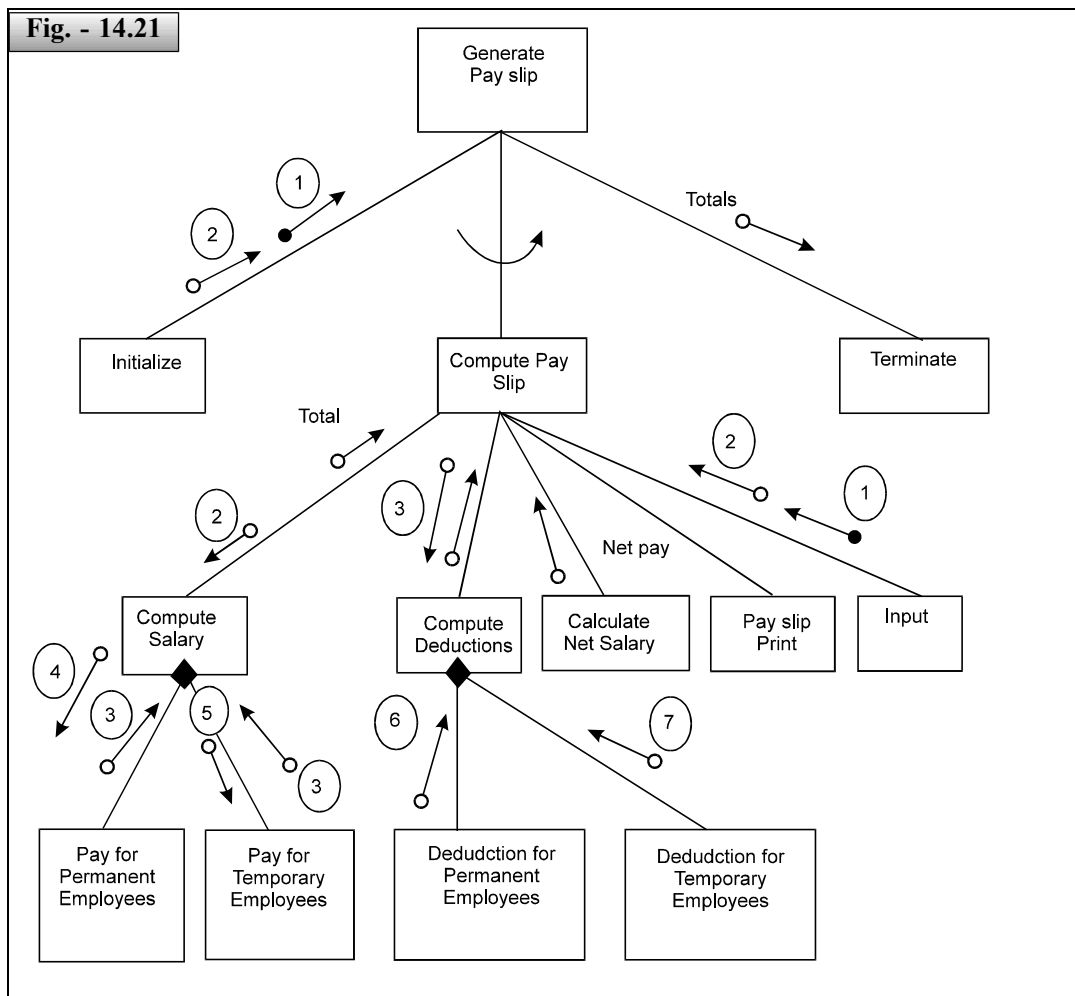
There are certain symbols which are used for created a structure chart for the above problem. The symbols are shown in fig.14.20.



Module A calls Module B in Figure 14.20(a). The control then comes back to Module A. Module B may get some data elements like X, which is shown along with the symbol - an arrow with one end having a small transparent circle. Module B finally sends data element Y and a control variable G (like a logical variable, flag, etc.) back to Module A. This is the fundamental representation in the structure chart. An arrow denotes flags with one end having a small opaque circle. An arrow denotes data with one end having a transparent circle.

Figure 14.20(b) illustrates the conditional execution of Module B from Module A. Conditional execution is denoted by a diamond symbol. Module D is not connected to the diamond symbol, so it is always executed. Figure 14.20(c) shows an arc on the link, connecting Module A and Module B. This arc indicates repeated execution of Module B. The double-lined box in Figure 14.20(d) represents standard library routines that can be sometimes incorporated in the structure chart.

The structure chart for the example of the simple pay-roll processing system is shown below in Figure 14.21



1)	End of File
2)	Employee Record
3)	Gross Pay
4)	Basic Pay
5)	Hours Worked
6)	Deductions for Permanent Employees
7)	Deductions for Temporary Employees

Let us take the module 'Generate Pay Slip' as the principal module. The module 'Initialize' initializes variables, accepts the present day's date and reads the first record of the employee file. The module 'Compute Pay Slip' is executed repeatedly till the last employee record. This module calls the module 'Compute Salary'. If the employee is a permanent employee, the module 'Pay for Permanent Employees' is executed; otherwise if the employee is a temporary employee, module 'Pay for Temporary Employees' is executed, to compute the gross salary.

The module 'Compute Deduction' works out the deductions using the gross salary, calling either the module 'Deductions for Permanent Employees' or the module 'Deductions for Temporary Employees'. The module 'Payslip Print' prints a pay-slip. The module 'Initialize' now reads the next record of the employee file and the whole process repeats in this way till the end of the employee file. The control then passes to the module 'Terminate', at which the running totals are printed. The program halts when the control passes back to the module 'Generate Pay Slip'.

This structure chart demonstrates the hierarchy of the modules and the interrelationship among the modules. To write a program, it is sufficient to have such a structure chart in conjunction with the logic of each module.

Deriving Structure Chart from DFDs

Structured analysis offers structured specifications made of DFDs, DD and PSPECs (we have learnt these in an earlier Unit on Structured Analysis). The two methods for producing the structure chart after taking into consideration its shape and complexity are the following.

- *Transform analysis:* Usually, commercial systems pick up the input, process it and then provide the output. If such a feature is present in the DFDs, then a structure chart can be created
- *Transaction analysis:* Generally, business applications entail identifying the type of transaction and then processing it. A structure chart can be created if such a feature is present in the DFDs -

It is possible for the ultimate structure chart to utilize a combination of both the above methods.

Transform analysis is a design approach, which uses a well-defined process to transform specific inputs into the necessary outputs.

There are four important steps involved in transform analysis, which are listed below.

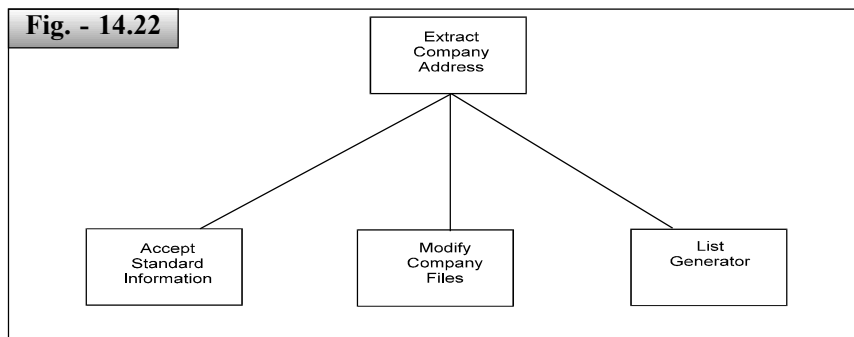
- Identifying central transform: The input (afferent) and the output (efferent) streams are to be eliminated, leaving behind that portion of the DFD called the central transform.
- Creating a first-cut structure chart: A boss from the central transform is identified. In case there does not exist a boss, then an artificial boss can be created. The central transform can be visualized as being tied by strings (just like table tennis balls) and hung by the boss. A first-cut structure chart is thus produced.
- Editing the first-cut structure chart using the guidelines given below.
 - Adding input and output streams
 - Adding `read` and `write` modules
 - Dividing the central transform into sub-modules on the basis of the various levels of DFDs
 - Adding modules to handle errors
 - Adding details of initialization and termination
 - Furnishing proper names to all modules
- Ensuring that the final design is satisfactory: To obtain a better design, the previous 3 steps can be revised too, wherever necessary.

Transaction analysis is a design approach, which is based on the different types of transactions that the system has to process.

There are four important steps involved in transaction analysis. These are listed below.

- Identifying a process that accepts one input data stream and creates many mutually exclusive data elements by the transaction type, in the DFD
- Drawing a structure chart wherein the transaction types are subordinate processes
- Drawing an action module for every transaction type module
- Drawing a detailed module for every action module

The first-cut structure chart for the University Placement System discussed during structured analysis is shown in Figure 14.22.



Now let us recall the example we had taken on when we learnt structured analysis - that of the university placement system. Assume that we have created the DFDs for the placement system based on our analysis. For ease of our understanding, we shall provide the following.

- DFD for the process “Mail to Company” in Figure 14.23
- DFD for the process “Update Late Communication” in Figure 14.24
- DFD for the process “Extract Company Address” in Figure 14.25
- The level 1 physical DFD of the placement system in Figure 14.26

Fig. - 14.23

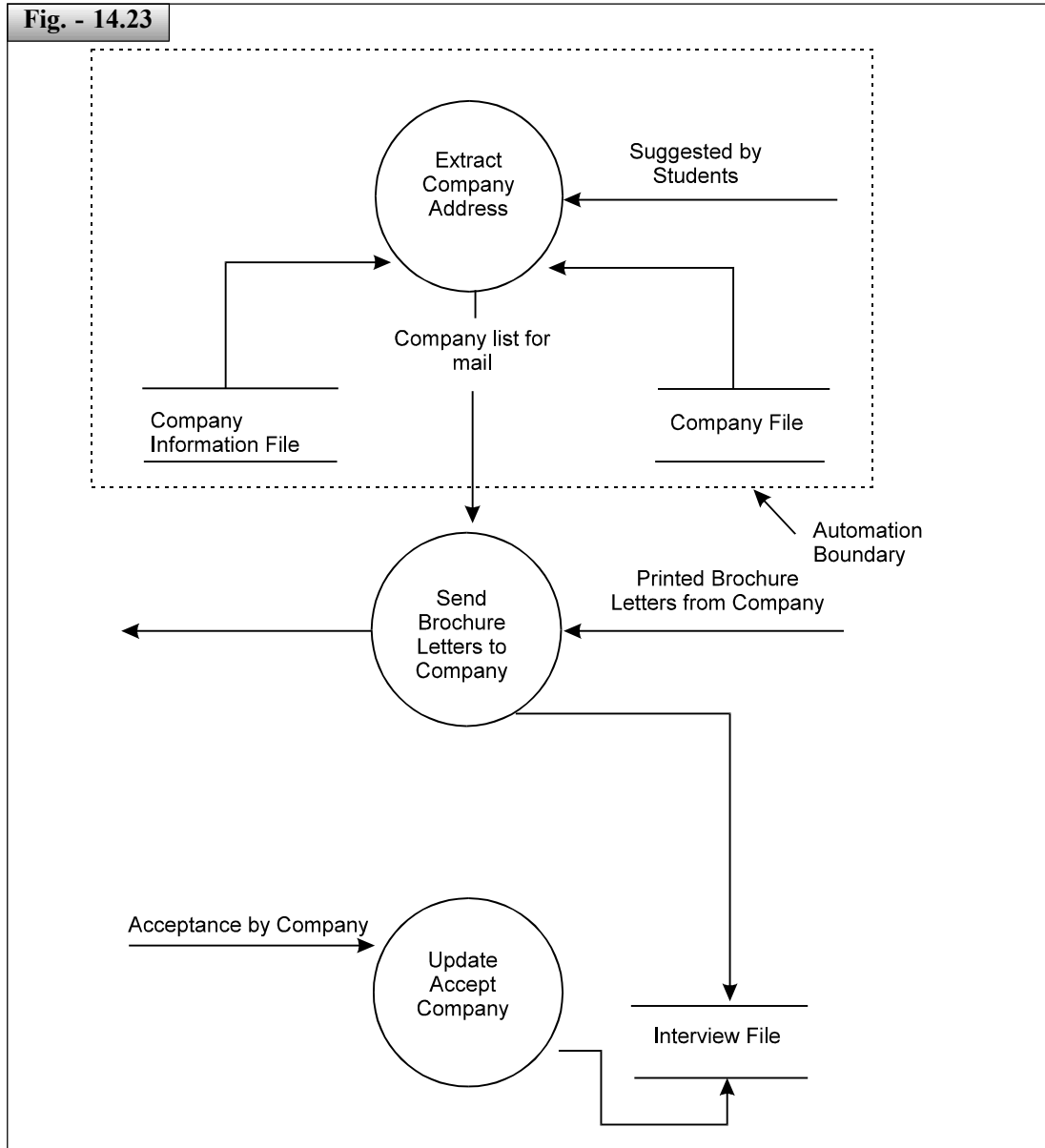


Fig. - 14.24

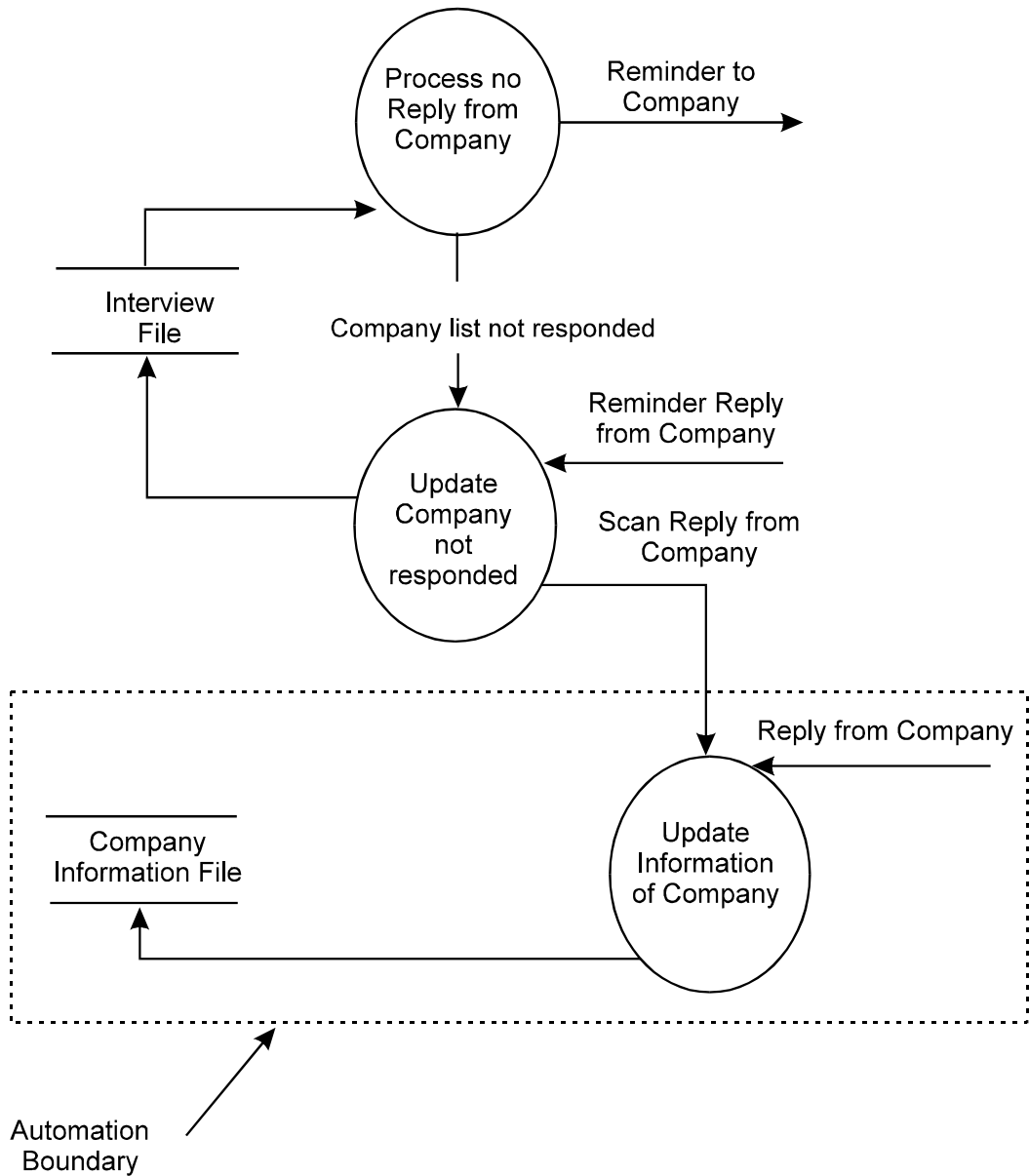


Fig. - 14.25

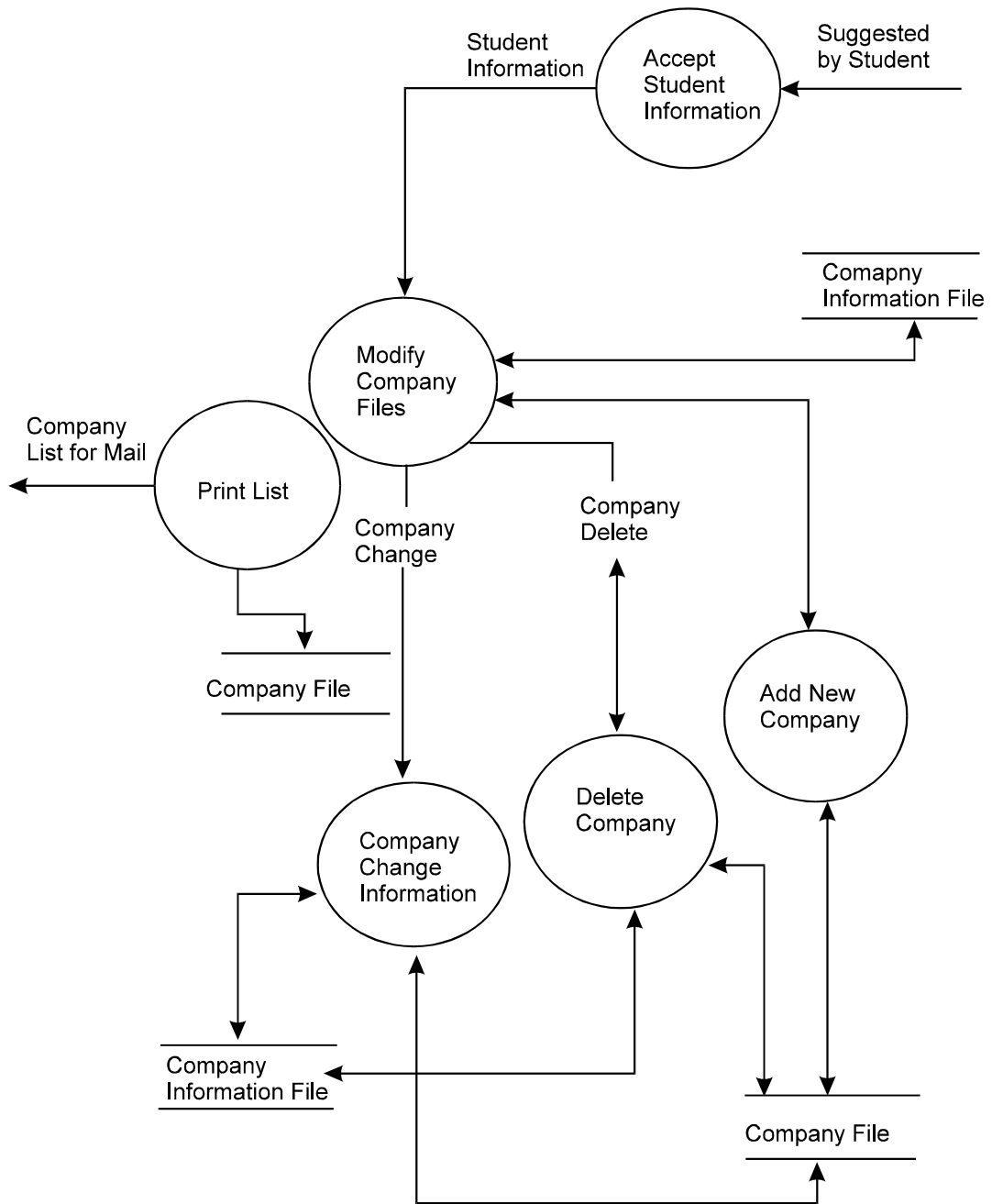
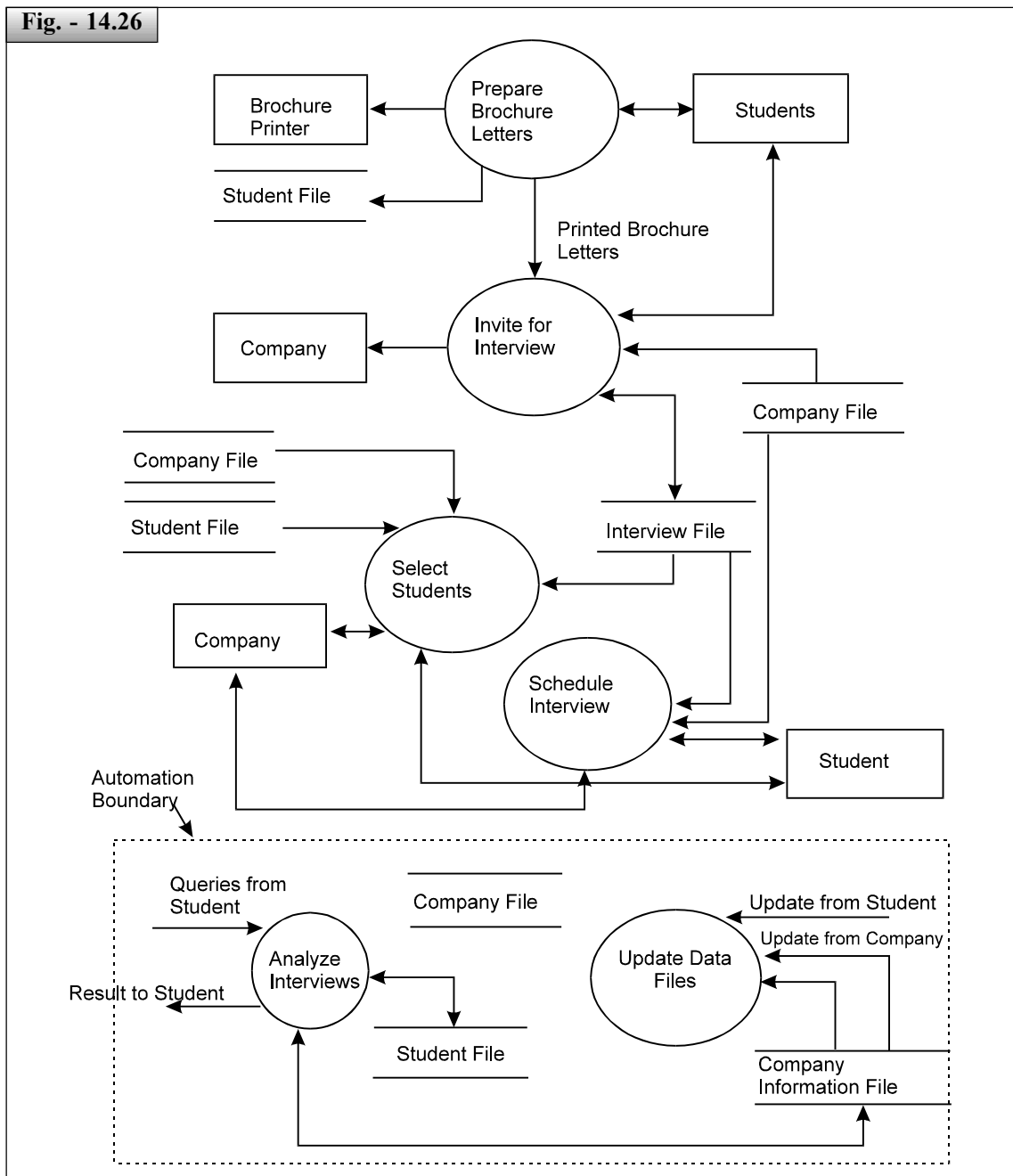


Fig. - 14.26



The new physical DFD shown in Figure 14.23 shows some sort of automated subsystems. Figure 14.24 shows the automated process “Extract Company Address”. A careful examination of this DFD shows that we need to do both transaction analysis as well as transform analysis. The “Accept Student Information” process is an input module. The “Print List” process is the output module. The “Modify Company Files” may be

treated as a central transform process. The “Modify Company Files” process accepts “student information” and, depending upon the input, suitable action is taken to add, delete or modify the company file. Some control logic is being shown in this DFD, which is not very commonly shown. The structure charts for this subsystem are based on transaction as well as transform analysis (See Figure 14.27).

We shall now do the transform analysis of the “Extract Company Address”.

- Accept “Student Information” as the input process and “Print List” as the output process. Therefore “Modify Company Files” is the central transform once the input and output processes are eliminated
- We need to create a parent process called “Extract Company Address” at a higher level
- The process “Extract Company Address” is considered to revise the first-cut structure (Figure 14.22). Note that this has one input, “Student Information” and multiple outputs, i.e. “Company, Address”, “Company Delete” and “Company Change”. We need to do transaction analysis now

We can make a few more changes by way of improving the solution. For instance, we add an “Initialize” module to initialize variables, to accept user input, etc. We can also add a “Terminate” module that does a clean up job by printing summary of addresses printed, among other things. The “Print Message” module is included to handle errors so that a suitable error message may be printed.

We shall now take up the transaction analysis of “Extract Company Address”.

- The “Modify Company Files” is the process, which has one input and multiple outputs
- We need to add separate modules for “Add Company”, “Delete Company”, and “Change Company” as modules subordinate to “Modify Company Files” for each of the outputs
- Since no complex processing is envisaged here, all the action modules are subsumed in these subordinate modules
- The module “Print Message” has been added by way of detail

The procedure for deriving structure charts from DFD is clear and simple. It is strongly suggested that you practice deriving structure charts for DFDs in Figures 14.24 to 14.26.

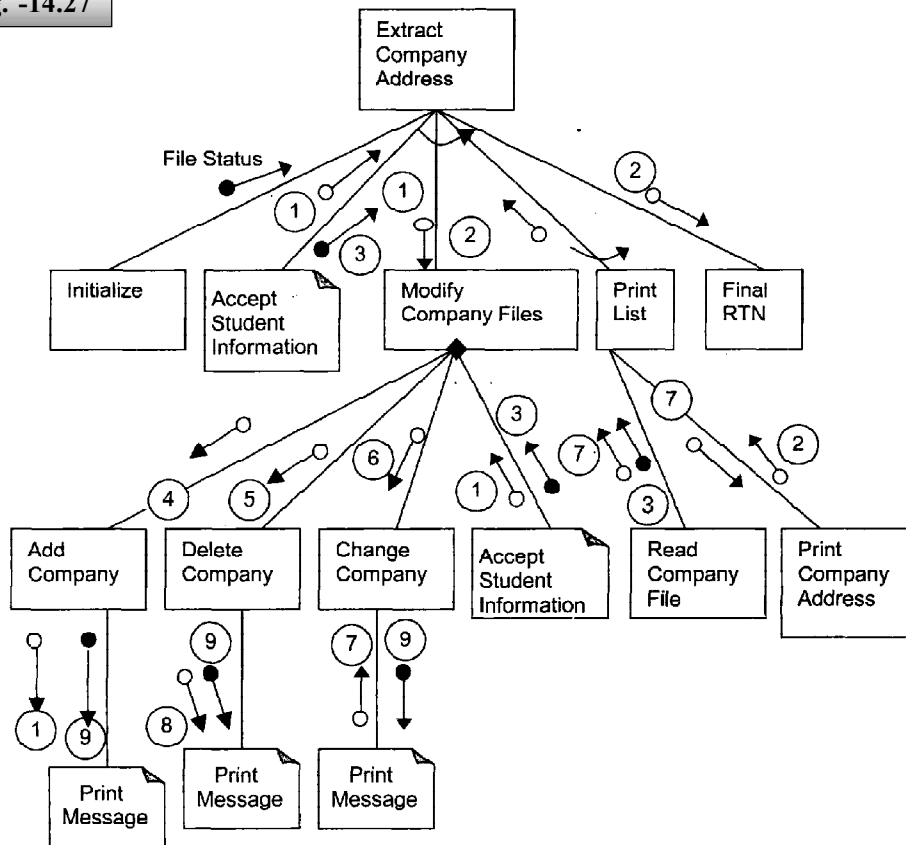
From the structure charts and other documents provided as part of the analysis, we can now move on to write down the pseudo code and proceed to the coding phase. But this will not be handled in this course. We shall just provide a brief discussion of structure charts and their relation to pseudo code.

Structure Charts and Pseudo Code

The structure chart is a mechanism for describing the *stepwise refinement* (sometimes called *functional decomposition*) process. Starting with an abstraction of a single process, we successively decompose abstract modules into ever more concrete and less abstract

modules until we arrive at a structure where the designer understands the logic of each module. As we decompose our modules, the level of control via calls to modules diminishes and level of activity (computations, comparisons, etc.) increases, until at the lowest levels we have no control but all computations and data manipulation.

Fig. -14.27



- 1) Student Information
- 2) Total Counter of Addresses Printed
- 3) End of File
- 4) Addition Information on Company
- 5) Deletion Information on Company
- 6) Change Information on Company
- 7) Change Company Record
- 8) Delete Company Record
- 9) Error Code while performing a task

The notion of stepwise refinement is a key component in the structured approach to software development, called “Structured Design”. The modules we are creating constitute our preliminary design, because they show the pieces have an overall design and how these pieces fit together: parent processes, child processes, and common processes. All boxes in the structure chart are to proceed “down” the page, not across. The chart itself should resemble an organization chart of a business.

The extent to which we successively decompose modules into lower-level modules is a matter of taste and experience. The guideline that most people use is to decompose the modules until we can conceptualize the logic/algorithm of the module in fewer than 30-35 lines (probably fewer in a language such as C). This is not absolute, though. In many instances a complicated bit of computation might require its own module, while a long series of simple data manipulations that is easy to understand might make a module “longer” yet not complicated but quite cohesive.

The detailed design of a program follows the preliminary design phase. The detailed design consists of writing the pseudo code for each of the modules that have been identified in the software design. Because the modules should be single-purpose (having high cohesion), the logic should be focused. If the logic has multiple functions, then the module should probably be decomposed even more.

Anytime a common module is to be altered in any way, the structure chart tells us other modules that are dependent upon it. These control modules must be checked to see if the proposed changes to a common module might have an adverse impact on their logic.

As the programs grow and the systems include more than one program, we will need a structure chart for each program and a single high level structure chart showing which “programs” call which “programs”.

The problem with pseudo code comes in its implementation. Pseudo code represents the algorithm, the logic, and the approach to solving the problem, module by module, with the structure chart as input. Since pseudo code is algorithmic, it should be language independent and should strive to meet this tenet. So consider the following guidelines:

For looping and selection, the keywords that are to be used include the following.

- Do While...EndDo
- Do Until...Enddo
- Case...EndCase
- If...Endif

When calling subprograms, we can use the following.

- Call ... with (parameters)
- Call; Return; Return

We should try to eliminate the use of the word Perform, as this is clearly language-dependent. We should not include the words Begin, End, Return and other similar words in our modules. We must never include any data declarations in the pseudo code. Pseudo code is logical procedure. It does not include data definitions.

As verbs, we should use the words *Generate*, *Compute*, *Process*, etc. Words such as *set*, *reset*, *increment*, *compute*, *calculate*, *add*, *sum*, *multiply*, ... *print*, *display*, *input*, *output*, *edit*, *test*, etc. with careful indentation tend to foster desirable pseudo code. Note that “increment” implies adding 1 to some data item. Similarly for decrement, which means subtract 1 from a data item, we should state things very clearly, like “decrement field by 1” or, “decrement field by 5”.

One way to make the pseudo code more language-independent is to use words such as *do*. Thus, a *do ... until (condition)* or a *do while (condition)*, each with an *Enddo* at the end of their scope is encouraged. Thus a *do While (not end of file)* followed by several statements, indented and followed by an *Enddo* lined up with the *do while* is acceptable form. For example, consider the example below.

```
DO WHILE (COUNTER < 100)
```

```
    . . .  
    . . .
```

```
ENDDO
```

or

```
DO
```

```
    . . .  
    . . .
```

```
UNTIL (NO MORE RECORDS)
```

This structure is good. Note that the *DO . . . UNTIL* has the test after the code is executed the first time, and once thereafter until has the test *after* the code is

Pseudo code is a step-by-step solution to the problem. Thus, there must be a level of detail in the algorithm to support program development. An entry like “Calculate Total Pay” without showing the formula is of not much help. Phrases like “Build Table” are too abstract to be of much use. We need to show *how* to build the table, and *what* is the formula for computing the total pay. It may be appropriate to abstract these into a separate module. But regardless, the pseudo code must include a certain level of detail.

d) Datastructure Diagram

It is a pictorial representation of the relation between entities (people, places, events, and things) in a system and the set of information about the entity.

Datastructure Diagrams, a useful tool for developing the frame work which includes various data items, record and files during data-flow analysis.

Data structure diagrams are graphic tools that show the logical data structure requirements of an system application.

There are following purposes for data structure diagram.

1. To verify information requirements.
2. To describe data associated with entities.
3. To represent the relationship among the entities.
4. To established a communication in between the requirements and the designers, involved in the designing phase.

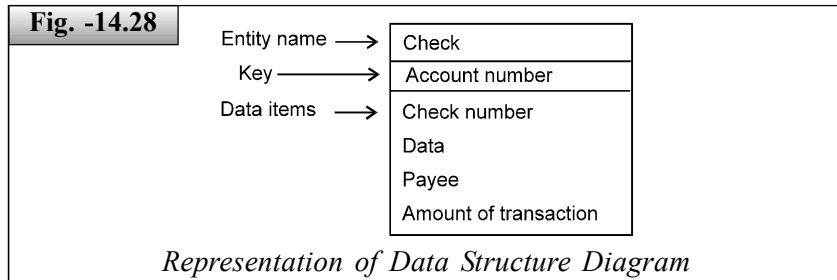
NOTATION REQUIRED FOR DATA STRUCTURE DIAGRAM

Common notations are used in preparing the data structure diagrams, are follows :

Entities are represented by rectangles. 

The entity name are represented at the top of the rectangles and a list of attributes (data items or fields) describing the entities, are represented next to the entity name.

The following figure represent a data structure diagram for checking an account.



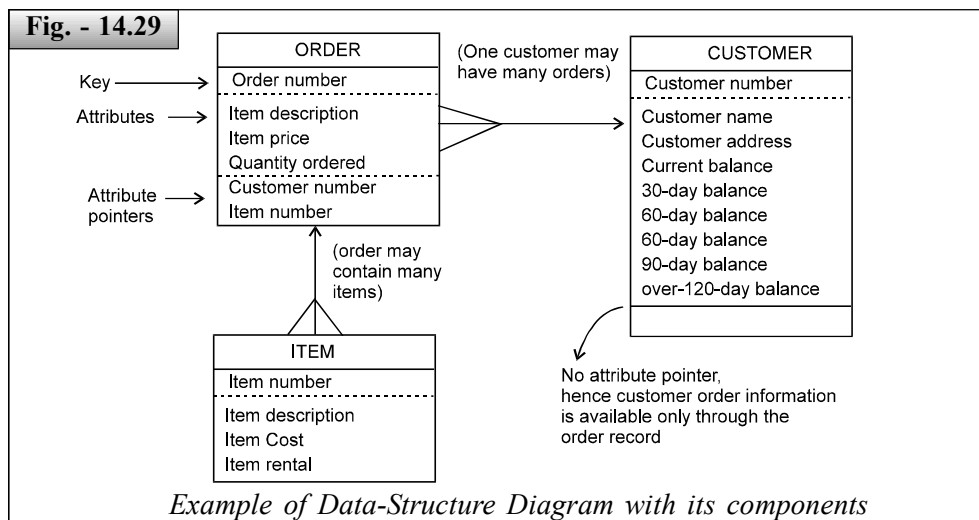
In addition to the basic component, a data structure diagram have to additional essential elements are required along with the basic components (entities, attributes and records). These two additional elements are :

1. Attribute pointer
2. Logical pointer.

1. **ATTRIBUTE POINTER** : It links two entities by common information, usually a key attribute in one and an attribute (non-key) in another.
2. **LOGICAL POINTER** : It identifies the relationships between entities; serve to gain immediate access to the information in one entity by defining a key attribute in another entity.

Logical pointers, usually listed at the bottom of the data structure diagram, are the links to other entities included in the diagram.

for example :



14.5.3 Tools for Proto-typing

The tools are :

- 1) Fourth Generation Language (4th G.L)
- 2) Application Generators
- 3) Reusable codes
- 4) Program Generators
- 5) Application packages
- 6) Personal computers

1. *Fourth Generation Languages (4th GL)*

Fourth-Generation languages include a broad range of computer languages with the emphasis on “what is to be done rather than how to perform a task”. Fourth-Generation languages were created to help meet the need to develop software more efficiently.

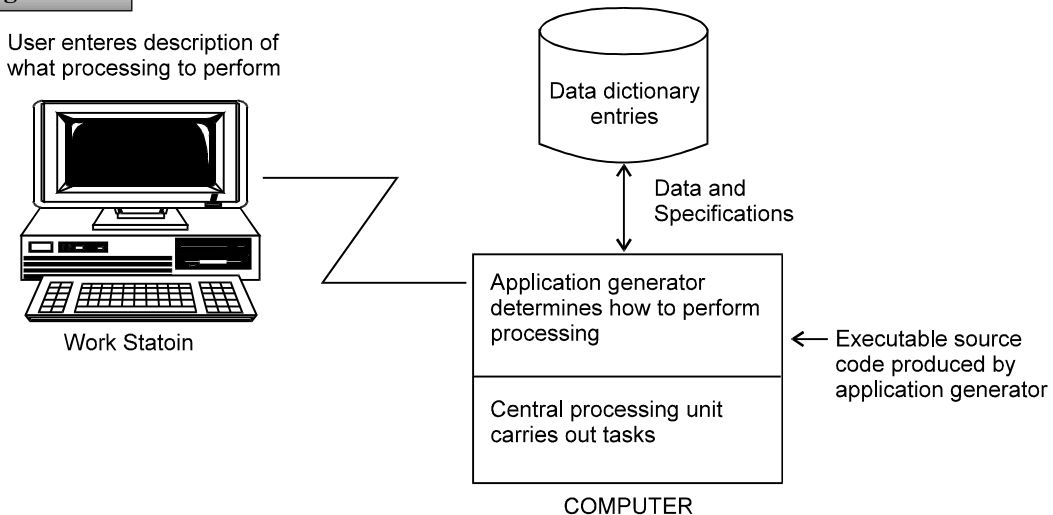
Examples of 4th G.L. are : FOCUS, SQL, NOMAND

2. *Application Generators*

Are software programs that permit the specification of an entire application at very high level.

Application Generators provide the capacity to develop applications that accept input, validate data, perform calculations and follow complicated processing logic, interact with files and produce reports and output.

Fig. - 14.30



Functions performed by application generator

List of Application Generators :

<i>Package / Language</i>	<i>Vendor</i>
ADS	Cullinet
Application Factory	Cortex Corporation
FOCUS	Information Builders
MAPPER	Unisys (sperry)
MANTIS	CINCOM
NATURAL	Software AG
RAMIS	Mathematica, Inc.

3) Reuseable Codes :

Many organizations encourage the creation of program libraries containing individual modules of code. Typically, the modules were developed for use in other systems and, if the modules are reusable, such a library can be a valuable Proto-typing tool. Selected reusable code modules can be retrieved from storage and modified, if necessary. By inserting the modules in the prototype, a working system is created quickly and relatively in expensively.

Some organizations use specific tools for Proto-typing more than others. However, all of those we have discussed have a place in application Proto-typing.

4) Program Generators

A screen generator is an interactive facility for “painting” screens and performing automatic input validation and processing (table look-ups etc.)

By responding to prompts or by entering simple, one-word commands, one can select performances for high lighting information, making blocks, with chosen colours or using reverse video.

Screen generators also allow users to automatically prepare. Components that assist the user-to-machine interaction, including location of data entry fields, data display fields, and column headings, labels and messages.

Application generators typically include an automatic screen generation capacity.

5) Application Packages

These are the application software designed for specific purpose of use for example : Tally package, V.B. Crystal Report.

6) Personal Computer

Personal computers embedded with software and accessories (hardwares) are used for application prototype development.

SUMMARY

A **programming tool** or **software development tool** is a program or application that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object.

EXERCISE

1. Why Structured Analysis is essential in case of software development ?
2. Why Data Dictionary is important ?
3. Differentiate Structured Analysis from Structured Design ?
4. What is a context diagram ? Give an example.
5. What is Program Structure chart ? Why it is necessary for system design ?

