# SOFTWARE ENGINEERING

**C-124**

**Self Learning Material**

Directorate of Distance Education

SWAMI VIVEKANAND SUBHARTI UNIVERSITY
MEERUT-250005
UTTAR PRADESH

**SIM Module Developed by : Bharat Bhushan Agarwal**

**Reviewed by the Study Material Assessment Committee Comprising:**

1. **Lt. (Gen.) B.S. Rathore, Vice-Chancellor**
2. **Dr. Sushmita Saxena, Pro-Vice-Chancellor**
3. **Dr. Mohan Gupta**
4. **Mr. Pushpendra**
5. **Mr. Rakesh Joshi**

Copyright © Laxmi Publications Pvt Ltd

No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the publisher.

Information contained in this book has been published by Laxmi Publications Pvt Ltd and has been obtained by its authors from sources believed to be reliable and are correct to the best of their knowledge. However, the publisher and its author shall in no event be liable for any errors, omissions or damages arising out of use of this information and specifically disclaim any implied warranties or merchantability or fitness for any particular use.

Published by : Laxmi Publications Pvt Ltd., 113, Golden House, Daryaganj, New Delhi-110 002.
Tel: 43532500, E-mail: info@laxmipublications.com

DEM-2232-62.00-SOFTWARE ENGG C-124 C-00256/04/19
Typeset at: Kalyani Computers, Delhi Printed at: Ajit Printing Press, Delhi
Edition : 2017.

# CONTENTS


| Units | Page No. |
|---|---|
| I. Software Engineering | 01-25 |
| II. Requirements Analysis | 26-52 |
| III. Designing Software Solutions | 53-91 |
| IV. Software Implementation | 92-103 |
| V. Software Maintenance | 104-128 |

# SYLLABUS

## SOFTWARE ENGINEERING

## C-124

**Unit-I:**

**Software Engineering :** Definition and paradigms, A generic view of software engineering.

**Unit-II:**

**Requirements Analysis :** Statement of system scope, isolation of top level processes and entitles and their allocation to physical elements, refinement and review. Analyzing a problem, creating a software specification document, review for correctness, consistency, and completeness.

**Unit-III:**

**Designing Software Solutions:** Refining the software Specifications; Application of fundamental design concept for data, architectural and procedural designs using software blue print methodology and object oriented design paradigm; creating design document : Review of conformance to software requirements and quality.

**Unit-IV:**

**Software Implementation:** Relationship between design and implementation: Implementation issues and programming support environment; Coding the procedural design, Good coding style & review of correctness and readability.

**Unit-V:**

**Software Maintenance :** Maintenance as part of software evaluation, reasons for maintenance, types of maintenance (Perceptive, adoptive, corrective), designing for maintainability, techniques for maintenance. Comprehensive examples using available software platforms/case tools.

# UNIT I SOFTWARE ENGINEERING

NOTES

## ★ STRUCTURE ★

1.0 Learning Objectives
1.1 Introduction
1.2 Introduction to Software Engineering
1.3 Software Components
1.4 Software Characteristics
1.5 Software Crisis
1.6 Software Myths
1.7 Software Applications
1.8 Software Engineering Processes
1.9 Evolution of Software
1.10 Some Terminologies
1.11 Program Versus Software Products
1.12 A Generic View of Software Engineering
- *Summary*
- *Review Questions*
- *Further Readings*

## 1.0 LEARNING OBJECTIVES

*After studying this unit, you will be able to:*

- explain software engineering and its various components, characteristics and applications
- describe evolution of software
- giv[illegible] generic view of software engineering.

## 1.1 INTRODUCTION

NOTES

Software is described by its capabilities. The capabilities relate to the functions it executes, the features it provides and the facilities it offers. Software written for sales-order processing would have different functions to process different types of sales orders from different market segments. The features, for example, would be to handle multicurrency computing, updating of product, sales and tax status in MIS reports and books of accounts. The facilities could be printing of sales orders, e-mail to customers, reports and advice to the stores department to dispatch the goods. The facilities and features could be optional and based on customer choice.

The software is developed keeping in mind certain hardware and operating system considerations, known as platform. Hence, software is described along with its capabilities and the platform specifications that are required to run it.

### Definition of Software

Software is a set of instructions to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the software. It also includes a set of documents, such as the software manual, meant for users to understand the software system. Today's software comprises the Source code, Executables, Design documents, Operations and System manuals and Installation and Implementation manuals.

Software is:

(*i*) Instructions (computer programs) that when executed provide desired function and performance.

(*ii*) Data structures that enable the programs to adequately manipulate information.

(*iii*) Documents that describe the operation and use of the programs

*Or*

The term software refers to the set of computer programs, procedures, and associated documents (flowcharts, manuals, etc.) that describe the programs and how they are to be used. To be precise, software means a collection of programs whose objective is to enhance the capabilities of the hardware.

Or

*Definition of Software given by IEEE*

*"Software is the collection of computer programs, procedure rules and associated documentation and data".*

NOTES

## Importance of Software

Computer software has become a driving force.

- It is engine that drives business decision-making.
- It serves as the basis for modern scientific investigation and engineering problem solving.
- It is embedded in all kinds of systems like transportation, medical, telecommunications, military, industrial processes, entertainment, office products etc.

It is important as it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture and our everyday activities. Software impact on our society and culture is significant. As software importance grows, the software community continually attempts to develop technologies that will make it easier, faster and less expensive to build high-quality computer programs.

# 1.2 INTRODUCTION TO SOFTWARE ENGINEERING

Few important definitions given by several authors and institutions are as under:

## IEEE Comprehensive Definition

"Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, *i.e.*, the application of engineering to software".

## According to Barry Boehm

"Software Engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation".

NOTES

### According to Fairley

"Software Engineering is a methodological and managerial discipline concerning the systematic production and maintenance of software products that are developed and maintained within anticipated and controlled time and cost limits".

### According to Fritz Bauer

"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".

### According to Somerville

"Software Engineering is concerned with the theories, methods and tools that are needed to develop the software products in a cost effective way".

***According to Dennis:*** *Software engineering is the application of principles, skills and art to design and construction of programs and systems of programs.*

***According to Morven Gentleman:*** *Software engineering is the use of methodologies, tools and techniques to resolve the practical problem that arise in the construction, deployment, support and evolution of software.*

***According to Stephen Schach:*** Software engineering is a discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements.

***According to Pomberger and Blaschck:*** Software engineering is the practical application of scientific knowledge for the economical production and use of high quality software.

***According to Rafael J. Barros:*** Software engineering is the application of methods and scientific knowledge to create practical cost effective solution for the design, construction, operation and maintenance of software and associated products in the service of mankind.

### Other Definitions

*"Software Engineering deals with cost effective solutions to practical problems by applying scientific knowledge in building software artifacts in the service of mankind".*

*Or*

*"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".*

*Or*

*"Software Engineering is a discipline whose aim is the production of fault free software that satisfies the user's needs and that is delivered on time and within budget".*

*Or*

*"The term Software Engineering refers to a movement, methods and techniques aimed at making software development more systematic".* Software methodologies like the OMG's UML and software tools (CASE tools) that help developer's model application designs and then generate code are all closely associated with Software Engineering.

*Or*

*"Software Engineering is an engineering discipline which is concerned with all aspects of software production".*

NOTES

## Software Engineering Principles

The principles deal with both the process of software engineering and the final product. The right process will help produce the right product, but the desired product will also affect the choice of which process to use. A traditional problem in software engineering has been the emphasis on either the process or the product to the exclusion of the other. Both are important.

The principles we develop are general enough to be applicable throughout the process of software construction and management. Principles, however, are not sufficient to drive software development. In fact, they are general and abstract statements describing desirable properties of software processes and products. But, to apply principles, the software engineer should be equipped with appropriate methods and specific techniques that help incorporate the desired properties into processes and products.

In principle, we should distinguish between methods and techniques. Methods are general guidelines that govern the execution of some activity; they are rigorous, systematic, and disciplined approaches. Techniques are more technical and mechanical than methods; often, they also have more restricted applicability. In general, however, the difference between the two is not sharp. We will therefore use the two terms interchangeably.

Sometimes, methods and techniques are packaged together to form a methodology. The purpose of a methodology is to promote a certain

approach to solving a problem by preselecting the methods and techniques to be used. Tools, in turn, are developed to support the application of techniques, methods, and methodologies.

NOTES

Figure 1 shows the relationship between principles, methods, methodologies, and tools. Each layer in the figure is based on the layer(s) below it and is more susceptible to change, due to passage of time. This figure shows clearly that principles are the basis of all methods, techniques, methodologies, and tools.

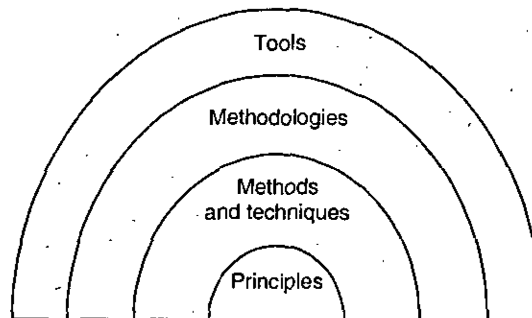



**Fig. 1** *Relationship between Principles, Techniques, Methodologies, and Tools*

## 1.3 SOFTWARE COMPONENTS

A **software component** is a system element offering a predefined service and able to communicate with other components. Clemens Szyperski and David Messerschmitt give the following five criteria for what a software component shall be to fulfill the definition:

- Multiple-use
- Non-context-specific
- Composable with other components
- Encapsulated *i.e.,* non-investigable through its interfaces
- A unit of independent deployment and versioning

A simpler definition can be: A component is an object written to a specification. It does not matter what the specification is: COM, Java Beans, etc., as long as the object adheres to the specification. It is only by adhering to the specification that the object becomes a component and gains features like reusability and so forth.

Software components often take the form of objects or collections of objects (from object-oriented programming), in some binary or textual form, adhering to some Interface Description Language (IDL) so that the component may exist autonomously from other components in a computer.

When a component is to be accessed or shared across execution contexts or network links, some form of serialization (also known as marshalling) is employed to turn the component or one of its interfaces into a bit stream.

NOTES

It takes significant effort and awareness to write a software component that is effectively reusable. The component needs:

- to be fully documented;
- more thorough testing;
- robust *input validity checking;*
- to pass back useful error messages as appropriate;
- to be built with an awareness that it will be put to unforeseen uses;
- a mechanism for compensating developers who invest the (substantial) effort implied above.

## 1.4 SOFTWARE CHARACTERISTICS

The key characteristics of software are as under:

### 1. Most Software is Custom-Built, Rather than being Assembled from Existing Components

Most software continues to be custom built, although recent developments tend to be component-based. Modern reusable components encapsulate both data and the processing applied to data, enabling the software engineer to create new applications from reusable part. For example, today GUI is built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

### 2. Software is Developed or Engineered; it is not Manufactured in the Classical Sense

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent for software. Both activities depend on people, but the relationship between people applied and work accomplished is

entirely different. Both require the construction of a "product". But the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

NOTES

### 3. Software is Flexible

We all feel that software is flexible. A program can be developed to do almost anything. Sometimes, this characteristic may be the best and may help us to accommodate any kind of change. Reuse of components from the libraries help in reduction of effort. Nowadays, we reuse not only algorithms but also data structures.

### 4. Software doesn't Wear Out

There is a well known **"bath-tub curve"** in reliability studies for the hardware products. Figure 2 depicts failure intensity as a function of time for hardware. The relationship, often called the **"bath-tub curve".** Note that, wear out means process of loosing the material.

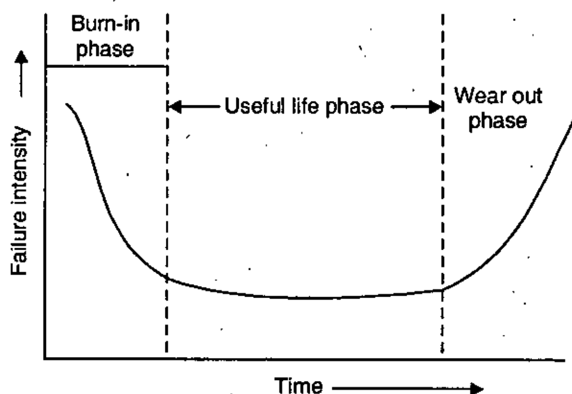



**Fig. 2** *Bath-tub Curve*

There are three phases for the life of a hardware product. Initial phase is burn-in phase, where failure intensity is high. It is expected to test the product in the industry before delivery. Due to testing and fixing faults, failure intensity will come down initially and may stabilize after certain time. The second phase is the useful life phase where failure intensity is approximately constant and is called useful life of a product. After few years, again failure intensity will increase due to wearing out of components. This phase is called wear out phase. We do not have this phase for the software, as it does not wear out. The curve for software is given in Fig. 3

NOTES

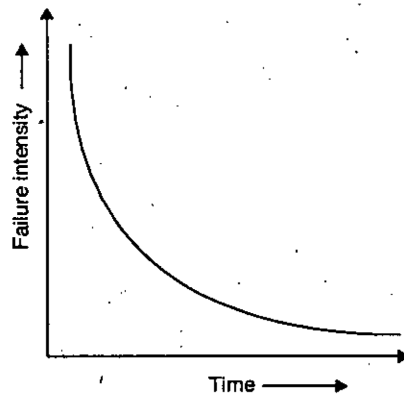



**Fig. 3** *Software Curve*

Important point is software becomes reliable overtime instead of wearing out. It becomes obsolete, if the environment, for which it was developed, changes. Hence, software may be retired due to environmental changes, new requirements, new expectations, etc.

## 1.5 SOFTWARE CRISIS

The software crisis has been with us since 1970s. As per the latest IBM report, "31% of the projects get cancelled before they are completed, 53% over-run their cost-estimates by an average of 189% and for every 100 projects, there are 94 restarts".

When software was developing then during development many problems are raised up, that set of problem is known as software crisis. When software is developing then on the different steps of development, problems are encountered associated with those steps. Now we will discuss the problem, and causes of software crisis encountered on different stages of software development.

### Problems

1. Schedule and cost estimates are often grossly inaccurate.
2. The "Productivity" of software people hasn't kept pace with the demand for their services.
3. The quality of software is sometimes less than adequate.
4. With no solid indication of productivity, we can't accurately evaluate the efficiency of new tools, methods or standards.
5. Communication between customer and software developer is often poor.

6. The software maintenance task devours the majority of all software rupees.

## Causes

NOTES

1. Quality of software is not good because most of the developer use the historical data to develop the software.
2. If there is delay in any process or stage (*i.e.*, analysis, design, coding and testing) then scheduling does not match with actual timing.
3. Communication between managers and customers, software developers, support staff etc., can breakdown because the special characteristics of software and the problems associated with its development are misunderstood.
4. The software people responsible for tapping that potential often change when it is discussed and resist change when it is introduced.

## Software Crisis in the Programmer's Point of View

1. Problem of compatibility.
2. Problem of portability.
3. Problem in documentation.
4. Problem of piracy of software.
5. Problem in coordination of work of different people.
6. Problem of maintenance in proper manner.

## Software Crisis in the User's Point of View

1. Software cost is very high.
2. Customers are moody or choosy.
3. Hardware goes very down.
4. Lack of specialization in development.
5. Problem of different versions of software.
6. Problem of views.
7. Problem of bugs.

NOTES

## 1.6 SOFTWARE MYTHS

1. If we get behind schedule, we can add more programmers and catch up.
2. If I decide to outsource the software project to a third party, I can just relax and let that firm build it.
3. Project requirement continuously changes, but changes can be easily accommodated because software is flexible.
4. The only deliverable work product for a successful project is the working program.
5. Software with more features is better software.
6. Once we write the program and get it to work, our job is done.
7. Until I get the program running, I have no way of assessing its quality.
8. Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.
9. A general statement of objectives is sufficient to begin writing programs.
10. We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?

## 1.7 SOFTWARE APPLICATIONS

Software applications are grouped into eight areas for convenience as shown in Figure 4.

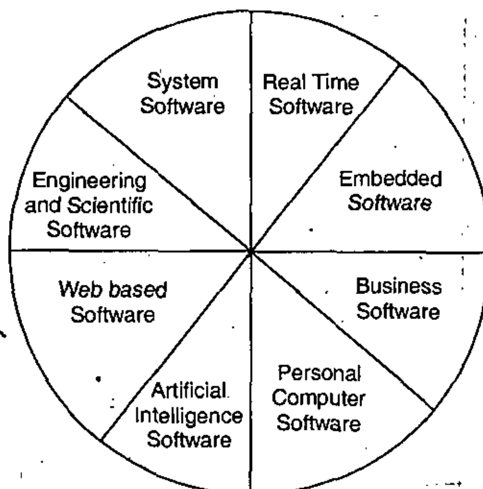



**Fig. 4** *Software Applications*

NOTES

### 1. System Software

System software is a collection of programs used to run the system as an assistance to use other software programs. The compilers, editors, utilities, operating system components, drivers and interfaces, *Assemblers, compilers, linkers* and *loaders* are examples of system software. This software resides in the computer system and consumes its resources. A computer system without system software cannot function.

System software directly interacts with the hardware, heavy usage by multiple users, concurrent operations that requires scheduling, resource sharing and sophisticated process management, complex data structures and multiple external interfaces.

### 2. Real Time Software

Real time software deals with changing environment. First it collects the input and convert it from analog to digital, control component that responds to the external environment, perform the action in the last.

The software is used to monitor, control and analyze real world events as they occur. Examples are Rocket launching, games etc.

### 3. Embedded Software

Software, when written to perform certain functions under control conditions and further embedded into hardware as a part of large systems, is called embedded software.

The software resides in Read-Only-Memory (ROM) and is used to control the various functions of the resident products. The products could be a car, washing machine, microwave oven, industrial processing products, gas stations, satellites and a host of other products, where the need is to acquire input, analyze, identify status, decide and take action that allows the product to perform in a predetermined manner. Because of their role and performance, they are also termed intelligent software.

### 4. Business Software

Software designed to process business applications is called business software. Business software could be a data-and information-processing application. It could drive the business process through transaction processing in online or in real-time mode.

This software is used for specific operations such as accounting package, Management information system, payroll package, inventory management. Business software restructures existing data in order to facilitate business

operations or management decision making. It also encompasses interactive computing. It is an integrated software related to a particular field.

NOTES

### 5. Personal Computer Software

The personal computer software market has burgeoned over the past two decades.

Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network or database access are only a few of hundreds of applications.

### 6. Artificial Intelligence Software

Artificial Intelligence Software uses non-numerical algorithms, which use the data and information generated in the system, to solve the complex problems. These problem scenarios are not generally amenable to problem-solving procedures, and require specific analysis and interpretation of the problem to solve it.

Application within this area include robotics, expert system, pattern recognition (image and voice), artificial neural networks, theorem proving and game playing, signal processing software.

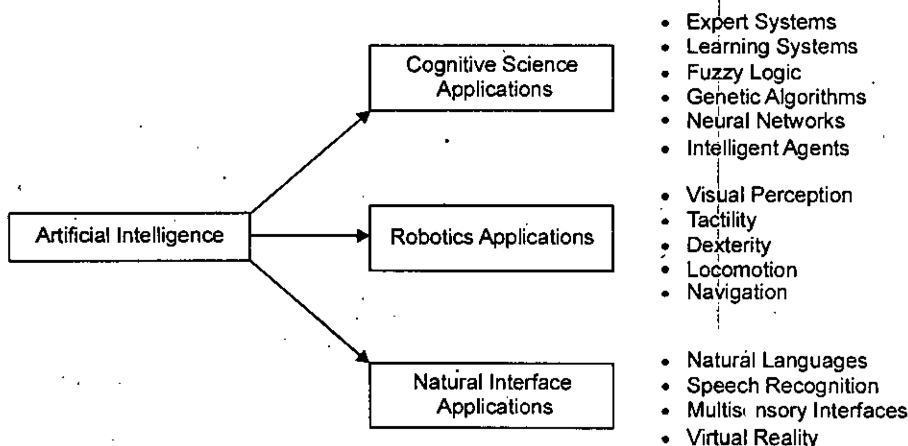



**Fig. 5** *Application Areas of Artificial Intelligence*

### 7. Web-based Software

Web-based software is the browsers by which web pages are processed *i.e.*, HTML, Java, CGI, Perl, DHTML etc.

NOTES

### 8. Engineering and Scientific Software

Design, engineering of scientific software's deal with processing requirements in their specific fields. They are written for specific applications using the principles and formulae of each field. In this type application areas are:

*Astronomy, volcanology, molecular biology, computer aided design (e.g., auto CAD software) system simulations.*

These software's service the need of drawing, drafting, modeling, lead calculations, specifications-building and so on. Dedicated software's are available for stress analysis or for analysis of engineering data, statistical data for interpretation and decision-making. CAD/CAM/CAE packages, SPSS, MATLAB, circuit analyzers are typical examples of such software.

## 1.8 SOFTWARE ENGINEERING PROCESSES

### Process

***"A process is a series of steps involving activities, constraints and resources that produce an intended output of some kind".***

Any process has the following characteristics.

1. The process prescribes all of the major process activities.
2. The process uses resources, subject to a set of constraints (such as a schedule), and produces intermediate and final products.
3. The process may be composed of sub processes that are linked in some way. The process may be defined as a hierarchy of processes, organized so that each sub-process has its own process model.
4. Each process activity has entry and exit criteria, so that we know when the activity begins and ends.
5. The activities are organized in a sequence, so that it is clear when one activity is performed relative to the other activities.
6. Every process has a set of guiding principles that explain the goals of each activity.
7. Constraints or controls may apply to an activity, resource, or product. For example, the budget or schedule may constrain the length of time an activity may take or a tool may limit the way in which a resource may be used.

NOTES

## What is a Software Process?

***"Software process is the related set of activities and processes that are involved in developing and evolving a software system".***

*Or*

***"A set of activities whose goal is the development or evolution of software".***

*Or*

***"A software process is a set of activities and associated results, which produce a software product".***

These activities are mostly carried out by software engineers. There are four fundamental process activities which are common to all software processes. These activities are:

1. **Software specification:** The functionality of the software and constraints on its operation must be defined.
2. **Software development:** The software to meet the specification must be produced.
3. **Software validation:** The software must be validated to ensure that it does what the customer wants.
4. **Software evolution:** The software must evolve to meet changing customer needs.

Different software processes organize these activities in different ways and are described at different levels of detail. The timing of the activities varies, as does the results of each activity. Different organizations may use different processes to produce the same type of product. However, some processes are more suitable than others for some types of application. If an inappropriate process is used, this will probably reduce the quality or the usefulness of the software product to be developed.

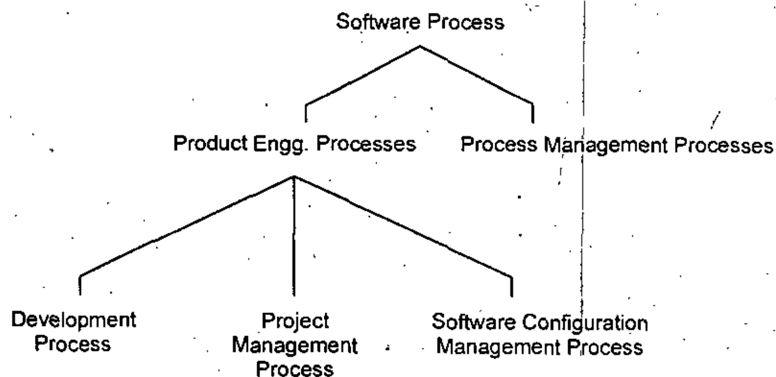



**Fig. 6** *The Software Process*

NOTES

A software process can be characterized as shown in Figure 6. A common process framework is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. A number of task sets-each a collection of software engineering work tasks, project milestones, software work products and deliverables, and quality assurance points-enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities-such as software quality assurance, software configuration management and measurement-overlay the process model. Umbrella activities are independent of any one-framework activity and occur throughout the process.

Thus, the software industry considers software development as a process. According to Booch and Rumbaugh, *"A process defines who is doing what, when and how to reach a certain goal"*? Software engineering is a field, which combines process, methods and tools for the development of software. The concep. of process is the main step in the software engineering approach. Thus, a software process is a set of activities. When those activities are performed in specific sequence in accordance with ordering constraints, the desired results are produced.

## 1.9 EVOLUTION OF SOFTWARE

Software engineering principles have evolved over the past more than fifty years from art to an engineering discipline. It can be shown with the help of the following Figure 7.

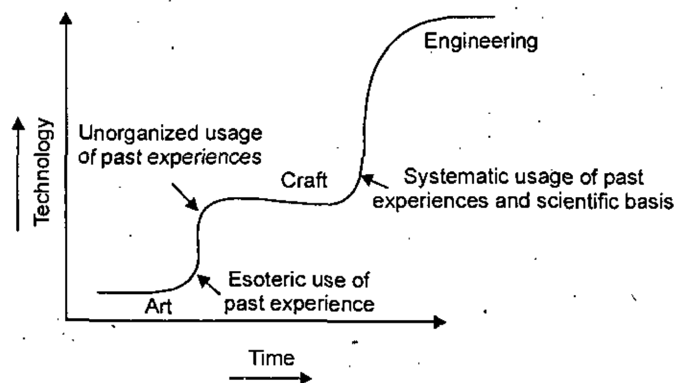



**Fig. 7** *Evolution of Art to an Engineering Discipline*

Development in the field of software and hardware computing make a significant change in the twentieth century. We can devide the software development process into four eras:

NOTES

### 1. Early Era

During the early eras general-purpose hardware became common place. Software, on the other hand, was custom-designed for each application and had a relatively limited distribution. Most software was developed and ultimately used by the same person or organization.

In this era the software are mainly based on (1950–1960)

- Limited Distribution
- Custom Software
- Batch Orientation

### 2. Second Era

The second era to computer system evolution introduced new concepts of human machine interaction. Interactive techniques opened a new world of application and new levels of hardware and software sophistication. Real time software deals with the changing environment and one other is multi-user in which many users can perform or work on a software at a time.

In this era the software are mainly based on (1960–1972)

- Multi-user
- Database
- Real time
- Product Software
- Multiprogramming

### 3. Third Era

In the earlier age the software was custom designed and limited distribution but in this era the software was consumer designed and the distribution is also not limited. The cost of the hardware is also very low in this era.

In this era the software are mainly based on (1973–1985)

- Embedded Intelligence
- Consumer Impact
- Distributed Systems
- Low Cost Hardware

### 4. Fourth Era

NOTES

The fourth era of computer system evolution moves us away from individual computers and computer programs and toward the collective impact of computers and software. As the fourth era progresses, new technologies have begun to emerge.

In this era the software are mainly based on (1987)

- Powerful Desktop Systems
- Expert Systems
- Artificial Intelligence
- Network Computers
- Parallel Computing
- Object Oriented Technology

At this time the concept of software making is object oriented technology or network computing etc.

## 1.10 SOME TERMINOLOGIES

Some terminologies are discussed in these sections which are frequently used in the field of Software Engineering.

### 1. Deliverables and Milestones

Different ***deliverables*** are generated during software development. The examples are source code, user manuals, operating procedure manuals etc.

The ***milestones*** are the events that are used to ascertain the status of the project. Finalization of specification is a milestone. Completion of design documentation is another milestone. The milestones are essential for project planning and management.

### 2. Product and Process

What is delivered to the customer is called a ***product***. It may include source code specification, document, manuals, documentation etc. Basically, it is nothing but a set of deliverables only.

***Process*** is the way in which we produce software. It is the collection of activity that leads to (a part of) a product. An efficient process is required to produce good quality products.

If the process is weak, the end product ... obsessive over reliance on process is also dangerous.

### 3. Measures, Metrics and Indicators

In software engineering *measures* provides a quantitative indication of amount, dimension, capacity or size of given attribute of a product.

The *metrics* is a quantitative measures of the degree to which a system, component, or process possesses a given attribute of a product. An *indicator* is a combination of metrics.

Measurement occurs as the result of the collection of one or more data points *e.g.*, a number of module reviews are investigated to collect measures of the number of errors in each module.

NOTES

## 1.11 PROGRAM VERSUS SOFTWARE PRODUCTS

### Program

A program is a subset of software and it becomes software only if documentation and operating procedure manuals are prepared. Program is a combination of source code and object code.

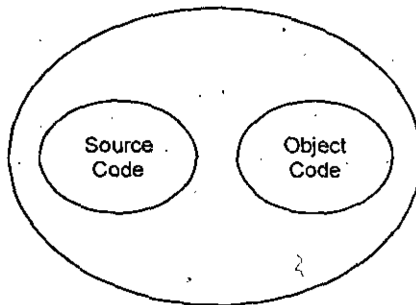



**Fig. 8** *Program = Source Code + Object Code*

### Software Products

A software product consists not only of the program code but also of all the associated documents such as the requirements specification documents, the design documents, the test document, the operating procedures which includes user manuals and operational manuals.

NOTES

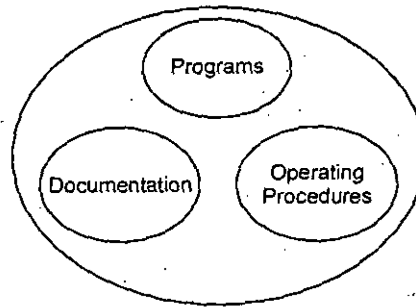



**Fig. 9** *Software = Program + Documentation + Operating Procedures*

### Difference between Program and Software Products

The various differences between a program product and a software produc are given in the tabular form as follows:

| *Programs* | *Software Products* |
|---|---|
| 1. Programs are developed by individuals for their personal use | 1. A software product is usually developed by a group of engineers working in a team |
| 2. Usually small in size | 2. Usually large in size |
| 3. Single user | 3. Large number of users |
| 4. Single developer | 4. Team of developers |
| 5. Lacks proper documentation | 5. Good documentation support |
| 6. Ad hoc development | 6. Systematic development |
| 7. Lack of user interface | 7. Good user interface |
| 8. Have limited functionality | 8. Exhibit more functionality |

## 1.12 A GENERIC VIEW OF SOFTWARE ENGINEERING

### Software Engineering as a Layered Technology

Process layer provides a farmework for effective use of software technologie: Forms a basis for management control and establishes context in whic technical methods are applied, work products produced, milestone established, quality is ensured and change is managed.

### Process Framework

Identifies a small number of framework activities that are applicable t all software projects.

In addition the framework encompasses umbrella activities that are applicabl across the software process.

NOTES

## Generic Process Framework Activities

Each framework activity is populated by a set of software engineering actions. An action, e.g., design is a collection of related tasks that produce a major software engineering work product.

Communication—lots of communication and collaboration with customer. Encompasses requirements gathering.

Planning—establishes plan for software engineering work that follows. Describes technical tasks, likely risky, required resources, works products and a work schedule.

Modeling—encompasses creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements.

Construction—code generation and testing.

Deployment—software, partial or complete, is delivered to the customer who evaluates it and provides feedback.

## Umbrella Activities

Framework is augmented by a number of umbrella activities. Typical ones are:

- Software project tracking and control—allows software team to assess progress against project plan and take necessary action to maintain schedule.
- Risk management—assess risk that may effect the outcome of the project or the product quality.
- formal technical reviews—assess software engineering work products to uncover and remove errors before they are propagated to the next action or activity.
- Measurement—defines and collects process, project and product measures that assist team in developing software.
- Software configuration management—manages the effect of change throughout the software process.
- Reusability management—defines criteria for work product reuse and establishes mechanism to achieve reusable components.
- Work product preparation and production – included work activities required to create work products such as documents, logs, forms and lists.

All processes can be described with the above framework. Intelligent adaptation of any process model to the problem, team, project, and organisational culture is essential.

## STUDENT ACTIVITY

1. What is software engineering?

2. What do you understand by software crisis?

3. Define software process.

4. Explain software components.

## SUMMARY

NOTES

- "Software is the collection of computer programs, procedure rules and associated documentation and data".
- Application software sits a top of system software because it needs help of system software to run.
- Operating system is a number of utilities for managing disk printers, other devices.
- Word processors use a computer to create, edit, and print documents.
- A spreadsheet is a table of values arranged in rows and columns.
- Presentation graphics is often called business graphics.
- A DBMS is a collection of programs that enable you to store, modify, and extract information from a database.
- A **software component** is a system element offering a predefined service and able to communicate with other components.
- System software is a collection of programs used to run the system as an assistance to use other software programs.
- Real time software deals with changing environment.
- Business software could be a data and information-processing application.
- The personal computer software market has burgeoned over the past two decades.
- Artificial Intelligence Software uses non-numerical algorithms, which use the data and information generated in the system, to solve the complex problems.
- ***Process*** is the way in which we produce software.
- Software is a set of instructions to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the software.
- "Software Engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation".

**NOTES**

- "The term Software Engineering refers to a movement, methods and techniques aimed at making software development more systematic".
- A component is an object written to a specification.
- Software, when written to perform certain functions under control conditions and further embedded into hardware as a part of large systems, is called embedded software.
- Web-based software is the browsers by which web pages are processed.
- A process is a series of steps involving activities, constraints and resources that produce an intended output of some kind.
- A software process is a set of activities and associated results, which produce a software product.

## REVIEW QUESTIONS

1. Define software.
2. What do you mean by the term "Software Engineering"? Describe the evolving role of software.
3. What are the different myths and realities about the software?
4. What is bath-tub curve?
5. Discuss the characteristics of the software.
6. What characteristics of software make it different from other engineering products (for example hardware)?
7. Explain some characteristics of software?
8. Comment on the statement "software does not wear out".
9. Discuss about the evolution of software engineering as a subject in the last 50 years.
10. What are the different software components?
11. 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?
12. What is software crisis? Give the problems of software crisis.

13. What do you mean by software myths?
14. Explain in detail software engineering process.
15. Distinguish between a program and a software product.
16. Define the followings:

    (a) Milestones (b) Product

    (c) Measures (d) Metrics

17. Explain the importance of software.
18. Discuss about different software applications.

NOTES

## FURTHER READINGS


1. **Software Engineering**, Bharat Bhushan Agarwal, Sumit Prakash Tayal, Firewall Media.
2. **Software Engineering**, D. Sunder, University Science Press.

# UNIT II REQUIREMENTS ANALYSIS

NOTES

★ STRUCTURE ★

2.0 Learning Objectives
2.1 Introduction
2.2 Functional and Non-Functional Requirements
2.3 User, System and Domain Requirements
2.4 Requirements of Engineering Process
2.5 SRS Document
2.6 IEEE Standards for SRS
2.7 SRS Validation
2.8 Components of SRS
2.9 Characteristics of SRS
2.10 Goals of SRS Document
2.11 Benefits of Involving Technical Writers in SRS
2.12 SRS Document Template
- *Summary*
- *Review Questions*
- *Further Readings*

## 2.0 LEARNING OBJECTIVES

*After studying this unit, you will be able to:*

1. describe functional and non-functional requirements
2. explain requirements of engineering process.
3. Illustrate SRS document and its various components, characteristics and goals.

## 2.1 INTRODUCTION

Requirements are defined as descriptions and specifications of a system. It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

System requirements may be either functional or non-functional requirements. In addition, requirements are classified under user requirements, system requirements and domain requirements.

NOTES

## 2.2 FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

### Functional Requirements

(*a*) Describe functionality or system services.

(*b*) Depend on the type of software, expected users and the type of system where the software is used.

(*c*) Functional user requirements may be high-level statements of what the system should do and functional system requirements should describe the system services in detail.

#### *Examples of Functional Requirements*

- The user shall be able to search either all of the initial set of databases or select a subset from it.
- The system shall provide appropriate viewers for the user to read documents in the document store.
- Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.
- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

### Non-functional Requirements

Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

(*a*) Define system properties and constraints *e.g.*, reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.

(*b*) Process requirements may also be specified mandating a particular case system, programming language or development method.

(*c*) Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

## Non-functional Classifications

NOTES

(a) *Product requirements.* Requirements, which specify that the delivered product must behave in a particular way *e.g.,* execution speed, reliability, etc.

(b) *Organisational requirements.* Requirements, which are a consequence of organisational policies and procedures, *e.g.,* process standards used, implementation requirements, etc.

(c) *External requirements.* Requirements which arise from factors which are external to the system and its development process *e.g.,* interoperability requirements, legislative requirements, etc.

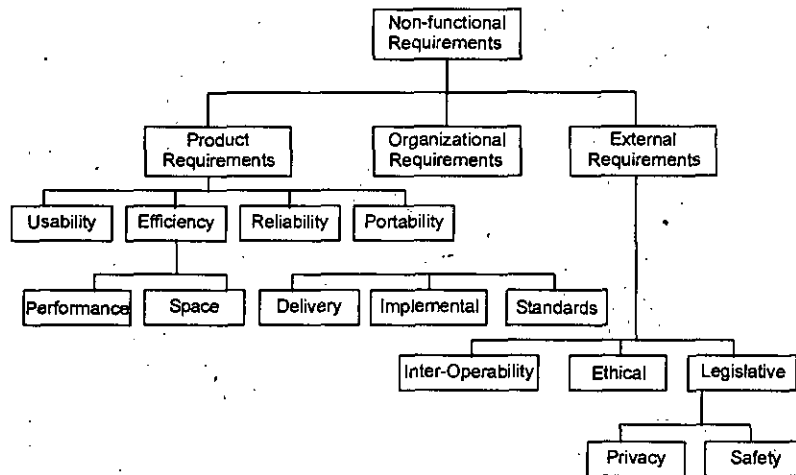



**Fig. 1** *External Requirement Specification*

### *Non-functional Requirements Example*

- *Product requirement.* It shall be possible for all necessary communication between the system and the user to be expressed in the standard Ada character set.
- *Organisational requirement.* The system development process and deliverable documents shall conform to the process and deliverables defined in IEEE STANDARD-95 FORMAT.
- *External requirement.* The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

## 2.3 USER, SYSTEM AND DOMAIN REQUIREMENTS

**User Requirements:** Statements in natural language plus diagrams of the services the system provides and its operational constraints.

**System Requirements:** A structured document setting out detailed descriptions of the system services. Written as a contract between client and contractor.

**Software Specification:** A detailed software description, which can serve as a basis for a design or implementation. Written for developers. User requirements should describe functional and non-functional requirements so that they are understandable by system users who do not have detailed technical knowledge. User requirements are defined using natural language, tables and diagrams.

**System Requirements:** More detailed specifications of user requirements serve as a basis for designing the system may be used as part of the system contract system requirements may be expressed using system models.

**Domain Requirements:** Derived from the application domain and describe system characteristics and features that reflect the domain. May be new functional requirements, constraints on existing requirements or define specific computations. If domain requirements are not satisfied, the system may be unworkable.

## 2.4 REQUIREMENTS OF ENGINEERING PROCESS

It involves the following tasks:

1. Requirements Elicitation and Analysis
2. Requirements Definition and Specification Document
3. System Modeling
4. Requirements Validation and Management.

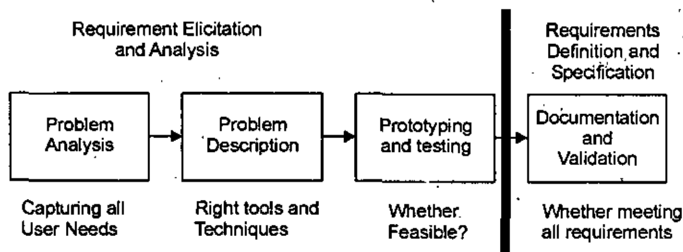



**Fig. 2** *Requirement Specifications*

### Requirements Elicitation and Analysis

It is a critical process in software development. It is conducted with the following objectives in mind:

NOTES

1. Identify the customer's need;
2. Evaluate the system concept for feasibility;
3. Perform economic and technical analysis;
4. Allocate functions to hardware, software, people, database and other system elements.
5. Establish cost and schedule constraints.
6. Create a system definition that forms the foundation for all subsequent engineering work. A variety of techniques are used to determine what the users and customers really want.

- **Identification of need:** The analyst (system engineer) meets with the customer and the end user with the intent of understanding the product's objectives and to define goals required meeting the objectives.

  The analyst then gathers the supplementary information likes technology availability, resources required, and bounds placed on costs and schedule etc. The overall information gathered are specified in a system concept document. It demands intense communication between the customer and the analyst.

- **Feasibility study:** Involves the study of economic feasibility, technical feasibility, legal feasibility and alternatives.

  Economic feasibility an evaluation of development cost weighted against the ultimate income or benefit derived from the developed system or product.

  Technical feasibility is the study of function, performance and constraints that may affect the ability to achieve an acceptable system. The considerations that are normally associated with technical feasibility include development risk, resource availability and technology capabilities.

  Legal feasibility encompasses a broad range of concerns that includes contracts, liability, infringement and myriad other traps frequently unknown to technical staff. Alternatives involve an evaluation of alternative approaches to the development of the system or product. The study is reviewed first by the project management and then by the upper management. The study may result in a 'go'/ 'no-go' decision.

- **Economic analysis:** The most important information contained in a feasibility study is the cost-benefit analysis—an assessment of the economic justification for a computer based system project. It delineates costs for project development and weights them against tangible and intangible benefits of a system.

- **Technical analysis:** The analyst evaluates the technical merits

of the system concept at the same time collecting additional information about performance, reliability, maintainability and producibility. It assesses the technical viability of the proposed system, technologies required to accomplish system function and performance, new material, methods, algorithms or processes required etc. The results obtained form the basis for another 'go' or 'no-go' decision on the system.

- **System specification:** It is a document that serves as the foundation for hardware engineering, software engineering, data base engineering and human engineering. It describes the function and performance of a computer based system and the constraints that will govern its development. The system specification also describes the information (data and control) that is input to and output from the system.

## Requirement Definition and Specification Document

The requirement definition document contains a record of the requirements in the customers' terms and describes what the customer would like to have. The document's outline is given below:

- General purpose of the system—Outline.
- Background and objectives of system development.

Brief description of the approach along with constraints implied by customers on the development. Detailed characteristics of the proposed system along with system boundaries and interfaces across the various modules. A complete list of data elements, classes and their characteristics is given. Then detailed relationships among data and functions, as well as the input and output to each process and function are discussed. Finally, the environment in which the system will operate is discussed and the discussion includes the special requirements for support, security, privacy, hardware and software.

Software requirement specification document is a technical specification of requirements for the software product. The goal of software requirement definition is to completely and consistently specify the technical requirements for the software product in a concise and unambiguous manner. It is based on the system definition document.

The format of the specification document is given below:

1. Product Overview and Summary
2. Development, Operating and Maintenance Environments
3. External Interfaces and Data Flow
4. Functional Requirements
5. Performance Requirements

NOTES

6. Exception Handling
7. Early Subsets and Implementation Priorities
8. Foreseeable Modification and Enhancements
9. Acceptance Criteria
10. Design hints and Guidelines
11. Cross Reference Index
12. Glossary of Terms.

Sections 1 and 2 present an overview of product features and summarizes the processing environments for development, operation and maintenance of the product.

Section 3 specifies the externally observable characteristics of the software product. It includes user displays and report formats, a summary of user commands and report options, data flow diagrams and a data dictionary. High level data flow diagrams and a data dictionary are derived in this section.

Section 4 specifies the functional requirements for the software product. Functional requirements are expressed in relational and state-oriented notations that specify relationships among inputs, actions and outputs.

Performance characteristics such as response time for various activities, processing time for various processes, throughput, primary and secondary memory constraints, required telecommunication bandwidth and special issues like security constraints, reliability requirements etc., are specified in section 5.

Exception handling, including actions to be taken and the messages to be displayed in response to events are described in section 6. Categories of exceptions include temporary resource failures, out of range-input data, capacity overload etc.

The early subsets and implementation priorities for the system under development are discussed in section 7. Software products are developed as a series of successive versions and the initial version may be the prototype demonstrating basic functions and capabilities. Each successive version can incorporate the capabilities of previous versions and provide additional processing functions.

Foreseeable modifications and enhancements may be incorporated in section 8.

The software product acceptance criteria are specified in section 9. Acceptance criteria specify functional and performance tests that must be performed, the standards to be applied to source code, internal documentation and external documents such as design specifications, test plans, user manual, installation and maintenance procedures.

Section 10 contains design hints and guidelines. It is concerned with

ınctional and performance aspects of the software product.

ection 11 relates to the sources of information used in deriving the ɘquirements. Knowing the sources of specific requirements permits erification and re-examination of requirements, constraints and assumptions.

ection 12 provides the definition of terms that may be unfamiliar to the ustomer and the product developer.

### *esirable Properties of Software Requirements Specification*

he requirement document should be Correct; Complete; Consistent; nambiguous; Functional; Verifiable; Traceable and Easily Changed. An ıcorrect or incomplete set of requirements can result in a software roduct, which does not satisfy the customer needs. An inconsistent pecification states contradictory requirements in different parts of the ocument resulting in different interpretations by different people.

oftware requirements should be functional in nature; *i.e.,* they should escribe what is required without implying how the system will meet its ɘquirements. Requirements must be verifiable from two points of view; must be possible to verify that the requirements satisfy the customer's eeds and it must be possible to verify that the subsequent work products atisfy the requirements. Finally, the requirements should be indexed, egmented and cross-referenced to permit easy use and easy modification.

## ystem Modeling

lodeling helps in gaining better understanding of the actual entity to be uilt. The analyst creates models of the system (prototyping) in an effort ɔ better understand data and control flow, functional processing and ehavioral operation and information content. In software, one must be apable of modeling the information that software transforms (information ıodel), the functions (and sub functions) that enable the transformation o occur (functional model) and the behavior of the system as the ransformation is taking place (behavioral model).

## equirement Validation and Management

**equirements Validation:** Requirements Validation is the process of etermining that the specification is consistent with the requirements efinition *i.e.,* validations make sure that the requirements will meet he customers' needs. Validation usually involves two steps, each of which nsures trace ability between the two requirements document. First, nake sure that each specification can be traced to a requirement in the efinition document. Next, check the definition to validate that each equirement is traceable to the specification. The techniques that can be dopted are given below:

NOTES

| Requirement and Validation Techniques | |
|---|---|
| Manual Techniques | Reading Cross Referencing Interviews Reviews Checklists Models to Check Functions Scenarios Mathematical Proofs. |
| Automated Techniques | Automated Cross-referencing Automated Models to Enact Functions Prototypes. |

A simple way to check the requirements is to perform a requirements review. Review is to be conducted by both the software developer and customer. As the specification forms the foundation for design and subsequent software engineering activities, care need to be taken in conducting the review. The review is conducted both at the macroscopic and detailed levels. The reviewers attempt to ensure that the specification is complete, consistent and accurate. Once, the review is complete, the software requirement specifications are signed-off by both the customer and developer.

**Requirements Management:** Requirements Management is a set of activities that help the software project team to identify, control and track requirements and changes to requirements at any time as the project proceeds. Many of the activities are identical to the 'Software Configuration Management'.

(SCM)'. Like SCM, each requirement is assigned a unique identifier of the form, <Requirement type> <Requirement #>

The requirement type takes value like F = functional requirement, D = Data requirement, B = Behavioral requirement, I = Interface requirement and P = Output requirement.

Once requirements have been identified, trace-ability tables like the following are developed.

*Features Trace-ability:* Customer observable system/product features. Source trace-ability: Source of each requirement. Dependency trace-ability: Relationship among requirements. Subsystem trace-ability: Categorize as per the sub-systems identified. Interface trace-ability: Relationship to internal and external system interfaces. Trace-ability features are maintained as part of requirements database so that they can be searched to understand how a requirement change will affect different aspects of the system to be developed.

## 2.5 SRS DOCUMENT

This document is generated as output of requirement analysis. The requirement analysis involves obtaining a clear and thorough understanding

f the product to be developed. Thus, SRS should be consistent, correct, ınambiguous and complete, document. The developer of the system can ırepare SRS after detailed communication with the customer. An SRS learly defines the following:

- ***External Interfaces of the system:*** They identify the information which is to flow 'from and to' to the system.
- ***Functional and non-functional requirements of the system:*** They stand for the finding of run time requirements.

'he ***functional requirements*** of the system as documented in the SRS ocument should clearly describe each function, which the system would upport along with the corresponding input and output data set.

'he ***non-functional requirements*** deal with the characteristics of the ystem that cannot be expressed as functions. Examples of non-functional equirements include aspects concerning maintainability, portability, and sability. The *non-functional requirements may also include reliability* ssues, accuracy of results, human-computer interface issues, and constraints n the system implementation.

'here are many ways to structure a requirements document. There is no ingle method that is suitable for all projects. IEEE and US Department f Defense have proposed a candidate format for representing SRS. The eneral outline of SRS is given below:

## )rganization of SRS

1. Introduction
   - Purpose
   - Scope
   - Definitions, Acronyms, and Abbreviations
   - References
   - Overview
2. The Overall Description
   - Product Perspective
     - System Interfaces
     - Interfaces
     - Hardware Interfaces
     - Software Interfaces
     - Communications Interfaces
     - Memory Constraints
     - Operations
     - Site Adaptation Requirements

NOTES

- Product Functions
- User Characteristics
- Constraints
- Operating environment
- User environment
- Assumptions and Dependencies
- Apportioning of Requirements

3. Specific Requirements
   - External interfaces
     (*i*) User interface
     (*ii*) Hardware Interfaces
     (*iii*) Software Interface
     (*iv*) Communication Interface
   - Functions
   - Performance Requirements
   - Logical Database Requirements
   - Design Constraints
     – Standards Compliance
   - Software System Attribute
     – Reliability
     – Availability
     – Security
     – Maintainability
     – Portability
   - Organizing the Specific Requirements
     – System Mode
     – User Class
     – Objects
     – Feature
     – Stimulus
     – Response
     – Functional Hierarchy
   - Additional Comments
4. Supporting Information
   - Table of contents and index
   - Appendices

## Uses of SRS Document

NOTES

The following are few major uses of SRS documents:

1. Project managers base their plans and estimates of schedule, effort and resources on it.
2. Development team needs it to develop product.
3. The testing group needs it to generate test plans based on the described external behaviour.
4. The maintenance and product support staff need is to understand what the software product is supposed to do.
5. The publications group writes documents, manuals, etc., from it.
6. Customers rely on it to know what product they can expect.
7. Training personnel can use it to help develop educational material for software product.
8. The maintenance and product support staff need is to understand what the software product is supposed to do.

# 2.6 IEEE STANDARDS FOR SRS

**IEEE Standards** documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard. Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard.

Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art.

Users are cautioned to check to determine that they have the latest edition of any IEEE Standard. Comments for revision of IEEE Standards

NOTES

are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

### IEEE Recommended Approaches for SRS

This recommended practice describes recommended approaches for the specification of software requirements. It is based on a model in which the result of the software requirements specification process is an unambiguous and complete specification document. It should help

1. Software customers to accurately describe what they wish to obtain;
2. Software suppliers to understand exactly what the customer wants;
3. Individuals to accomplish the following goals:
   (*i*) Develop a standard software requirements specification (SRS) outline for their own organizations;
   (*ii*) Define the format and content of their specific software requirements specifications;
4. Develop additional local supporting items such as an SRS quality checklist, or an SRS writer's handbook.

### Benefits of SRS

To the customers, suppliers, and other individuals, a good SRS should provide several specific benefits, such as the following:

1. ***Establish the basis for agreement between the customers and the suppliers on what the software product is to do.***

   The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs.
2. ***Reduce the development effort.***

   The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

3. ***Provide a basis for estimating costs and schedules.***
   The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.
4. ***Provide a baseline for validation and verification.***
   Organizations can develop their validation and verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.
5. ***Facilitate transfer.***
   The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.
6. ***Serve as a basis for enhancement.***
   Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

NOTES

### IEEE Recommended Practice for Software Requirements Specifications

1. Overview
2. References
3. Definitions
4. Considerations for producing a good SRS
5. The parts of an SRS

## 2.7 SRS VALIDATION

It is extremely important to detect errors in requirements document before going to other phases of system development. The major objective of SRS validation is to ensure that user requirements are complete and correctly recorded in the SRS and it is free from errors. It is also needed to check that the SRS itself is of good quality. Some most common type of errors in SRS is:

1. **Omission.** Some user requirement is not included in SRS. This error directly affects the external completeness of the system.
2. **Inconsistency.** Is due to contradictions in requirements or incompatibility of state requirements.

3. **Incorrect fact.** Some facts recorded in SRS are not correct.
4. **Ambiguity.** Some requirements have multiple meanings.

Besides improving the quality of SRS, SRS validation should uncover and rectify all possible types of errors.

NOTES

## 2.8 COMPONENTS OF SRS

The following requirements are used in specification of SRS:

1. Functional requirements
2. Performance requirements
3. Design constraints
4. External interface requirements

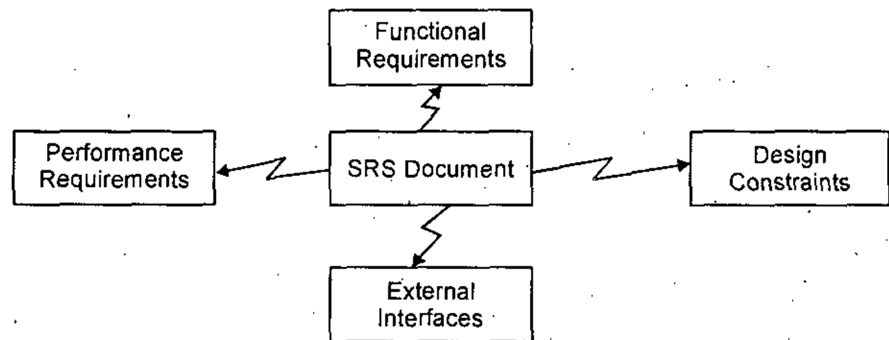



**Fig. 3** *Components of SRS*

### 1. Functional Requirements

Functional requirements specify which outputs should be produced from the given inputs. They describe the relationship between the input and output of the system. For each functional requirement, a detailed description of all the data inputs and their source, the units of measure, and the range of valid inputs must be specified.

All the operations to be performed on the input data to obtain should be specified. This includes specifying the validity checks on the input and output data, parameters affected by the operation, and equations or other logical operations that must be used to transform the inputs into corresponding outputs. For example, if there is a formula for computing the output, it should be specified. Care must be taken not to specify any algorithms that are not part of the system but that may be needed to implement the system. These decisions should be left for the designer. In addition some abnormal input, system behaviour for invalid inputs must be specified.

## 2. Performance Requirements

This part of an SRS specifies the performance constraints on the software system. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements: ***static** and **dynamic**.*

***Static requirements*** are those that do not impose constraint on the execution characteristics of the system. These include requirements like the number of terminals to be supported, the number of simultaneous users to be supported, and the number of files that the system has to process and their sizes. These are also called ***capacity requirements*** of the system.

***Dynamic requirements*** specify constraints on the execution behaviour of the system. These typically include response time and throughput constraints on the system. Response time is the expected time for the completion of an operation under specified circumstances. Throughput is the expected number of operations that can be performed in a unit time. For example, the SRS may specify the number of transactions that must be processed per unit time, or what the response time for a particular command should be. Acceptable ranges of the different performance parameters should be specified, as well as acceptable performance for both normal and peak workload conditions.

## 3. Design Constraints

There are a number of factors in the client's environment that may restrict the choices of a designer. Such factors include standards that must be followed, resource limits, operating environment, reliability and security requirements, and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

(*i*) **Standards Compliance.** This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures. There may be audit-tracing requirements, which require certain kinds of changes, or operations that must be recorded in an audit file.

(*ii*) **Hardware Limitations.** The software may have to operate on some existing or predetermined hardware, thus imposing restrictions on the design. Hardware limitations can include the type of machines to be used, operating system. available on the system, languages supported, and limits on primary and secondary storage.

(*iii*) **Reliability and Fault Tolerance.** Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex

NOTES

and expensive. Recovery requirements must specify what the system should do if some fault occurs. Recovery requirements are often an integral part in the design constraints.

(*iv*) **Security.** Security requirements are particularly significant in defense system and many database systems. Security requirements place restrictions on the use of certain commands, control access to data, provide different kinds of access requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system. Given the current security needs even of common systems, they may also require proper assessment of security threats, proper programming techniques, and use of tools to detect flaws like buffer overflow.

### 4. External Interface Requirements

All the interactions of the software with people, hardware, and other software should be clearly specified. For the user interface, the characteristics of each user interface of the software product should be specified. User interface is becoming increasingly important and must be given proper attention. A preliminary user manual should be created with all user commands, screen formats, an explanation of how the system will appear to the user, and feedback and error messages. Like other specifications these requirements should be precise and verifiable. So, a statement like "the system should be user friendly" should be avoided and statements like "commands should be no longer than six characters" or "commands names should reflect the function they perform" used.

For hardware interface requirements, the SRS should specify the logical characteristics of each interface between the software product and the hardware components. If the software is to execute on existing hardware or on predetermined hardware, all the characteristics of the hardware, including memory restrictions, should be specified. In addition, the current use and load characteristics of the hardware should be given.

The interface requirement should specify the interface with other software the system will use or that will use the system. This includes the interface with the operating system and other applications. The message content and format of each interface should be specified.

## 2.9 CHARACTERISTICS OF SRS

A good SRS document has certain characteristics that must be present to qualify as a good. The characteristics are:

### 1. Correctness

An SRS is correct if every requirement included in the SRS represents something required in the final system.

### 2. Completeness

SRS is complete when it is documented after:

(*i*) The involvement of all types of concerned personnel.

(*ii*) Focusing on all problem, goals and objectives, and not only on functions and features.

(*iii*) Correct definition of scope and boundaries of the software and system.

### 3. Unambiguous

An SRS is unambiguous if and only if every requirement stated has one and only one interpretation. Requirements are often written in natural language, the SRS writer has to be especially careful to ensure that there are no ambiguities. One way to avoid ambiguities is to use some formal requirements specification language. The major disadvantage of using formal languages is the large effort required to write an SRS, the high cost of doing so, and the increased difficulty reading and understanding formally stated requirements (particularly by the users and clients).

### 4. Verifiable

An SRS is verifiable if and only if there exists some cost-effective process that can check whether the final product meets the requirement.

### 5. Modifiable

An SRS is modifiable if its structure and style are such that any necessary change can be made easily while preserving completeness and consistency. Presence of redundancy is a major hindrance to modifiability, as it can easily lead to errors. For example, assume that a requirement is stated in two places and that the requirement later needs to be changed. If only one occurrence of the requirement is modified, the resulting SRS will be inconsistent.

### 6. Traceable

The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended.

(i) **Backward traceability.** This depends upon each requirement explicitly referencing its source in earlier documents.

(ii) **Forward traceability.** This depends upon each requirement in the SRS having a unique name or reference number.

NOTES

### 7. Consistency

Consistency in SRS is essential to achieve correct results across the system. This is achieved by,

(i) Use of standard terms and definitions.
(ii) Consistent application of business rules in all functionality.
(iii) Use of data dictionary.
(iv) Lack of consistency results in incorrect SRS and failure in deliverables to customer.

### 8. Testability

SRS should be written in such a way that it is possible to create a test plan to confirm whether specifications can be met and requirements can be delivered

(i) Considering functional and non-functional requirements.
(ii) Determining features and facilities required for each requirement.
(iii) Ensuring that 'users' and 'stakeholders' freeze the requirement.

### 9. Clarity

An SRS is clear when it has a single interpretation for the author (analysis), the user, the end user, the stakeholder, the developer, the tester and the customer. This is possible if the language of the SRS is unambiguous. Clarity can be ascertained after reviewing the SRS by a third party. It can be enhanced if SRS includes diagrams, models and charts.

### 10. Feasibility

RDD-SRS needs to be confirmed on technical and operational feasibility. SRS many times assumes use of technology and tools based on the information given by their vendors. It needs to be confirmed whether the technology is capable enough to deliver what is expected in SRS. The operational feasibility must be checked through environment checking. It is assumed that sources of data, user capability, system culture, work culture and such other aspects satisfy the expectation of the developer. These must be confirmed before development launch.

NOTES

## 2.10 GOALS OF SRS DOCUMENT

A well designed, well written SRS document accomplishes the following four major goals:

**Feedback to Customer:**

- SRS document provides a feedback to customer.
- It is the customer's assurance that the development organization understands the issues or problems to be solved and the software behaviour necessary to address those problems.
- Therefore, the SRS should be written in natural language, in an unambiguous manner that may also include charts, tables, data flow diagrams, decision tables and so on.

**Problem Decomposition**

- SRS document decomposes the problem into component parts.
- The simple act of writing down software requirements in a well-designed format organizes information, places borders around the problem, solidifies ideas, and helps breakdown the problem into its component parts in an orderly fashion.

**Input to Design Specification**

- SRS document serves as an input to the design specification.
- The SRS also serves as the parent document to subsequent documents, such as the software design specification and statement of work.
- Therefore, the SRS must contain sufficient detail in the functional system requirements so that a design solution can be devised.

**Production Validation Check:**

- SRS document serves as a product validation check.
- The SRS also serves as the parent document for testing and validation strategies that will be applied to the requirements for verification.

## 2.11 BENEFITS OF INVOLVING TECHNICAL WRITERS IN SRS

Having technical writers involved throughout the entire SRS development process can offer several benefits:

- Technical writers are skilled information gatherers, ideal for eliciting and articulating customer requirements. The presence of a technical writer on the requirements gathering team helps balance the type and amount of information extracted from customers, which can help improve the SRS.
- Technical writers can better access and plan documentation projects and better meet customer document needs. Working on SRSs provides

NOTES

technical writers with an opportunity for learning about customer needs firsthand—early in the product development process.

- Technical writers know how to determine the questions that are of concern to the user or customer regarding ease of use and usability. Technical writers can then take that knowledge and apply it not only to the specification and documentation development, but also to user interface development.
- Technical writers involved early and often in the process, can become an information resource throughout the process, rather than an information gatherer at the end of the process.

## 2.12 SRS DOCUMENT TEMPLATE

The easiest way to writing an SRS document is to use SRS template. Most of the software development organizations develop their own SRS template, which can serve the purpose for all the software projects undertaken for development.

SRS Document Template

One such SRS Document Template Structure is described in given Table.

**Document Title:**

Author(s)

Affiliation

Address

Date

Document version control information.

1. **Introduction**
   - **Purpose of this Document.** Describe the purpose of document, and the intended audience.
   - **Scope of this Document.** Describe the scope of this requirements definition effort. Introduces the requirements elicitation team, including users, customers, system engineers, and developers.

     This section also details any constraint that were placed upon the requirements elicitation process, such as schedules, costs, or the software engineering environment used to develop requirements.
   - **Overview.** Provides a brief overview of the product defined as a result of the requirements elicitation process.
   - **Business Context.** Provides an overview of the business organization sponsoring the development of this product.

This overview should include the business's mission statement and its organizational objectives or goals.

2. **General Description**
   - **Product Functions.** Describes the general functionality of the product, which will be discussed in more detail below.
   - **Similar System Information.** Describes the relationship of this product with any other products. Specifies if this product is intended to be stand-alone, or else used as a component of a larger product. If the latter, this section discusses the relationship of this product to the larger product.
   - **User Characteristics.** Describes the features of the user community, including their expected expertise with software systems and the application domain.
   - **User Problem Statement.** This section describes the essential problem(s) currently confronted by the user community.
   - **User Objectives.** This section describes the set of objectives and requirements for the system from the user's perspective. It may include a "wish list" of desirable characteristics, along with more feasible solutions that are in line with the business objectives.
   - **General Constraints.** Lists general constraints placed upon the design team, including speed requirements, industry protocols, hardware platforms, and so forth.
3. **Function Requirements.** This section lists the functional requirements in ranked order. Functional requirements describe the possible effects of a software system, in other words, what the system must accomplish. Other kinds of requirements (such as interface requirements, performance requirements, or reliability requirements) describe how the system accomplishes its functional requirements.

NOTES

Each functional requirement should be specified in a format similar to the following:

1. ***Short, imperative sentence stating highest ranked functional requirement***
   - Description

     A full description of the requirements.
   - Criticality

     Describes how essential this requirement is to the overall system.
   - Technical issues

     Describes any design or implementation issues involved in satisfying this requirement.
   - Cost and Schedule

NOTES

Describes the relative or absolute costs associated with this issue.

- Risks

  Describes the circumstances under which this requirement might not able to be satisfied, and what actions can be taken to reduce the probability of this occurrence.

- Dependencies with other requirements

  Describe interactions with the other requirements.

- .....others as appropriate

2. ***<Name of second highest ranked requirements>***

   And so forth.....

3. ***Interface Requirements.*** This section describes how the software interfaces with other software products or users for input or output. Examples of such interfaces include library routines, token streams, shared memory, data streams, and so forth.

   - **User Interfaces.** Describes how this product interfaces with user.

     – **GUI.** Describes the graphical user interface if present. This section should include a set of screen dumps or mocks-ups to illustrate user interface features.

     If the system is menu-driven, a description of all menus and their components should be provided.

     – **CLI.** Describes the command line interface (or command user interface, CUI, if present). For each command, a description of all arguments and example values and invocations should be provided.

     – **API.** Describes the application programming interface, if present. For each public interface function, the name, arguments, return values, examples of invocation, and interactions with other functions should be provided.

     – **Diagnostics or ROM.** Describes how to obtain debugging information or other diagnostic data.

   - **Hardware Interfaces.** Describes interfaces to hardware devices.

   - **Communications Interfaces.** Describe network interfaces.

   - **Software Interfaces.** Describes any remaining software interfaces not included above.

4. ***Performance Requirements.*** Specifies speed and memory requirements.

NOTES

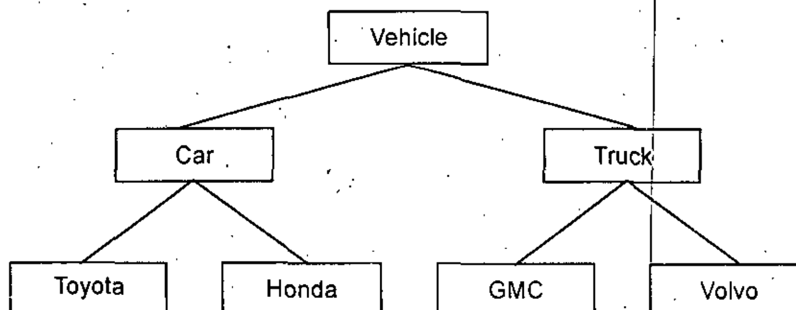
5. ***Design Constraints.*** Specifies any constraints for the design team using this document.
   - Standards Compliance
   - Hardware Limitations

   ...others as appropriate.

6. ***Other Non-Functional Attributes.*** Specifies any other particular non-functional attributes required by the system. Examples are provided below:
   - Security
   - Binary compatibility
   - Reliability
   - Maintainability
   - Portability
   - Extensibility
   - Reusability
   - Application Affinity/compatibility
   - Resource utilization.
   - Serviceability

   ...others as appropriate.

7. ***Preliminary Object Oriented Domain Analysis.*** This section presents a list of the fundamental objects that must be modelled within the system to satisfy its requirements. The purpose is to provide an alternative, "structural" view on the requirements stated above and how they might be satisfied in the system.
   - **Inheritance Relationships.** This section should contain a set of graphs that illustrate the primary inheritance hierarchy (is kind-of) for the system. *e.g.*,




   - **Class Descriptions.** This section presents a more detailed

NOTES

description of each class identified during the OO domain analysis. Each class description should conform to the following structure:

- **<class name>**
- **Abstract or Concrete.** Indicates whether this class is abstract or concrete.
- **List of superclasses.** Names all immediate superclasses.
- **List of subclasses.** Names all immediate subclasses.
- **Purpose.** States the basic purpose of the class.
- **Collaborations.** Names each class with which this class must interact in order to accomplish its purpose, & how.
- **Attributes.** Lists each attribute (state variables) associated with each instance of this class, and indicates examples of possible values (or a range).
- **Operations.** Lists each operations that can be invoked upon instances of this class. For each operation, the arguments (and their type), the return value (and its type), and any side effects of the operation should be specified.
- **Constraints.** Lists any restrictions upon the general state or behaviour of instances of this class.

**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.

**Preliminary Schedule.** This section provides an initial version of the project plan, including the major tasks to be accomplished, their independencies, and their tentative start/stop dates. The plan also includes information on hardware, software, and netware resource requirements. The project plan should be accompanied by one or more PERT or GANTT charts.

**Preliminary Budget.** This section provides an initial budget for the project, itemized by cost factor.

**Appendices:** Specifies other useful information for understanding the requirements. All SRS documents should include at least the following two appendices:

- **Definitions, Acronyms, Abbreviations.** Provides definitions of unfamiliar definitions, terms, and acronyms.
- **References.** Provides complete citations to all documents and meetings referenced or used in the preparation of this document.

## STUDENT ACTIVITY

1. Write a short note on SRS document.

2. Describe in brief the components of SRS.

## SUMMARY

- Functional requirements specify which outputs should be produced from the given inputs.
- Hardware limitations can include the type of machines to be used, operating system available on the system, languages supported, and limits on primary and secondary storage.
- Security requirements are particularly significant in defense system and many database systems.
- The interface requirement should specify the interface with other software the system will use or that will use the system.
- Requirements are defined as descriptions and specifications of a system. It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- The requirement document should be Correct; Complete; Consistent; Unambiguous; Functional; Verifiable; Traceable and Easily Changed.
- Requirements Validation is the process of determining that the specification is consistent with the requirements definition *i.e.,* validations make sure that the requirements will meet the customers' needs.

NOTES

## REVIEW QUESTIONS

1. What are the types of software requerement specifications?
2. Give an example for functional and non-functional requirement for the software requirements.
3. What is the format of specification?
4. Discuss the major uses of SRS document.
5. What are the benefits of a good SRS?
6. Explain various types of errors in SRS.
7. What are the characteristics of a good SRS document?
8. Explain the major goals of SRS document.

## FURTHER READINGS


1. **Software Engineering**, Bharat Bhushan Agarwal, Sumit Prakash Tayal, Firewall Media.
2. **Software Engineering**, D. Sunder, University Science Press.

# UNIT III DESIGNING SOFTWARE SOLUTIONS

NOTES

★ STRUCTURE ★

3.0 Learning Objectives
3.1 Introduction
3.2 Definition of Software Design
3.3 Architectural Design
3.4 Low-Level Design
3.5 Structured Design Methodology
3.6 Aim of Structured Design
3.7 Relationship Between Coupling and Cohesion
3.8 Tools for Structured Design
3.9 Modules Identification and Specification Techniques
3.10 *Module Specification Method*
3.11 Object-Oriented Design Methods
3.12 Reuse-Based Design Method
3.13 Design Specification
3.14 Verification for Design
- *Summary*
- *Review Questions*
- *Further Readings*

## 3.0 LEARNING OBJECTIVES

*After studying this unit, you will be able to:*

- explain software design
- describe architectural design
- illustrate low-level design

NOTES

## 3.1 INTRODUCTION

Design is a meaningful representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design.

A set of design concepts has evolved over the years. **According to M.A. Jackson,** "The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work and getting it right," The various design concepts discussed as under provide the necessary framework for "getting it right".

## 3.2 DEFINITION OF SOFTWARE DESIGN

The definitions of software design are as diverse as design methods. Some important software design definitions are outlined below.

***According to Coad and Yourdon***

"Software Design is the practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation, including human interaction, task management, and data management details".

***According to Webster***

"In a sense, design is representation of an object being created. A design information base that describes aspects of this object, and the design process can be viewed as successive elaboration of representations, such as adding more information or even backtracking and exploring alternatives".

***According to Stevens***

"Software Design is the process of inventing and selecting programs that meet the objectives for software systems".

Input includes an understanding of the following

(*i*) Requirements

(*ii*) Environmental constraints

(*iii*) Design criteria

The output of the design effort is composed of the following.

(i) Architecture design which shows how pieces are interrelated.

(ii) Specifications for any new pieces.

(iii) Definitions for any new data".

## Design Objectives/Properties

The various desirable properties or objectives of software design are:

### 1. *Correctness*

The design of a system is correct if a system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase is to produce correct designs. However, correctness is not the sole criterion during the design phase, as there can be many correct designs. The goal of the design process is not simply to produce a design for the system. Instead, the goal is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.

### 2. *Verifiability*

Design should be correct and it should be verified for correctness. Verifiability is concerned with how easily the correctness of the design can be checked. Various verification techniques should be easily applied to design.

### 3. *Completeness*

Completeness requires that all the different components of the design should be verified *i.e.*, all the relevant data structure; modules, external interfaces and module interconnections are specified.

### 4. *Traceability*

Traceability is an important property that can get design verification. It requires that the entire design element must be traceable to the requirements.

### 5. *Efficiency*

Efficiency of any system is concerned with the proper use of scarce

NOTES

resources by the system. The need for efficiency arises due to cost considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. Two of the important such resources are processor time and memory. An efficient system consumes less processor time and memory.

### 6. Simplicity

Simplicity is perhaps the most important quality criteria for software systems. Maintenance of software system is usually quite expensive. The design of the system is one of the most important factors affecting the maintainability of the system.

## Design Principles

The three design principles are as follows:

(*a*) Problem partitioning.

(*b*) Abstraction.

(*c*) Top-down and Bottom-up design.

### 1. Problem Partitioning

When solving a small problem, the entire problem can be tackled at once. For solving larger problems, the basic principle is the time-tested principle of "divide and conquer". This principle states that divide into smaller pieces, so that each piece can be conquered separately.

For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces:

However, the different pieces cannot be entirely independent of each other as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. It is at this point that no further partitioning needs to be done. The designer has to make the judgement about when to stop partitioning.

Problem partitioning can be divided into two categories:

(*i*) Horizontal partitioning

(*ii*) Vertical partitioning

### *(i) Horizontal Partitioning*

Horizontal partitioning defines separate branches of modular hierarchy for each major program function. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called processing) and output. Partitioning their architecture horizontally provides a number of distinct benefits:

- Software that is easier to test.
- Software that is easier to maintain.
- Software that is easier to extend.
- Propagation of fewer side effects.

On the negative part, horizontal partitioning often causes more data to be passed across modules interfaces and can complicate the overall control of program flow.

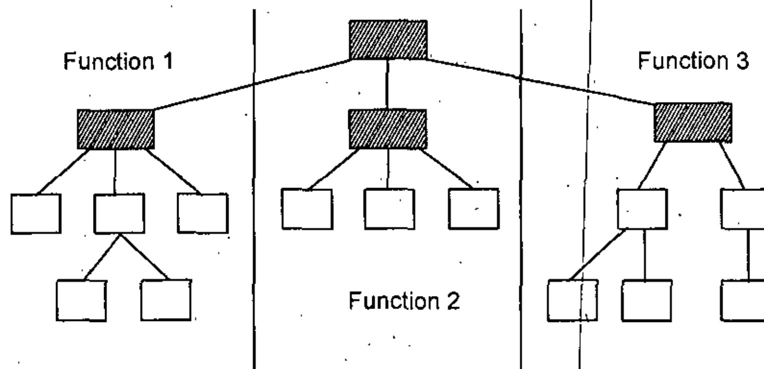



**Fig. 1** *Horizontal Partitioning*

### *(ii) Vertical Partitioning*

Vertical partitioning, often called factoring, suggests that control and work should be distributed from top-down in the programme structure. Top level modules should perform control function and do actual processing work. Modules that reside low in the structure should be the workers, performing all input, compilation and output tasks.

NOTES

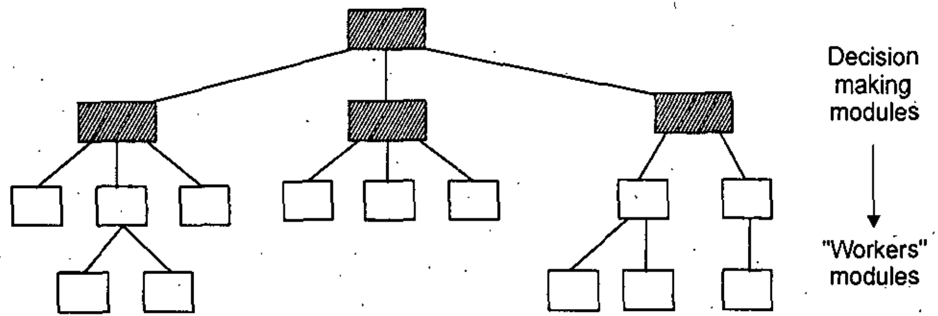



**Fig. 2** *Vertical Partitioning*

## 2. Abstraction

An abstraction of a component describes the external behaviour of that component without bothering with the internal details that produce the behaviour.

Abstraction is an indispensable part of the design process and it is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of a system. However, these components are not isolated from each other, but interacts with other components. In order to allow the designer to concentrate on one component at a time, abstraction of other component is used.

Abstraction is used for existing components as well as component that are being designed. Abstraction of existing components plays an important role in the maintenance phase.

During the design process, abstractions are used in the reverse manner than in the process of understanding a system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

There are two common abstraction mechanisms for software systems: Functional abstraction and data abstraction. In functional abstraction, a module is specified by the function it performs. For example, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. That is when the problem is being partitioned; the overall transformation function for the system is partitioned into smaller functions that comprise the system function.

The second unit for abstraction is data abstraction. There are certain operations required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this

view. Data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects. From outside an object, the internals of the object are hidden; only the operations on the object are visible.

### 3. *Top-down and Bottom-up Design*

A system consists of components, which have components of their own; indeed a system is a hierarchy of components. The highest-level components correspond to the total system.

To design such hierarchies there are two possible approaches: top-down and bottom-up. The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast, a bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved. Top-down design methods often result in some form of stepwise refinement. Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. The top-down approach has been promulgated by many researchers and has been found to be extremely useful for design. Most design methodologies are based on the top-down approach.

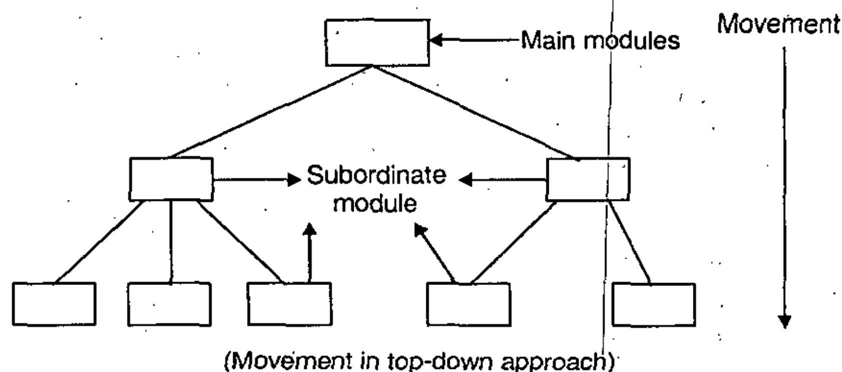



**Fig. 3** *Top-down Approach*

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components. Bottom-up methods work with layers of abstraction. Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and a still higher

layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

NOTES

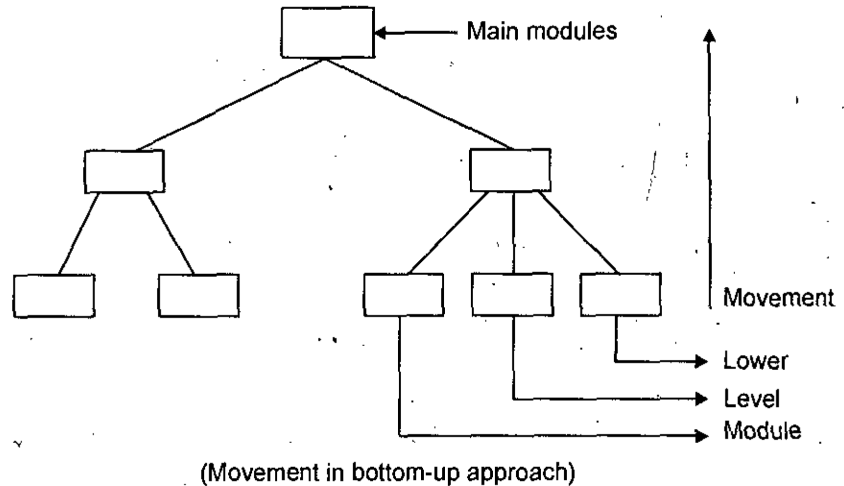



**Fig. 4** *Bottom-up Approach*

A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch. However, if a system is to be built from an existing system, a bottom-up approach is more suitable, as it starts from some existing components. So, for example, if an iterative enhancement type of process is being followed, in later iterations, the bottom-up approach could be more suitable (in the first iteration a top-down approach can be used).

## 3.3 ARCHITECTURAL DESIGN

Large systems are always decomposed into subsystems that provide some related set of services. The initial design process of identifying these sub-systems and establishing a framework for subsystem control and communication is called ***architectural design.***

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Architectural design methods have a look into various alternates' architectural style of designing a system. These are:

1. Data centric architecture
2. Data flow architecture
3. Object oriented architecture
4. Layered architecture

***Data centric architecture*** approach involves the use of a central database operations of inserting, updating it in the form of a table. ***Data flow architecture*** is central round the pipe and filter mechanism. This architecture is applied when input data takes the form of output after passing through various phases of transformations. These transformations can be via manipulations or various computations done on the data. In ***object oriented architecture*** the software design moves around the clauses and object of the system. The class encapsulates the data and methods. At least ***layered architecture*** defines a number of layers and each layer performs tasks. The outermost layer handles the functionality of the user interface and the innermost layer mainly handles interaction with the hardware.

## Objectives of Architectural Design

The objective of architectural design is to develop a model of software architecture, which gives a overall organization of program module in the software product. Software architecture encompasses two aspects of structure of the data and hierarchical structure of the software components. Let us see how a single problem can be translated to a collection of solution domains (refer to Fig. 5)

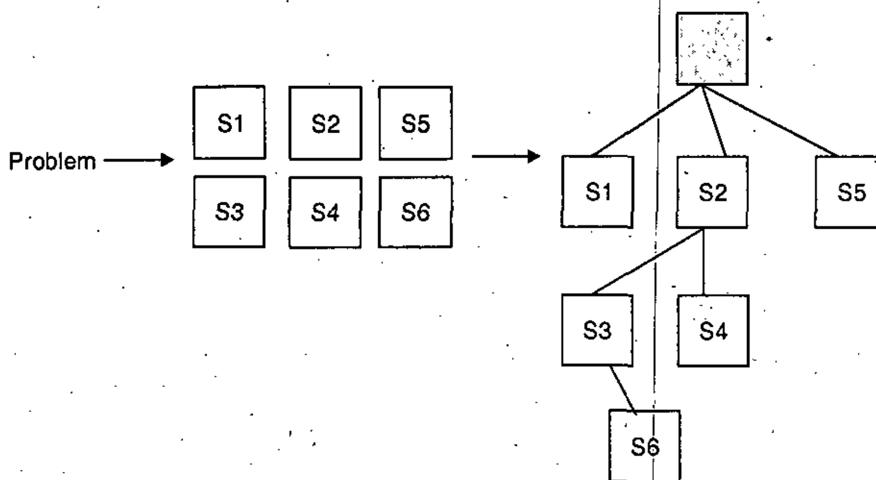



**Fig. 5** *Problems, Solutions and Architecture*

Architectural design defines organization of program components. It does not provide the details of each components and its implementation. Figure 6 depicts the architecture of a Financial Accounting System.

The objective of architectural design is also to control relationship between modules. One module may control another module or may be controlled by another module. These characteristics are defined by the fan-in and fan-out of a particular module. The organization of module can be represented through a tree like structure.

NOTES

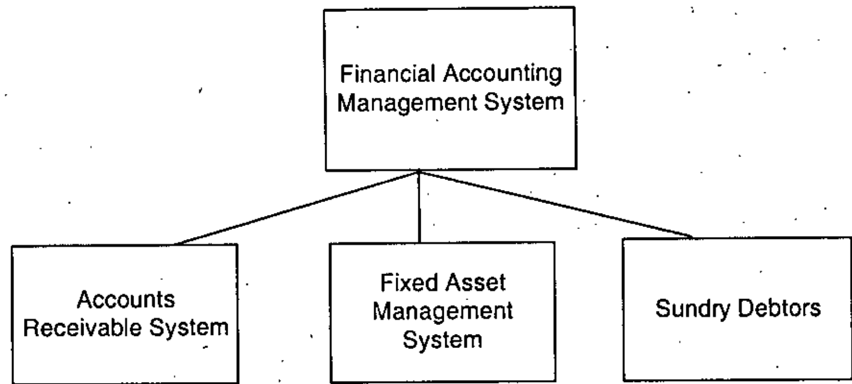



**Fig. 6** *Architecture of a Financial Accounting System*

The number of level of component in the structure is called depth and the number component across the horizontal section is called *width*. The number of components, which controls a said component, is called *fan-in i.e.,* the number of incoming edges to a component. The number of components that are controlled by the module is called *fan-out i.e.,* the number of outgoing edges.

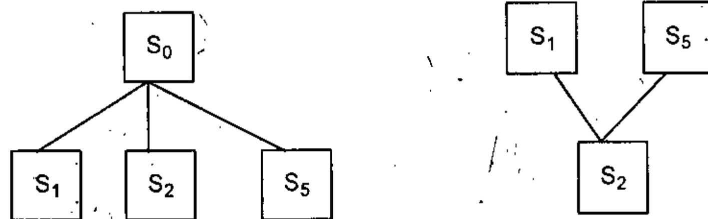



**Fig. 7** *Fan-in and Fan-out*

$S_0$ controls three components hence the fan-out is 3. $S_2$ is controlled by two components, namely, $S_1$ and $S_2$; hence the fan-in is 2 (refer to Fig. 7).

## 3.4 LOW-LEVEL DESIGN

### Modularization

A system is considered modular if it consists of discrete components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

OR

"A system is modular if it is composed of well defined, conceptually simple and independent units interacting through well defined interfaces".

There are many definitions of the term "module". They range from "a module is a FORTRAN subroutine to "a module is an ADA package" to "a module is a work assignment" for an individual programmer". All of

these definitions are correct, in the sense that modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction, and each control abstraction handles a local aspect of the problem being solved. Modular system consists of well-defined, manageable units with well-defined interfaces among the units. Desirable properties of a modular system include:

1. Each function in each abstraction has a single, well-defined purpose.
2. Each function manipulates no more than one major data structure.
3. Functions share global data selectively. It is easy to identify all routines that share a major data structure.
4. Functions that manipulate instances of abstract data types are encapsulated with the data structure being manipulated.

Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

OR

"Modularity is probability of the single most important characteristics of a well designed software system".

Modules may be created during program modularizations are:

- ***Process support modules:*** In it all the functions and data items that are required to support a particular business process are grouped together.
- ***Data abstraction:*** These are abstract types that are created by associating data with processing components.
- ***Functional modules:*** In it all the function that carries out similar or closely related tasks is grouped together.
- ***Hardware modules:*** In it all the functions, which controls on particular hardware are grouped together.

## *Classification of Modules*

A module can be classified into three types depending on activatic mechanism.

1. An incremental module is activated by an interruption and can be interrupted by another interrupt during the execution prior to completion.
2. A sequential module is a module that is referenced by another module and without interruption of any external software.
3. Parallel modules are executed in parallel with another modules.

The main purpose of modularity is that it allows the principle of separation

NOTES

of concerns to be applied in two phases: when dealing with the details of each module in isolation (and ignoring details of other modules) and when dealing with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system. If the two phases are temporally executed in the order mentioned, then we say that the system is designed ***bottom-up***; the converse denotes ***top-down*** design.

### *Advantages of Modular Systems*

1. Modular systems are easier to understand and explain because their parts make are functionally independent.
2. Modular systems are easier to document because each part can be documented as an independent unit.
3. Programming individual modules is easier because the programmer can focus on just one small, simple problem rather than a large complex problem.
4. Testing and debugging individual modules is easier because they can be dealt within isolation from the rest of the program.
5. Bugs are easier to isolate and understand, and they can be fixed without fear of introducing problems outside the module.
6. Well-composed modules are more reusable because they are more likely to comprise part of a solution to many problems. Also a good module should be easy to extract from one program and insert into another. Example.

Modularity is an important property of most engineering processes and products. For example, in the automobile industry, the construction of cars proceeds by assembling building blocks that are designed and built separately. Furthermore, parts are often reused from model to model, perhaps after minor changes. Most industrial processes are essentially modular, made out of work packages that are combined in simple ways (sequentially or overlapping) to achieve the desired result.

## 3.5 STRUCTURED DESIGN METHODOLOGY

Structured Design Methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system. The software is viewed as a transformation function that transforms the given inputs into the desired outputs, and the central problem of designing software systems is considered to be properly

designing this transformation function. Due to this view of software, the structured design methodology is primarily function oriented and relies heavily on functional abstraction and functional decomposition.

The concept of the structure of a program lies at the heart of the structured design method.

During design, Structured Design Methodology aims to control and influence the structure of the final program. The aim is to design a system so that programs implementing the design would have a hierarchical structure, with functionally cohesive modules and as few interconnections between modules as possible.

Solutions

NOTES

## 3.6 AIM OF STRUCTURED DESIGN

The aim of structured design is to specify modules that can be developed, written, tested, modified, and reused independently, and combined to form the final program. Good modularization is thus a primary goal.

Module quality depends mainly on two factors:

- Coupling, and
- Cohesion

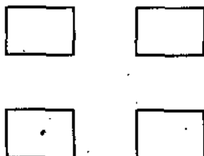
### Coupling

"Coupling is a measure of interconnection among modules in a software structure."

***The coupling between two modules indicates the degree of interdependence 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 Modules:
No Dependencies

NOTES

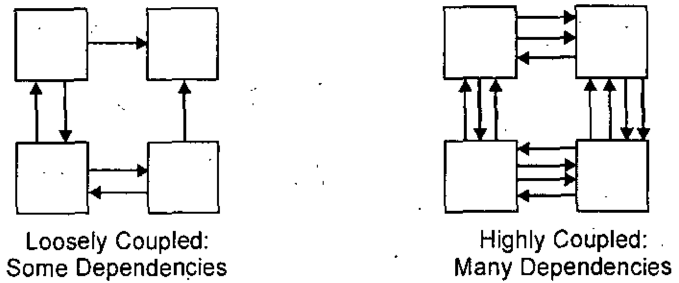



**Fig. 8** *Coupling*

**Uncoupled:** When the different modules have no interconnection among them then it is called uncoupled module.

### *Factors Affecting Coupling between Modules*

The various factors which affect the coupling between the modules are depicted in the tabular form below:

**Table 1 Factors Affecting Coupling**

| | ***Interface Complexity*** | ***Type of Connection*** | ***Type of Communication*** |
|---|---|---|---|
| Low | Simple Obvious | To module by name | Data |
| High | Complicated Obscure | To internal elements | Control Hybrid |

### *Types of Coupling*

Different types of coupling are content, common, external, control, stamp and data. The strength of coupling from lowest coupling (best) to highest coupling (worst) is given in Fig. 9.

| Data coupling | Best |
|---|---|
| Stamp coupling | ↑ |
| Control coupling | |
| External coupling | |
| Common coupling | |
| Content coupling | Worst |

**Fig. 9** *The Types of Module Coupling*

### *1. Data Coupling*

Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, for example,

an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

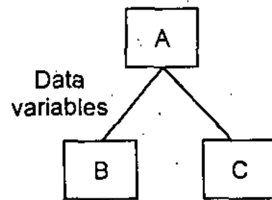



**Fig. 10** *Data Coupling*

When non-global variable is passed to a module, modules are called data coupled. It is the lowest form of coupling. For example, passing int variable from one module in C and receiving the variable by value (*i.e.*, call by value).

### *2. Stamp Coupling*

Two modules are stamp coupled, if they communicate using a composite data item such as a record, structure, object etc. When module passes non-global data structure or entire 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.

### *3. Control Coupling*

Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module and tested in another module.

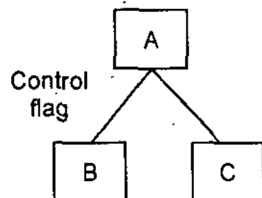



**Fig. 11** *Control Coupling*

The sending module must know a great deal about the inner working of the receiving module. A variable that controls decisions in subordinate module C is set in super ordinate module A and then passed to C.

### *4. External Coupling*

It occurs when modules are tried to an environment external to software. External coupling is essential but should be limited to a small number of modules with structure.

### 5. Common Coupling

NOTES

Two modules are common coupled, 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 does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them.

### 6. Content Coupling

Content coupling exists between two modules, if their code is shared, for example, a branch from one module into another module. It means when one module directly refer to the inner workings of another module. Modules are highly interdependent to each other. It is the highest form of coupling. It is least desirable coupling as one component actually modifies another and thereby the modified component is completely dependent on the modifying one.

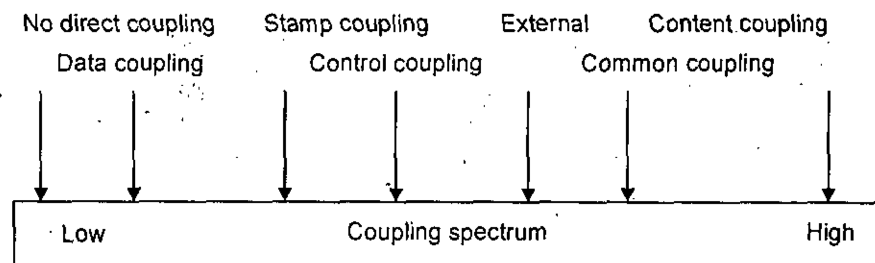



**Fig. 12** *Coupling*

High coupling among modules not only makes a design difficult to understand 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.

## Cohesion

"Cohesion is a natural extension of information hiding concept."

Cohesion is a measure of the relative functional strength of a module. The 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.

A strongly cohesive module implements functionality that is related to one feature of the solution and requires little or no interaction with

NOTES

other modules. This is shown in Fig. 13. Cohesion may be viewed as glue that keeps the module together. It is a measure of the mutual officity of the components of a module.

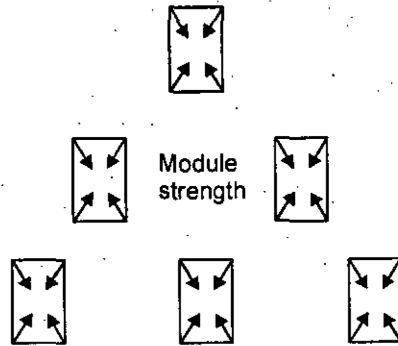



**Fig. 13** *Cohesion=Strength of Relation Within Modules*

Thus, we want to maximize the interaction within a module. Hence, 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 decreasing order to desirability which are as follows:

| | |
|---|---|
| Functional Cohesion | Best (high) |
| Sequential Cohesion | ↑ |
| Communicational Cohesion | |
| Procedural Cohesion | |
| Temporal Cohesion | |
| Logical Cohesion | |
| Coincidental Cohesion | Worst (low) |

**Fig. 14** *The Types of Module Cohesion*

## *1. Functional Cohesion*

Functional cohesion is said to exist if different elements of a module cooperate to achieve a single function, *e.g.*, managing an employee's payroll. When a module displays functional cohesion, and if we are asked to describe what the module does we can describe it using a single sentence.

NOTES

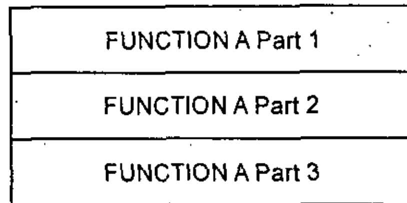



**Fig. 15** *Functional Cohesion: Sequential with Complete, Related Functions*

## 2. Sequential Cohesion

A module is said to possess sequential cohesion, if the elements of a module form the parts of a sequence, where the output from one element of the sequence is input to the next.

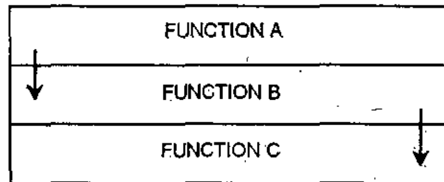



**Fig. 16** *Sequential Cohesion: Output of one Part is Input to Next*

## 3. Communicational Cohesion

A module is said to have communicational cohesion, if all the functions of the module refer to or update the same 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 the set of functions defined on an array or a stack.

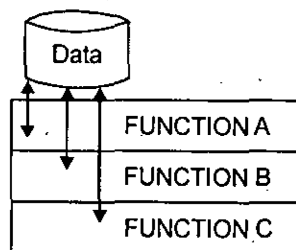



**Fig. 17** *Communicational Cohesion: Access Same Data*

## 4. Procedural Cohesion

A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps has to be carried out for achieving an objective, for example, the algorithm for decoding a message.

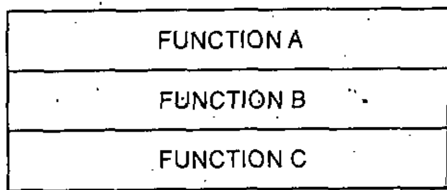



**Fig. 18** *Procedural Cohesion Related by Order of Function*

### *5. Temporal Cohesion*

When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc., exhibit temporal cohesion.

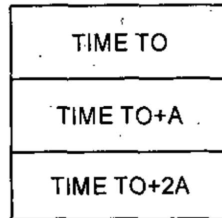



**Fig. 19** *Temporal Cohesion Related by Time*

### *6. Logical Cohesion*

A module is said to be logically cohesive, if all elements of the module perform 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 into a single module.

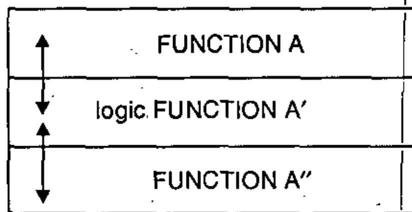



**Fig. 20** *Logical Cohesion Similar Functions*

### *7. Coincidental Cohesion*

A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely. In this case, 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.

NOTES

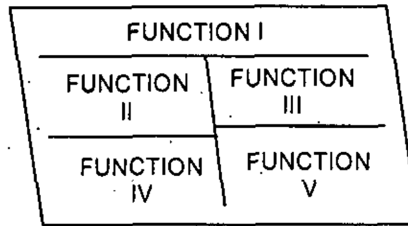



**Fig. 21** *Coincidental Cohesion Parts Unrelated*

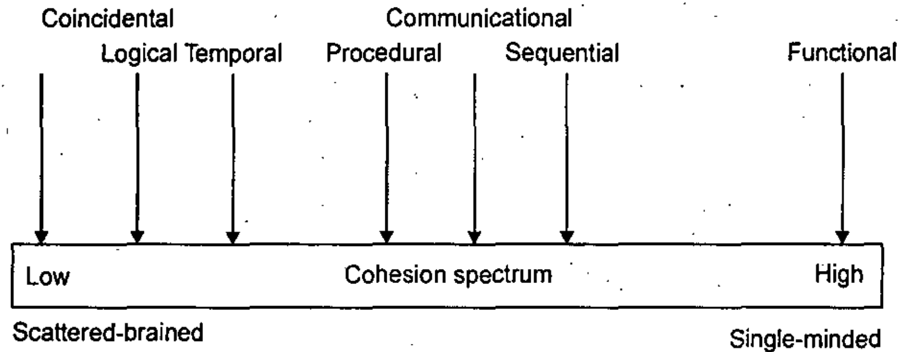



**Fig. 22** *Cohesion*

## 3.7 RELATIONSHIP BETWEEN COUPLING AND COHESION

A software engineer must design the modules with goal of high cohesion and low coupling.

A good example of a system that has high cohesion and low coupling is the 'plug and play' feature of the computer system. Various slots in the mother board of the system simply facilitate to add or remove the various services/functionalities without affecting the entire system. This is because the add on components provide the services in highly cohesive manner. Fig. 23 provides a graphical review of cohesion and coupling.

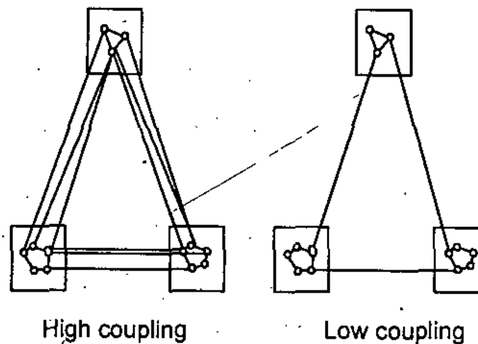



**Fig. 23** *View of Cohesion and Coupling*

Module design with high cohesion and low coupling characterizes a module as black box when the entire structure of the system is described. Each module can be dealt separately when the module functionality is described.

## 3.8 TOOLS FOR STRUCTURED DESIGN

NOTES

The following are the key tools used for the structured design:

- Structure charts
- Modules identification and specification techniques.

These key tools are discussed below.

### Structure chart

The Structure chart is one of the most commonly used methods for system design. Structure charts are used during architectural design to document hierarchical structure, parameters, and interconnections in a system.

It partitions a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design. Inputs are given to black box and appropriate outputs are generated by the black box. This concept reduces the complexity because details are hidden from those who have no need or desire to know. Thus systems are easy to construct and easy to maintain. Here, black boxes are arranged in hierarchical format as shown in Fig. 24 (*a*) & (*b*)

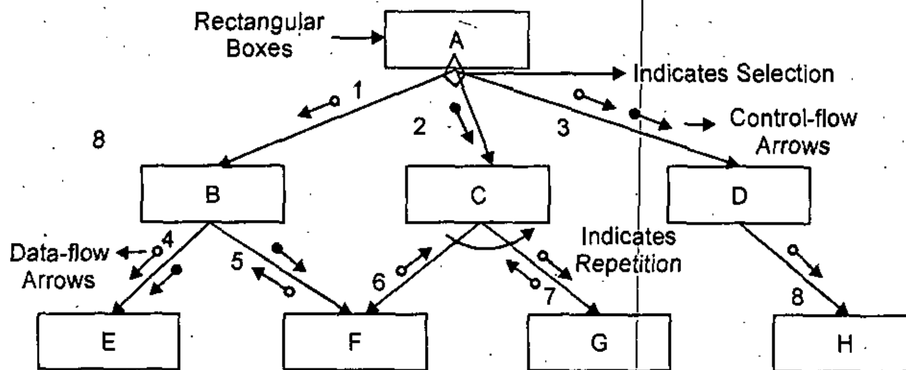



**Fig. 24 (a)** *Hierarchical Format of a Structure Chart*

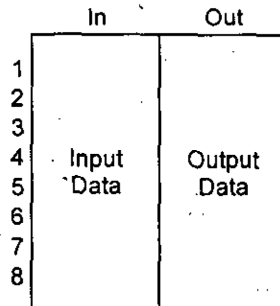



**Fig. 24 (*b*)** *Format of a Structure Chart*

NOTES

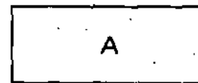
Modules at the top-level call the modules at the lower level. The connections between modules are represented by lines between the rectangular boxes. The components are generally read from top to bottom, left to right. Modules are numbered in hierarchical numbering scheme. In any structure chart there is one and only one module at the top called the root.

## Basic Building Blocks of Structure Chart

The basic building blocks of structure chart are the following:

### *1. Rectangular Boxes*

A rectangular box represents a module. Usually a rectangular box is annotated with the name of the module it represents.

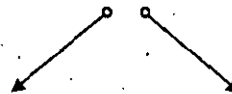



### *2. Arrows*

An arrow connecting two modules implies that during program execution, control is passed from one module to the other in the direction of the connecting arrow.

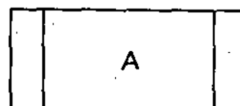
### *3. Data Flow Arrows*

Data flow arrow represents that the named data passes from one module to the other in the direction of the arrow.



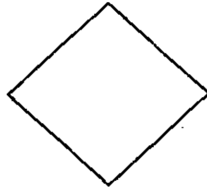
### *4. Library Modules*

Library comprises the frequently called modules and is usually represented by a rectangle with double edges. Usually when a module is invoked by many other modules, it is made into a library module.



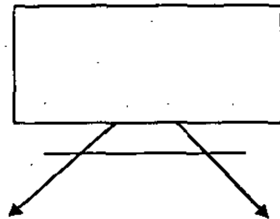
### 5. Selection

The diamond symbol represents that one module out of several modules connected with the diamond symbol are invoked depending on the condition satisfied, which is written in the diamond symbol.



### 6. Repetitions

A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.



**Example.** A software system called RMS calculating software reads three integral numbers from the user in the range between –1000 and +1000 and determines the root mean square (rms) of the three input numbers and then displays it.

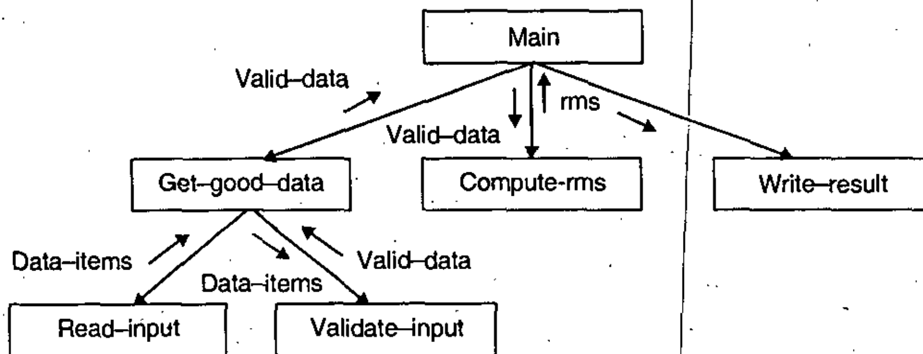



**Fig. 25** *Structure Chart for Example*

## *Difference between Flowchart and Structure Chart*

A structure chart differs from a flow chart in following ways:

1. It is usually difficult to identify different modules of the software from its flow chart representation.
2. Data interchange among different modules is not represented in a flow chart.

3. Sequential ordering of tasks inherent in a flowchart is suppressed in a structure chart.

4. A structure chart has no decision boxes.

NOTES

Unlike flow charts, structure charts show how different modules within program interact and data that is passed between them.

## 3.9 MODULES IDENTIFICATION AND SPECIFICATION TECHNIQUES

When developing the implementation model, and specifically the structure charts, arguably the largest task is the identification and specification of the modules within the system. As with most methodologies and techniques there are a few guidelines to be used when developing individual modules as well as when considering the relationships between modules.

## 3.10 MODULE SPECIFICATION METHOD

There are several methods that can be used to specify a module. Two possible methods are:

- Interface/Functional
- Pseudocode

### Interface/Functional

Interface/functional specification provides a good balance of specification detail, and is in line with the "black box" spirit of SASD. (System Analysis and System Design).

The inferface, or rather, the input and output of the module, provides the detail of what the module needs and produces, while the functional specification provides good documentation as to what, exactly, the module is supposed to do.

### Pseudocode

"Pseudo" means imitation or false and "Code" refers to the instructions written in a programming language. Pseudocode notation can be used

in both the preliminary and detailed design phases. Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by keywords such as If-Then-Else, While-Do, and End. Keywords and indentation describe the flow of control, while the English phrases describe processing actions. Pseudocode is also known as program design language or structured English. A program design language should have the following characteristics:



1. A fixed syntax of keywords that provide for all structured constructs, data declarations and modularity characteristics.
2. A free syntax of natural language that describes processing feature.
3. A data declaration facility.
4. Subprogram definition and calling techniques.

### *Advantages of Pseudocodes*

The various advantages of pseudo-codes are as follows:

1. Converting a pseudo-code to a programming language is much easier as compared to converting a flowchart or decision table.
2. As compared to a flow chart, it is easier to modify the pseudo-code of program logic whenever program modifications are necessary.
3. Writing of pseudocode involves much less time and effort than equivalent flow chart.
4. Pseudocode is easier to write- than writing a program in a programming language because pseudocode method has only a few rules to follow.

### *Disadvantages of Pseudocodes*

The various disadvantages of pseudocodes are as follows:

1. In case of pseudocode, a graphic representation of program logic is not available as in flow charts.
2. There are no standard rules to follow in using pseudocode. Different programmers use their own style of writing pseudocode and hence communication problems occur due to lack of standardization.
3. For a beginner, it is more difficult to follow the logic or write the pseudocode, as compared to flowcharting.

**Example.** Pseudocode consists of English-like statements describing an algorithm. It is written using simple phrases and avoids cryptic symbols.

NOTES

It is independent of high-level languages and is a very goods means of expressing an algorithm. It is written in structured manner and indentation is used to increase clarity. As an example, the use of pseudocode for detailed design specification is illustrated in fig 26.

```
INITIALIZE tables and counters; OPEN files
READ the first text record
WHILE there are more text records DO
WHILE there are more words in the text record DO
EXTRACT the next word
SEARCH word_table for the extracted word
IF the extracted word is found THEN
        INCREMENT the extracted word's occurrence count
ELSE
        INSERT the extracted word into the word_table
ENDIF
INCREMENT the words_processed counter
ENDWHILE at the end of the text record
ENDWHILE when all text records have been processed
PRINT the word_table and the words_processed counter
CLOSE files
TERMINATE the program
```

**Fig. 26** *An Example of a Pseudocode Design Specification*

Pseudocode consists of English–like statements describing an algorithm. It is written using simple phrases and avoids cryptic symbols. It is written in structured manner and identification is used to increase clarity.

## 3.11 OBJECT–ORIENTED DESIGN METHODS

Object-oriented technology is one of the latest approaches to S/W development, and it shows much promise in solving the problems associated with building modern software systems (Shlaer 1988, Meyers 1988, Rambaugh *et.al* 1991, Rubin 1992, Agha 1990).

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.

True object-oriented design occurs then 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.

Object-oriented technology contains these three key aspects:

**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.

**Messages.** Communication mechanisms are established that provide the means by which objects work together.

**Methods.** Methods are services that objects perform to satisfy the functional requirements of the problem domain. Objects request services of other objects through messages.

**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.

## Benefits of OOD

The benefits of object-oriented development as claimed by its proponents are many:

- Objects are inherently reusable.
- The concept of objects performing services is a more natural way of thinking.
- 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 object-oriented development process is consistent from analysis, through design to coding.

## Types of OOD Methods

The following are popular object-oriented design methods:

- Booch's Object-Oriented Design
- Yourdon and Coad's Object-Oriented Design

The above methods are discussed below.

## Booch's Object-Oriented Design

NOTES

Grady Booch's Object-Oriented Design (OOD), also known as Object-Oriented Analysis and Design (OOAD), is a precursor to the Unified Modeling Language (UML). The Booch method (Booch 1994) includes 6 types of diagrams:

- Class,
- Object,
- State transition,
- Interaction,
- Module, and
- Process.

### Booch's Static Diagrams

Booch's class and object diagrams differentiate this methodology (at least in notation) from similar object oriented systems.

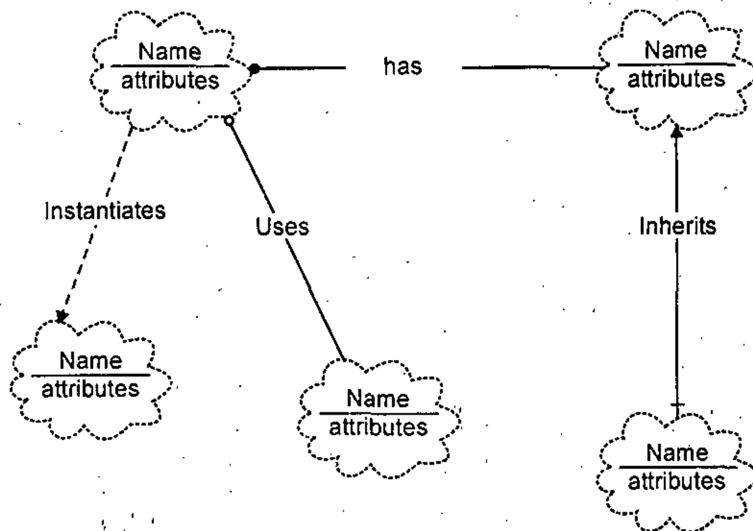



**Fig. 27** *A Booch Class Diagram*

### Booch's Class and Object Diagram Notations

**Classes.** Illustrate classes using a cloud shape with a dashed border.

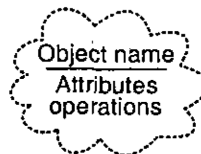



**Fig. 27** (*a*) *Class*

**Object.** Draw objects using a cloud shape with a solid border.

NOTES

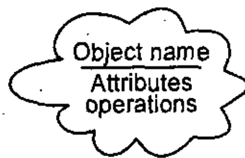



**Fig. 27** (*b*) *Object*

**Class Adornments.** Use adornments to provide additional information about a class. Adornment can be created using the basic triangle shape.

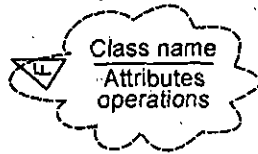



**Fig. 27** (*c*) *Class Adornment*

A letter is placed inside the triangle to represent the following:

- **A-Abstract.** An abstract class cannot be instantiated because it represents a wide variety of object classes and does not represent any one of them fully. For example, mammal could be thought of as an abstract class.
- **F-Friend.** A friend class allows access to the non-public functions of other classes.
- **S-Static.** A static class provides data.
- **V-Virtual.** A virtual class is a shared base class, the most generalized class in a system.

**Metaclass.** A metaclass is a class whose instances are also classes.

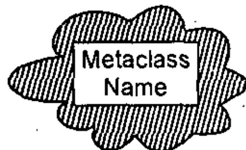



**Fig. 27** (*d*) *Metaclass*

**Class Categories.** A class category represents a cluster of similar classes. Illustrate a class category by drawing a rectangle with two compartments.

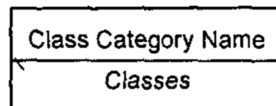



**Fig. 27** (*e*) *Class Category*

**Class Templates.** Draw a template using the standard class symbol attached to a box with a dashed outline. List template parameters or formal arguments in this box. When you draw a class created from a template, replace the dashed border with a solid one.

NOTES

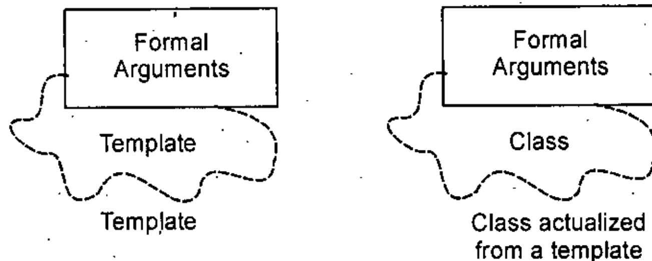



**Fig. 27 (f)** *Class Template*

**Class Utilities.** Class utilities describe a group of non-member functions or subprograms. Illustrate a class utility with a shadowed cloud.

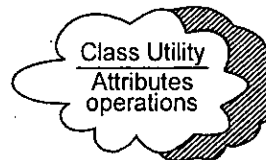



**Fig. 27 (g)** *Class Utility*

**Class Visibility.** Visibility markers signify who can access the information contained within a class. Public visibility allows an attribute or an operation to be viewed by any other class. Private visibility means that the attribute or the operation is only accessible by the class itself and its friends. Protected Visibility makes an attribute or operation visible only to friend classes and classes that inherit it. Implementation restricts the accessibility of an attribute to the class only (not even friends have access to these attributes).

Place visibility markers next to the attribute they refer to.

| private

|| protected

||| implementation

**Fig. 9.20(h)** *Class Visibility Markers*

**Object Visibility.** Draw a visibility marker on a link to signify the relationship between the connected objects. These markers can be:

- **G – Global**
- **P – Parameter**
- **F – Field**
- **L – Local.**

**Relationships.** Relationships between objects are indicated using lines and arrows. The meaning and the relationships are outlined in Table 2.

NOTES

**Table 2. Relationship Symbols**

| *Meaning* | *Relationship* |
|---|---|
| Aggregation (has) | ●——— Label ——— |
| Aggregation by value | ●——— Label ———■ |
| Aggregation by reference | ●——— Label ———□ |
| Uses | ○——— Label ——— |
| Instantiates–compatible type | - - - - Label - - - -▶ |
| Instantiates–New type | +········ Label ········▶ |
| Inherits–Compatible type | ——— Label ———▶ |
| Inherits–New type | +——— Label ———▶ |
| Metaclass | ——— Label ———▶ |
| Undefined | - - - - Label - - - - |

## Booch's Dynamic Diagrams

State transition and interaction diagrams are used to illustrate the dynamic nature of an application (Booch 1991). Below is a table that lists what each of the dynamic Booch diagrams corresponds to in UML.

| *Booch (OOD)* | *Unified Modeling Language (UML)* |
|---|---|
| State transition diagram | State chart diagram |
| Interaction diagram | Sequence Diagram |

## Booch's Dynamic Diagram Notations

States represent situations during the life of an object. A Booch state symbol is drawn using a rectangle with rounded corners and two compartments. The oval-shaped H symbol is used to indicate the most recently visited state. It is illustrated in Fig. 28.

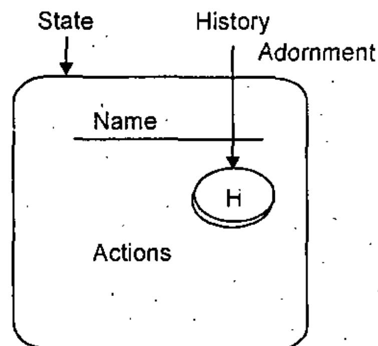



**Fig. 28** *Booch's Dynamic Diagram*

## Yourdon and Coad's Object-Oriented Design

NOTES

Yourdon and Coad's design method (Coad 1991) is an object-oriented design method. There are mainly five steps for developing Yourdon and Coad diagrams.

- Find classes and objects
- Identify the structures
- Define subjects
- Define attributes
- Define services

The notations used by Yourdon and Coad are described below in Fig. 29 (*a*) to 29 (*d*).

### Class and Object

Objects and classes are abstractions of entities with exclusive services and attributes.

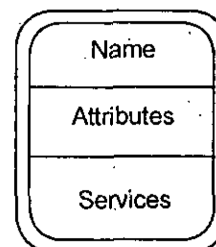



**Fig. 29 (*a*)**

### Whole-part Relationships

Whole-part relationship refer to objects that contain one or more objects. There are several types of whole-part relationships including: assembly-parts (airplane-wings), container-contents (cabinet-files), and collection-members (organization-members).

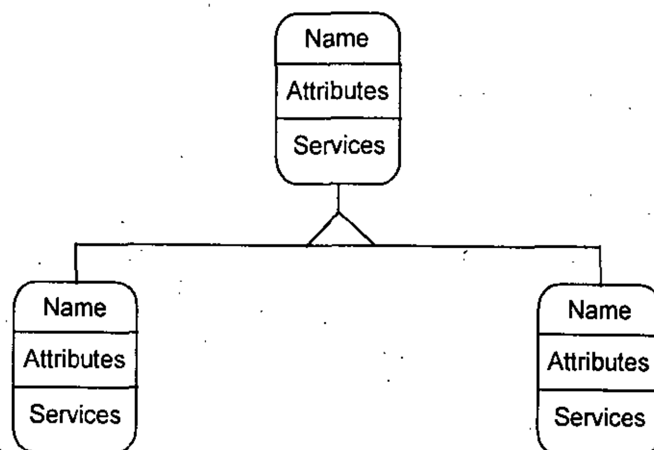



**Fig. 29 (*b*)**

### Generalization-Specialization (Gen-spec) Relationships.

Generalization-specialization relationships refer to classes that inherit attributes and services from other classes. One class can inherit from

NOTES

multiple superclasses.

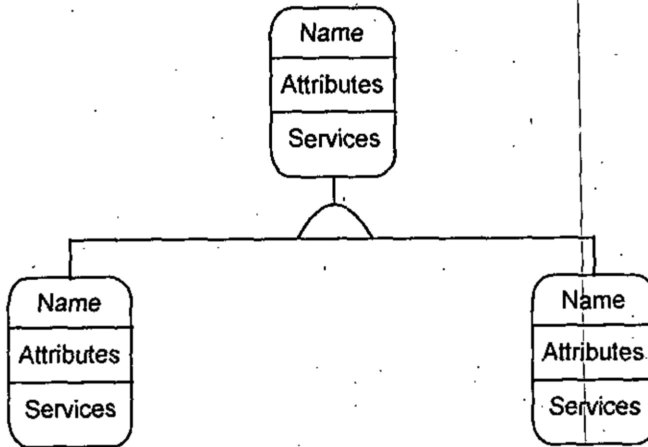



**Fig. 29 (*c*)**

## Connections

Connections illustrate the dependency of one object on the services or processing of another object.

Instance Connection

Message Connection

**Fig. 29 (*d*)**

Yourdon and Coad diagram is illustrated in Fig. 30.

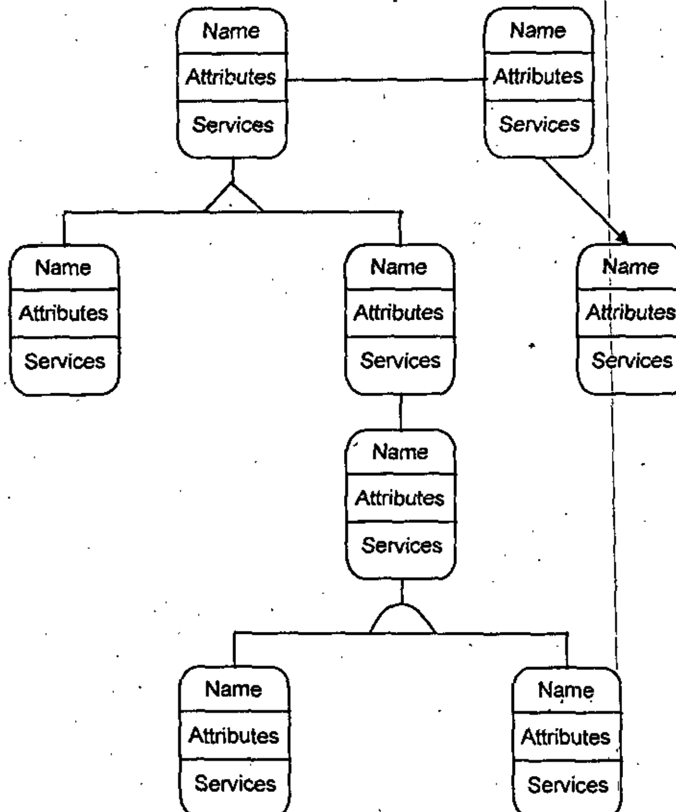



**Fig. 30** *Yourdon and Coad Class Diagram*

## 3.12 REUSE-BASED DESIGN METHOD

NOTES

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 as shown in Fig. 31.

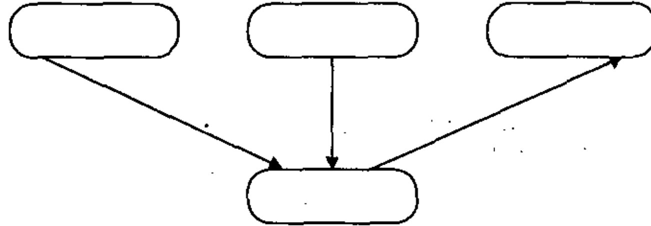


**Fig. 31** *Design–Reusable structure*

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.

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.

## 3.13 DESIGN SPECIFICATION

The Design Specification addresses different aspects of the design model and is completed as the designer refines his representation of

the software. First, the overall scope of the design effort is described, which is derived from system specification and the analysis model (Software Requirements Specification)

Then, **Data Design** is specified, which includes data structures, any external file structures, internal data structures, and a cross reference that connects data objects to specific files are all defined.

Then **Architectural Design** indicates how the program architecture has been derived from the analysis model. Structure charts are used to represent the module hierarchy

**Interface Design** indicates the design of external and internal program interfaces along with a detailed design of the human/machine interface is described. A detailed prototype of a GUI may be represented.

**Procedural Design** specifies components-separately addressable elements of software such as subroutines, functions or procedures in form of English language processing narratives. This narrative explains the procedural function of a component (module).

Design specification contains a **requirements cross-reference.** The purpose of this cross-reference is:

(*i*) To establish that all requirements are satisfied by the software design.

(*ii*) To indicate which components are critical to the implementation of specific requirements?

The final section of the Design specification contains supplementary data like algorithm descriptions, alternative procedures, tabular data, excerpts from other documents and other relevant information presented as a special note or a separate **appendix.**

Design specification format is as under.

| System objective | Human-machine interface specification and design. |
|---|---|
| Major Software requirements<br>Design constraints, limitations | External interface design.<br>Interfaces to external systems. |
| Data Design | Internal design rules. |
| Data Objects and resultant data structure | Processing narrative. |
| File and database structures | Interface description. |
| External file structure | Design language description. |
| Logical structure | Modules used. |

NOTES

| | |
|---|---|
| Access method | Data structures used |
| Global data | Comments |
| File and data cross-reference | Requirements cross-reference |
| Architectural Design | Test provisions |
| Review of data and control flow | Test guidelines |
| Derived program structure | Integration strategy |
| | Special considerations |
| | Appendices. |

## 3.14 VERIFICATION FOR DESIGN

The output of the system design phase, like the output of other phases in the development process, should be verified before proceeding with the activities of the next phase. If the design is expressed in some formal notation for which analysis tools are available; then through tools it can be checked for internal consistency (*e.g.*, those modules used by another are defined, the interface of a module is consistent with the way others use it, data usage is consistent with declaration, etc.) If the design is not specified in a formal, executable language, it cannot be processed through tools, and other means for verification have to be used.

There are ***two fundamental approaches*** to verification. The ***first*** consists of ***experimenting with the behaviour*** of a product to see whether the product performs as expected (*i.e.*, testing the product.) The other consists of ***analyzing the product-or any design documentation*** related to it-to deduce its correct operation as a logical consequence of the design decisions. The two categories of verification techniques are also classified as ***dynamic or static***, since the former requires-by definition executing the system to be verified, while the latter does not. Not surprisingly, the two techniques turn out to be nicely complementary.

## STUDENT ACTIVITY

1. Define software design.

2. Write short notes on the following:
   (a) Horizontal partitioning (b) Vertical partitioning.

## SUMMARY

- Simplicity is perhaps the most important quality criteria for software systems.
- Horizontal partitioning defines separate branches of modular hierarchy for each major program function.
- Vertical partitioning, often called factoring, suggests that control and work should be distributed from top-down in the programme structure.
- An abstraction of a component describes the external behaviour of that component without bothering with the internal details that produce the behaviour.
- Architectural design represents the structure of data and program components that are required to build a computer-based system.
- The objective of architectural design is to develop a model of software architecture, which gives a overall organization of program module in the software product.
- Structured Design Methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system.
- Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another.

NOTES

- External coupling is essential but should be limited to a small number of modules with structure.
- Content coupling exists between two modules, if their code is shared, for example, a branch from one module into another module.
- The Structure chart is one of the most commonly used methods for system design.
- Relationships between objects are indicated using lines and arrows.
- Whole-part relationship refer to objects that contain one or more objects.
- Generalization-specialization relationships refer to classes that inherit attributes and services from other classes.
- Connections illustrate the dependency of one object on the services or processing of another object.
- The Design Specification addresses different aspects of the design model and is completed as the designer refines his representation of the software.

## REVIEW QUESTIONS

1. What are the various design objectives of software design?
2. What do you mean by problem partitioning?
3. Write a short note on abstraction.
4. What do you mean by architectural design? Discuss its objectives.
5. What are the desirable properties of a modular system?
6. What are the advantages of a modular systems?
7. What are the different types of coupling? Explain.
8. Explain the different types of cohesion.
9. Describe the various building blocks of structure chart.
10. What is the difference between flow chart and structure chart?
11. What do you mean by pseudo code? Discuss advantages and disadvantages of pseudo code.
12. What are the various benefits of object-oriented development?
13. Define the following:

    (*a*) Abstract class

NOTES

(*b*) Friend class

(*c*) Virtual class

(*d*) Metaclass

(*e*) Whole-port relationship

(*f*) Generalization-specialization relationship

**14.** What are the two fundamental approaches of verification for design?

## FURTHER READINGS

1. **Software Engineering**, Bharat Bhushan Agarwal, Sumit Prakash Tayal, Firewall Media.
2. **Software Engineering**, D. Sunder, University Science Press.

# UNIT IV SOFTWARE IMPLEMENTATION

NOTES

★ STRUCTURE ★

4.0 Learning Objectives
4.1 Introduction
4.2 Software Implementation Guidelines
4.3 Relation Between Design and Implementation
4.4 Coding
4.5 Coding Standards and Guidelines
4.6 Code Review
4.7 Clean Room Testing
4.8 Software Documentation
- *Summary*
- *Review Questions*
- *Further Readings*

## 4.0 LEARNING OBJECTIVES

*After studying the unit, you will be able to:*

- explain guidelines for software implementation
- describe relation between design and implementation
- describe coding and code review

## 4.1 INTRODUCTION

Implementation is the process of first ensuring that the information system is operational and then allowing users to take over its operation for use and evaluation. This involves training the users to handle the system. The analyst needs to plan for a smooth conversion from the old system to the new one. This includes converting files old formats to new ones or building new databases etc.

Once the Information System has been developed and acceptance testing is completed, the implementation process starts. Users must be trained on the use of the new system, focusing on its requirements

and its capabilities. Many organisations combine testing and training in the same stage. This works well because users can become familiar with the new IS as well as ensure that it can handle errors at the same time. Training, like testing and documentation, is ultimately a management responsibility.

## 4.2 SOFTWARE IMPLEMENTATION GUIDELINES

Software implementation should be done with proper planning, hasty decisions lead to problems and delays. The process of implementation of software should be consistent with least amount of disturbance.

Following are the some of the basic considerations that should be kept in mind for smooth implementation of software.

### Proper Equipment

The hardware and software requirements should be first re-examined with the software supplier. The equipments for general-purpose applications should be delivered several weeks before the installation of an application. Thes helps to have a basic idea about the hardware before major implementation of the applications. Sufficient time for networking should be given, if the system uses the network resources. Good quality wiring and the fastest hubs should be used to gain best performance from the system. Proper connectivity devices such as wiring, hubs and routers should be used to reduce bottleneck arising in system performance.

### Conversion

Conversion methodology ensures a professional result in line with expectations and within budgeted time period and cost. The conversion methodology clearly improves communication between the project team and management by providing a readily understandable, structured approach.

### Training

Formal training about the functioning of software should be provided to the employees for the successful use of the application software. Hardly ever, one reads a manual and implements the application in a smooth manner. Application training should be designed to teach users how to use the software.

NOTES

### Implementing Applications

Follow a definite sequence to install all the inter-related applications. For example, an application consists of general accounting, payroll and utility billing. In the application, accounting will come first because the other systems require its availability.

### Backups

Regular system backups should be taken so that the users can revert to an older copy of the data files when they commit any mistake. The backup procedure can be performed during the free hours of at the end of the day. You can take the take the backup of only the critical points in an application. Backups are stored in the removable disk or in high capacity tapes. You can also store backups in another folder of the hard disk drive of your system.

## 4.3 RELATION BETWEEN DESIGN AND IMPLEMENTATION

### System Design

Design is the most creative and challenging phase of the system development life cycle. The term design describes the final system and process by which it is developed. Different stages of design phase are shown in Fig1. This phase is very important phase of life cycle. This is a creative as well as a technical activity including the following tasks:

- Appraising the terms of reference
- Appraising the analysis of the existing system, particularly regarding problem areas
- Defining precisely the required system output
- Determining data required to produce the output
- Deciding the medium and open the files
- Devising processing methods and use of software to handle files and to produce output
- Determining methods of data capture and data input
- Designing the output forms
- Difining detailed critical procedures
- Calculating timings of processing and data movements
- Documenting all aspects of design

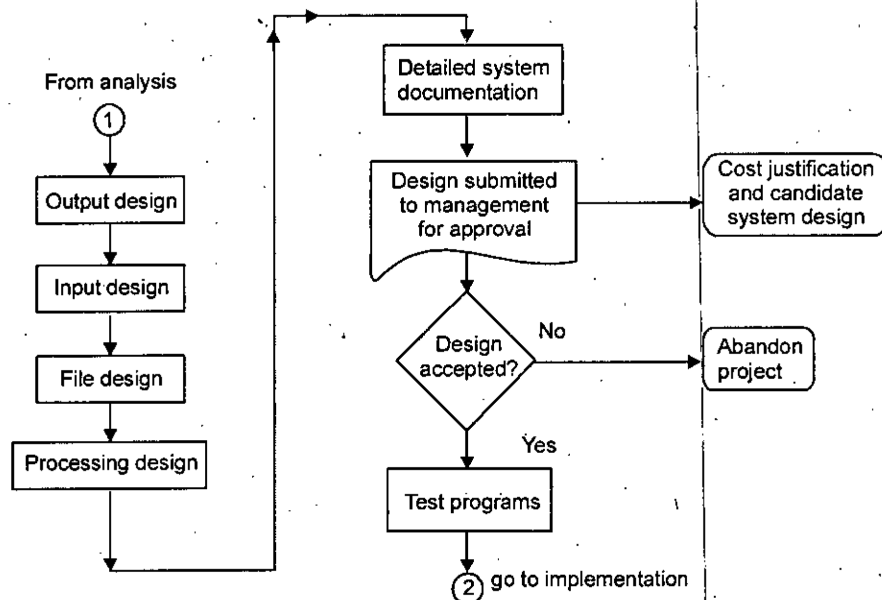



**Fig. 1.** *Cycle of Design Phase*

NOTES

## Implementation

Implementation phase is less creative than system design. It is mainly concerned with user training, site selection preparation and file conversion. Once the system has been designed, it is ready for implementation. Implemention is concerned with those tasks leading immediately to a fully operational system. It involves programmers, users and operations management, but it's planning and timing is a prime function of system's analyst. It includes the final testing of complete system to user satisfaction, and supervision of initial operation of the system. Implementation of the system includes providing security to the system, also so that some person may not misuse it.

Types of Implementation

There are three types of implementation:

- Implementation of a computer system to replace a manual system.
- Implementation of a new computer system to replace an existing one.
- Implementation of a modified application (software) to replace an existing one using the same computer.

## 4.4 CODING

Good software development organizations normally require their programmers to adhere to some well-defined and standard style of

coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously. The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It enhances code understanding.
- It encourages good programming practices.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

## 4.5 CODING STANDARDS AND GUIDELINES

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop.

The following are some representative coding standards.

**Rules for limiting the use of global:** These rules list what types of data can be declared global and what cannot.

**Contents of the headers preceding codes for different modules:** The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:

- Name of the module.
- Date on which the module was created.
- Author's name.
- Modification history.
- Synopsis of the module.
- Different functions supported, along with their input/output parameters.
- Global variables accessed/modified by the module.

**Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.

**Error return conventions and exception handling mechanisms:** The way error conditions are reported by different functions in a program

are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

NOTES

The following are some representative coding guidelines recommended by many *software development organizations.*

**Do not use a coding style that is too clever or too difficult to understand:** Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.

**Avoid obscure side effects:** The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.

**Do not use an identifier for multiple purposes:** Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, *e.g.,* three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by use of variables for multiple purposes as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes future enhancements more difficult.

**The code should be well-documented:** *As a rule of thumb,* there must be at least one comment line on the average for every three-source line.

**The length of any function should not exceed 10 source lines:** A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

**Do not use go to statements:** Use of go to statements makes a program unstructured and makes it very difficult to understand.



## 4.6 CODE REVIEW

Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated. Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module.

These two types code review techniques are code inspection and code walk through.

### Code Walk Through

Code walk through is an informal code analysis technique. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated. A few members of the development team are given the code few days before the walk through meeting to read and understand code. Each member selects some test cases and simulates execution of the code by hand (*i.e.*, trace execution through each statement and function execution). The main objectives of the walk through are to discover the algorithmic and logical errors in the code. The members note down their findings to discuss these in a walk through meeting where the coder of the module is present.

Even though a code walk through is an informal analysis technique, several guidelines have evolved over the years for making this naive but useful analysis technique more effective. Of course, these guidelines are based on personal experience, common sense, and several subjective factors. Therefore, these guidelines should be considered as examples rather than accepted as rules to be applied dogmatically. Some of these guidelines are the following.

- The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.
- In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

## Code Inspection

In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming. In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs. For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

NOTES

Following is a list of some classical programming errors which can be checked during code inspection:

1. Use of uninitialized variables.
2. Jumps into loops.
3. Nonterminating loops.
4. Incompatible assignments.
5. Array indices out of bounds.
6. Improper storage allocation and deallocation.
7. Mismatches between actual and formal parameter in procedure calls.
8. Use of incorrect logical operators or incorrect precedence among operators.
9. Improper modification of loop variables.
10. Comparison of equally of floating point variables, etc.

## 4.7 CLEAN ROOM TESTING

Clean room testing was pioneered by IBM. This type of testing relies heavily on walk throughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. The software development philosophy is based on avoiding software defects by using a rigorous inspection process. The objective of this software is zero-defect software.

NOTES

The name 'clean room' was derived from the analogy with semiconductor fabrication units. In these units (clean rooms), defects are avoided by manufacturing in ultra-clean atmosphere. In this kind of development, inspections to check the consistency of the components with their specifications has replaced unit-testing.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing.

The clean room approach to software development is based on five characteristics:

- **Formal specification:** The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.
- **Incremental development:** The software is partitioned into increments which are developed and validated separately using the clean room process. These increments are specified, with customer input, at an early stage in the process.
- **Structured programming:** Only a limited number of control and data abstraction constructs are used. The program development process is process of stepwise refinement of the specification
- **Static verification:** The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
- **Statistical testing of the system:** The integrated software increment is tested statistically to determine its reliability. These statistical tests are based on the operational profile which is developed in parallel with the system specification.

The main problem with this approach is that testing effort is increased as walk through, inspection, and verification are time-consuming.

## 4.8 SOFTWARE DOCUMENTATION

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, Software Requirements Specification (SRS) documents, design documents, test documents, installation manual, etc., are also developed as part of any software engineering process. All these documents are a vital part of good software development practice. Good documents are very useful and server the following purposes:

NOTES

- Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.
- Use documents help the users in effectively using the system.
- Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.
- Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

Different types of software documents can broadly be classified into the following:

- Internal documentation
- External documentation

***Internal documentation:*** is the code comprehension features provided as part of the source code itself. Internal documentation is provided through appropriate module headers and comments embedded in the source code. Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc. Careful experiments.

Suggest that out of all types of internal documentation meaningful variable names is most useful in understanding the code. This is of course in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without thought. For example, the following style of code commenting does not in any way help in understanding the code.

***a = 10;/* a made 10*/***

But even when code is carefully commented, meaningful variable names still are more helpful in understanding a piece of code. Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines.

***External documentation:*** is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

## STUDENT ACTIVITY

1. What is meant by Implementation?

_______________________________________________

_______________________________________________

_______________________________________________

_______________________________________________

_______________________________________________

2. Describe the relation between system design and Implementation.

_______________________________________________

_______________________________________________

_______________________________________________

_______________________________________________

_______________________________________________

## SUMMARY

- Implementation is the process of first ensuring that the information system is operational and then allowing users to take over its operation for use and evaluation.
- Software implementation should be done with proper planning, hsty decisions lead to problems and delays.
- Design is the most creative and challenging please of the system development life cycle. The term design describe the final system and process by which it is developed.
- Implementation phase is less creative than system design. It is mainly concerned with user training, site selection and prep-aration and file conversion.
- Generally almost all Software Engineers will used to formulate their own coding standards and expect their tea should implement it. The coding standard will give guidelines for the good programming style, but the implementation of the guidelines is left to the discretion of the individual engineers.
- Code reviewing is a more efficient way of removing errors compared to testing because code review identifies errors whereas testing identifies failures. Therefore, after identifying the failures efforts may be taken to locate and fix the errors.

NOTES

## REVIEW QUESTIONS

1. Explain the software Implementation Guidelines.
2. What are the good coding guidelines?
3. Define code review.
4. What is a code analysis technique?
5. With help walk through and inspection explain code analysis technique.
6. How the code can inspect?
7. What are the different approaches available for clean room testing?
8. Describe the relation between system design and Implementation.
9. What are the different approaches available for clean room testing?

## FURTHER READINGS


1. **Software Engineering,** Bharat Bhushan Agarwal, Sumit Prakash Tayal, Firewall Media.
2. **Software Engineering**, D. Sunder, University Science Press.

# UNIT V SOFTWARE MAINTENANCE

NOTES

## ★ STRUCTURE ★

5.0 Learning Objectives
5.1 Introduction
5.2 Necessity of software Maintenance
5.3 Types of software Maintenance
5.4 Problems Associated With Software Maintenance
5.5 Software Reverse Engineering
5.6 Legacy software Products
5.7 Factors on which Software Maintenance Activities Depend
5.8 Software Maintenace Process Models
5.9 Software-Re-Engineering
5.10 Estimation of Approximate Maintenance Cost
5.11 Documentation
5.12 Case and its scope
5.13 Levels of CASE
5.14 Architecture of CASE Environment
5.15 Building Blocks for CASE
5.16 CASE Support in Software Life Cycle
5.17 Objective of CASE
5.18 CASE Repository
5.19 Characteristics of CASE Tools
5.20 CASE Classification
5.21 Categories of CASE Tools
5.22 Advantages of CASE Tools
5.23 Disadvantages of CASE Tools
5.24 Limitations of CASE Tools
5.25 CASE for Future
- *Summary*
- *Review Questions*
- *Further Readings*

NOTES

## 5.0 LEARNING OBJECTIVES

*After studying the unit, you will be able to:*

- explain necessity and types of software maintenance
- describe software reverse engineering.
- illustrate architecture of CASE environment and documentation.
- explain characteristics, classification, categories, advantage, disadvantages and limitations of CASE tools.
- give a presentation on CASE for future.

## 5.1 INTRODUCTION

It is a task that every development team has to face when the software is delivered to the customer's site, installed and is operational. In general, it means fixing things that breaks out or wear out. In software, nothing wears out it is either wrong from the beginning or we decide later that we want to do something different.

It is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities and optimization. Because change is inevitable, mechanisms, must be developed for evaluating, controlling and making modifications. So any work done to change the software after it is in operation is considered to be maintenance work.

## 5.2 NECESSITY OF SOFTWARE MAINTENANCE

Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform is changed, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts. Therefore, it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

NOTES

## 5.3 TYPES OF SOFTWARE MAINTENANCE

There are basically three types of software maintenance. These are:

- **Corrective:** Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.
- **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
- **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

## 5.4 PROBLEMS ASSOCIATED WITH SOFTWARE MAINTENANCE

Software maintenance work typically is much more expensive than what it should be and takes more time than required. In software organizations, maintenance work is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities. Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

## 5.5 SOFTWARE REVERSE ENGINEERING

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce

NOTES

the necessary documents for a legacy system. Reverse engineering is becoming important, since, legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing of its functionalities. A process model for reverse engineering has been shown in Fig. 1(a). A program can be reformatted using any of the several available pretty printer programs which layout the program neatly. Many legacy software products with complex control structure and un thoughtful variable names are difficult to comprehend. Assigning meaningful variable names is important because meaningful variable names are the most helpful thing in code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

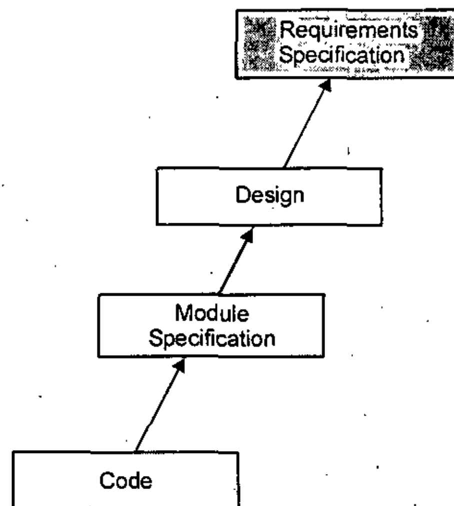



**Fig .1** *(a) A process model for reverse engineering*

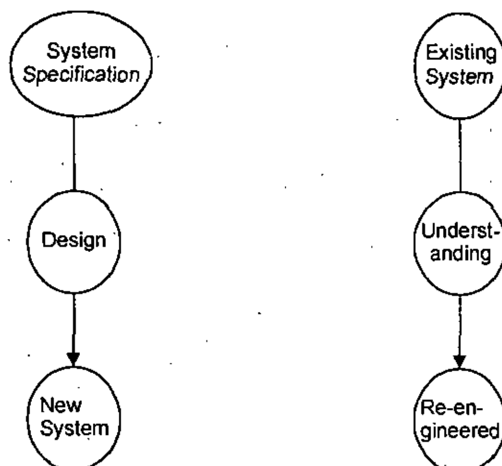



**Fig. 1** *(b) New software development* *(c) Re-engineering*

NOTES

After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Fig.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

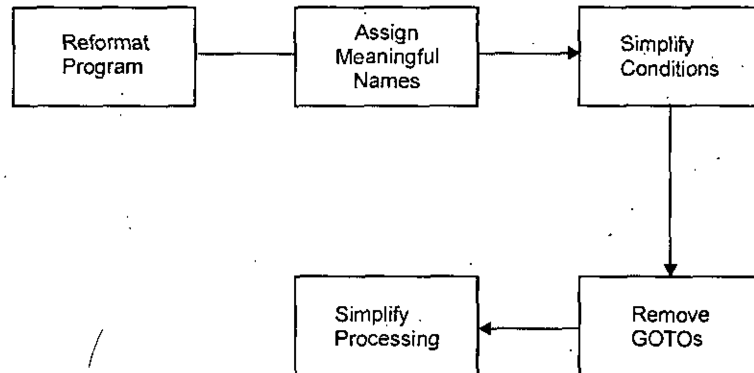



**Fig. 2** *Cosmetic changes carried out before reverse engineering*

## 5.6 LEGACY SOFTWARE PRODUCTS

It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

## 5.7 FACTORS ON WHICH SOFTWARE MAINTENANCE ACTIVITIES DEPEND

The activities involved in a software maintenance project are not unique and depend on several factors such as:

- the extent of modification to the product required
- the resources available to the maintenance team
- the conditions of the existing product (*e.g.*, how structured it is, how well documented it is, etc.)
- the expected project risks, etc.

NOTES

When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents. But more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

## 5.8 SOFTWARE MAINTENANCE PROCESS MODELS

Two broad categories of process models for software maintenance can be proposed. The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in Fig.3.

In this approach, the project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle team, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the re-engineered system becomes easier as the program traces of both the systems can be compared to localize the bugs.

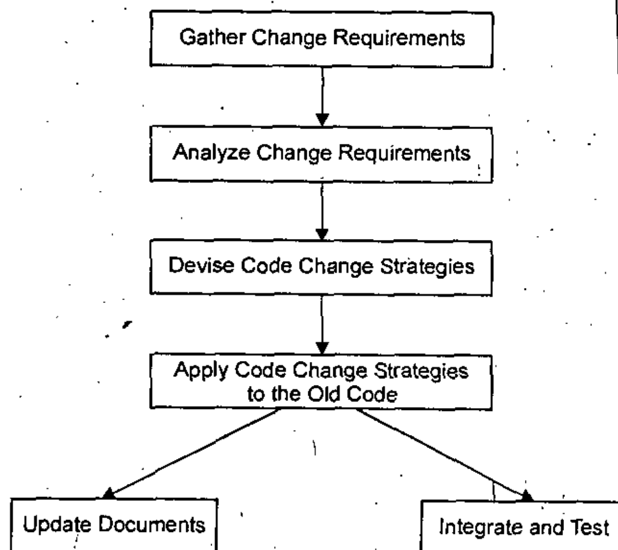



**Fig. 3** *Maintenance process model 1*

NOTES

The second process model for software maintenance is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software re-engineering. This process model is depicted in Fig. 4. The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analyzed (abstracted) to extract the module specifications. The module specifications are then analyzed to produce the design. The design is analyzed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more efficient design. However, this approach is more costly than the first approach.

An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15% (as shown in Fig.5). Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2:

- Re-engineering might be preferable for products which exhibit a high failure rate.
- Re-engineering might also be preferable for legacy products having poor design and code structure.

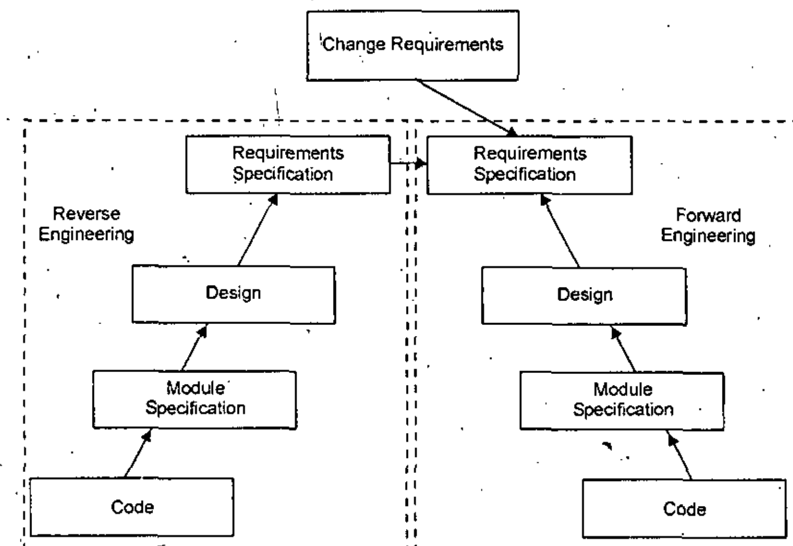



**Fig. 4** *Maintenance process model 2*

NOTES

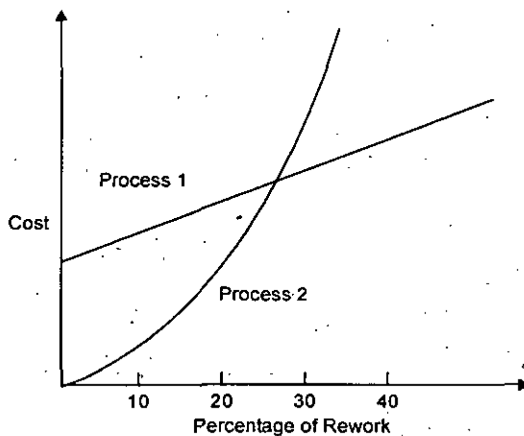



**Fig. 5** *Empirical estimation of maintenance cost versus percentage rework*

## 5.9 SOFTWARE RE-ENGINEERING

Software re-engineering is a combination of two consecutive processes *i.e.*, Software reverse engineering and software forward engineering as shown in the Fig. 4.

## 5.10 ESTIMATION OF APPROXIMATE MAINTENANCE COST

It is well known that maintenance efforts require about 60% of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost. Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic (ACT).

Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$\text{ACT} = \frac{\text{KLOC}_{\text{added}} + \text{KLOC}_{\text{deleted}}}{\text{KLOC}_{\text{total}}}$$

Where, *KLOC added* is the total kilo lines of source code added during maintenance. *KLOC deleted* is the total KLOC deleted during maintenance. Thus, the code that is changed, should be counted in both the code added and the code deleted. The Annual Change Traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost: maintenance cost = *ACT* × development cost.

Most maintenance cost estimation models, however, yield only approximate results because they do not take into account several factors such as

experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

NOTES

## 5.11 DOCUMENTATION

Software documentation is the written of facts about a software system recorded with the intent to convey purpose, content and clarity. The recording process usually begins when the need for the system is conceived and continues until the system is no longer in use.

### User Documentation

It refers to those documents, containing descriptions of the functions of a system without reference to how these functions are implemented. A list of user documentation is given in the following table:

**Table 1**

| *S.No.* | *Document* | *Function* |
|---|---|---|
| 1 | System overview | About general description of system's functions. |
| 2 | Beginners guide | It explains how to start the system. |
| 3 | Reference guide | Provides depth description of each system facility and how it can be used. |
| 4 | Enhancement | Summary of features. |
| 5 | Quick reference | Servers as a factual lookup. |
| 6 | System administration | It provides system informations like networking, security and upgrading. |

### System Documentation

It refers to those documentation containing all facets of system including analysis, specifications, design, implementation testing, security, error diagnosis and recovery.

## 5.12 CASE AND ITS SCOPE

CASE stands for Computer Aided Software Engineering.

"CASE is a tool which aids a software engineer to maintain and develop software". ***The workshop for software engineering is called an Integrated Project Support Environment (IPSE) and the tool set that fills the workshop is called CASE.***

CASE is a computer aided software engineering technology. CASE is an automated support tool for the software engineers in any software engineering process.

Software engineering mainly includes the following processes:

(*i*) Translation of user needs into software requirements

(*ii*) Transaction of software requirements into design specification

(*iii*) Implementation of design into code

(*iv*) Testing of the code

(*v*) Documentation.

CASE technology provides software process support by automating some process activities and by providing information about the software, which is being developed. Examples of activities, which can be automated using CASE, include:

1. The development of graphical system models as part of the requirements specification or the software design.
2. Understanding a design using a data dictionary, which holds information about the entities and relations in a design.
3. The generation of user interfaces from a graphical interface description, which is created interactively by the user.
4. 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.* The use of Computer Aided Software Engineering (CASE) tool reduce the effort of development of achieving quality goals and managing change and configuration throughout the product life cycle, it also help the project manager, the software developer and other key personnel to improve their productivity in the development team.

## 5.13 LEVELS OF CASE

There are three different levels of CASE technology:

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 include tool to support process modeling 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 management support tools.

NOTES

## 5.14 ARCHITECTURE OF CASE ENVIRONMENT

The architecture of CASE environment is illustrated in the Fig 6.

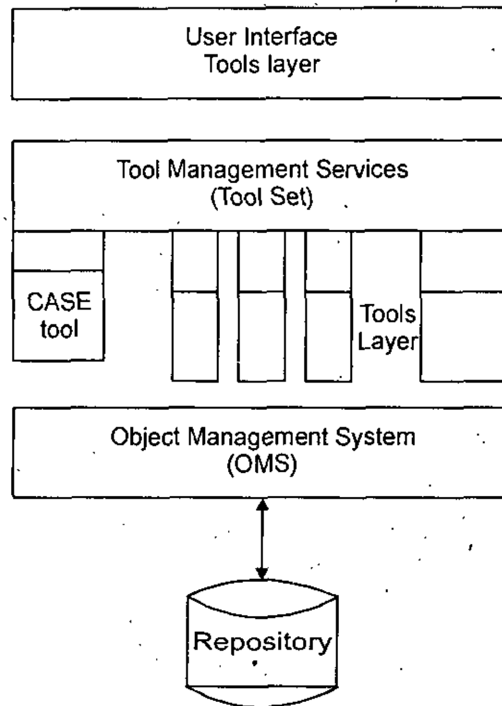



**Fig. 6** *Architecture of CASE Environment*

The important components of a modern CASE environment are the user interface, the Tools Management System (Tools set), the Object Management System (OMS) and the repository. These various components are discussed as under:

### 1. User Interface

It provides a consistent framework for accessing different tools; thus making it easier for the user to interact with different tools and reduces learning time of how the different tools are used.

### 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, coordinates 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.

NOTES

### 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.

### 4. Repository

It is the CASE database and the access control functions that enable the OMS to interact with the database. The word CASE repository is referred in different ways such as project database, IPSE database, data dictionary, CASE database and so on.

## 5.15 BUILDING BLOCKS FOR CASE

The building blocks for CASE are illustrated in Fig. 7

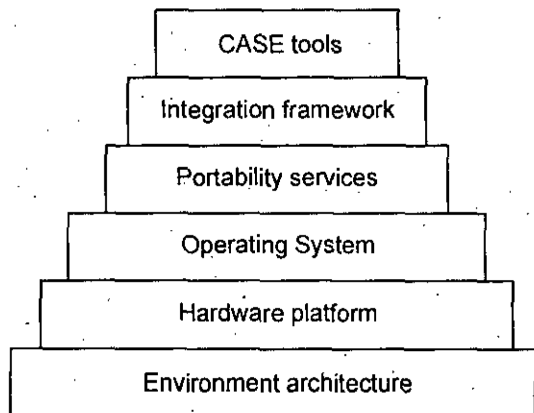



**Fig. 7** *CASE Building Blocks*

### 1. 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.

NOTES

### 2. Portability Services

A set of portability service 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.

### 3. Integration Framework

It is a collection of specialized programs that enables individual CASE tools to communicate with one another, to create a project database.

### 4. CASE Tools

CASE Tools are used to assist software engineering activities (like analysis modeling, code generation etc.) either communicating with other tools, project database (integrated CASE environment) or as point solutions.

## 5.16 CASE SUPPORT IN SOFTWARE LIFE CYCLE

There are various types of support that CASE provides during the different phases of a software life cycle.

### 1. Prototyping Support

The prototyping is useful to understand the requirements of complex software products, to market new ideas and so on. The prototyping CASE tools requirements are as follows:

(*i*) Define user interaction

(*ii*) Define the system control flow

(*iii*) Store and retrieve data required by the system

(*iv*) Incorporate some processing logic.

Few features, which are supported by prototyping tools, are:

- Main use of prototyping CASE tool is developing Graphical User Interface (GUI) development. The user should be allowed to define all data entry forms, menus and control.
- Integrate well with the data dictionary of a CASE environment.
- 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.

NOTES

## 2. 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 (at the analysis level) 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 comprises 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 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 based applications.

## 4. Test CASE Generator

The CASE tool 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 interfaces. In addition to the functions noted, many test management tools also serve as generic test drivers. 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, and then invokes the software to be tested.

NOTES

## 5.17 OBJECTIVE OF CASE

### 1. Improve Productivity

Most organizations use CASE to increase the speeds 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

When tools improve processes, they usually 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 (*i.e.*, deciding the best task to perform to achieve the desired result). Tools can suggest procedures (the right way) 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

- The start-up cost of purchasing and using CASE
- The high cost of training personnel
- The big benefits of using CASE come in the late stages of the SDLC
- CASE often lengthens the duration of early stage of the project
- CASE tools cannot 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
- IS personnel view CASE as a threat to their job security.

Despite these issues, in long-term, CASE is very good. The functionality of CASE tools is increasing and the costs are coming down. During the next several years, CASE technologies and the market for CASE will begin to mature.

NOTES

## 5.18 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 often use formal notation in data dictionary, a 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.

NOTES

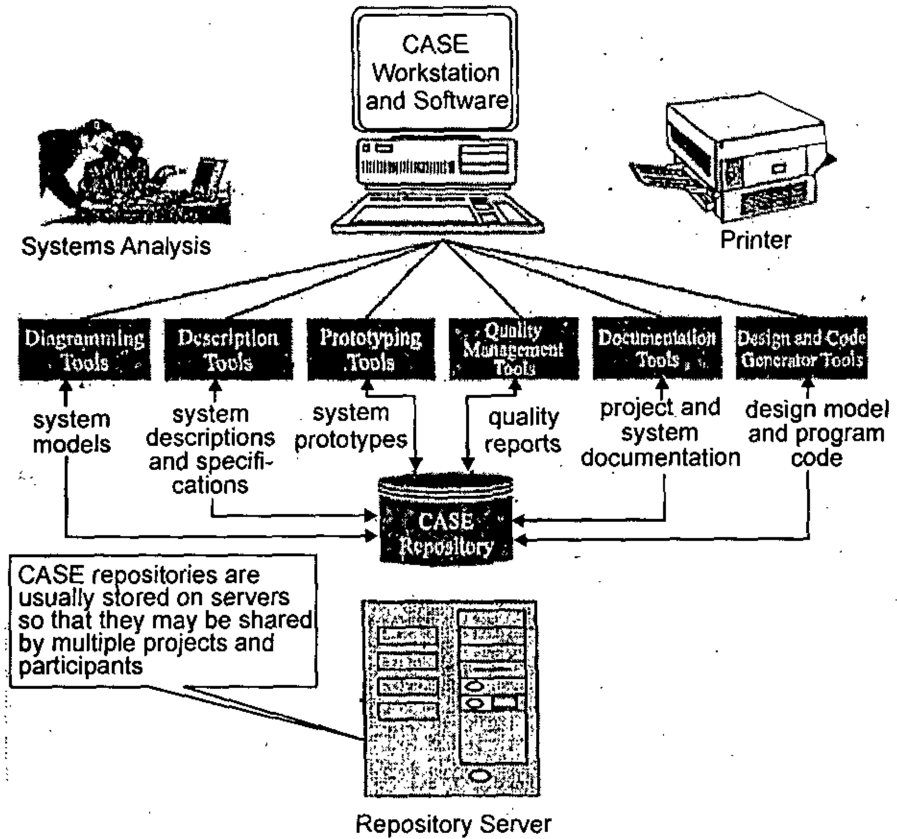



**Fig. 8** *CASE Repository*

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 tool integration, there are two additional advantages of using a comprehensive CASE repository that relate to project management and reusability.* The CASE repository provides a wealth of information to the project manager and allows the manager to exert an appropriate amount of control on the project. If all organizational systems were created using CASE technology with a common repository, it would be possible to reuse significant portions of prior systems in the development of new ones.

## 5.19 CHARACTERISTICS OF CASE TOOLS

All CASE tools have the following characteristics:

1. A graphic interface to draw diagrams, charts, models (upper case, middle case, lower case)
2. An information repository, a data dictionary for efficient information management selection, usage, application and storage
3. Common user interface for integration of multiple tools used in various phase
4. Automatic code generators
5. Automatic testing tools.

## 5.20 CASE CLASSIFICATION

NOTES

CASE classifications help us understand the different types of CASE tools and their role in supporting software process activities. There are various different ways of classifying CASE tools, each of which gives us a different perspective on these tools. In this section, I discuss CASE tools from three of these perspectives, namely:

1. A functional perspective where CASE tools are classified according to their specific function.
2. A process perspective where tools are classified according to the process activities which they support.
3. An integration perspective where CASE tools are classified according to how they are organized into integrated units which provide support for one or more process activities.

### List of CASE Tools

| *Application* | *Case Tool* | *Purpose of Tool* |
|---|---|---|
| 1. Planning | Excel spreadsheet, MS-Project, PERT/CPM Network, Estimation tools | Functional Application: Planning, scheduling, control |
| 2. Editing | Diagram editors, Text editors, Word Processors | 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 Interface, connectivity | Programming of high quality with no errors, System Integration |
| 7. Templates | | Guided Systematic development |
| 8. Re-engineering tools | Cross reference systems, program re-structuring systems | Reverse-engineering to find structure, design and design information |
| 9. Program analysis tool | Cross reference generators Static analyzers, dynamic analyzers | Analyses risks, functions, features |

NOTES

## 5.21 CATEGORIES OF CASE TOOLS

The schematic diagram of CASE tools is drawn below in Fig. 9.

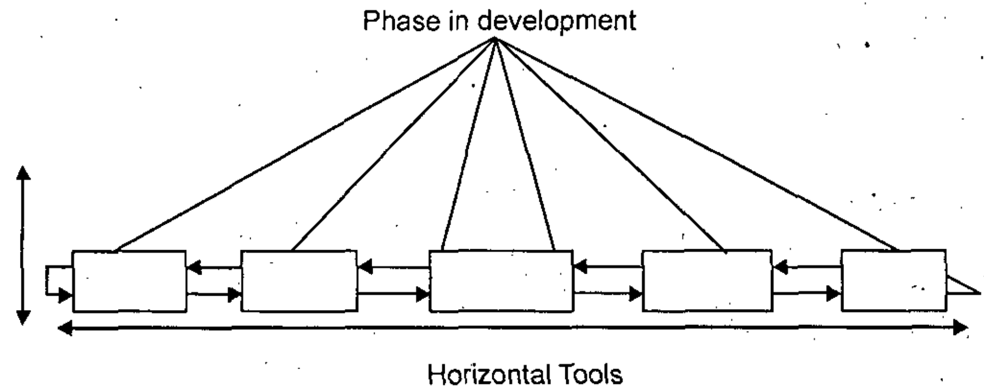



**Fig. 9** *Categories of CASE Tools*

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 subcategories of vertical CASE tools:

*(i)* **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.

*(ii)* **Second Category.** It is a tool that is used in more than one phase, but does not support moving from one phase to the next. These tools ensure that the developer does move on the next phase as appropriate.

### 2. Horizontal CASE Tools

These tools support automated transfer of information between the phases 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:

NOTES

### 1. *Upper CASE Tools/Front-End CASE Tools*

CASE tools are 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:** Analysis tools enable automatic checking for incomplete, inconsistent, or incorrect specifications in diagrams, forms and reports.

### 2. *Lower CASE or Back-End 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.

### 3. *Cross Life Cycle CASE or Integrated Tools*

CASE tools used to support activities that occur across multiple phases of the SDLC. While such tools include both front-end and back-end capabilities, they also facilitate design, management, and maintenance of code. In addition, they provide an efficient environment for the creation, storage, manipulation, and documentation of systems.

### 4. *Reverse Engineering Tools*

These tools build bridges from lower CASE tools to upper CASE tools. They help in the process of analyzing existing applications, performa and database code to create higher level representations of the code.

## 5.22 ADVANTAGES OF CASE TOOLS

NOTES

The major benefits of using CASE tools include the following:

1. Improved productivity
2. Better documentation
3. Improved accuracy
4. Intangible benefits
5. Improved quality
6. Reduced lifetime maintenance
7. Opportunity to non-programmers
8. Reduced cost of software
9. Produce high quality and consistent documents
10. Impact on the style of a working of company
11. Reduce the drudgery in a software engineer's work
12. Increase speed of processing
13. Easy to program software
14. Improved coordination among staff members who are working on a large software project
15. An increase in project control through better planning, monitoring and communication.

## 5.23 DISADVANTAGES OF CASE TOOLS

1. **Purchasing of CASE tools is not an easy task.** Its cost is very high. Due to this reason small software development firm do not invest in case tools.
2. **Learning Curve:** In general cases programmer productivity may fall in initial phase of implementation as user need time to learn this technology.
3. **Tool Mix:** It is important to make proper selection of case tools to get maximum benefit from the case tools, so wrong selection may lead to wrong result.

NOTES

## 5.24 LIMITATIONS OF CASE TOOLS

The major limitations of using CASE tools include:

- *Cost*
- Learning Curve
- Tool Mix

### Cost

Using CASE tools is a very costly affair. In fact, most firms engaged in software development on a small scale do not invest in CASE tools because they think that the benefits of CASE are justifiable only in the development of large systems.

The cost of outfitting every system developer with a preferred CASE tool kit can be quite high. Hardware and systems, software, training and consulting are all factors in the total cost equation of using CASE tools.

### Learning Curve

In most CASES, programmer productivity may fall in the initial phase of implementation, because users need time to learn the technology. In fact, a CASE consulting industry has evolved to support uses of CASE tools.

The consultants offer training and on-site services that can be crucial to accelerate the learning curve and to the development and use of the tools.

### Tool Mix

It is most important to make an appropriate selection of tool mix to get cost advantage. CASE integration and data integration across all platforms is also very important. The ability to share the results of work done on one CASE tool with another CASE tool is perhaps the most important type of CASE integratio.

## 5.25 CASE FOR FUTURE

CASE is not the only new technology that promises to eliminate problems of software development productivity. Object-oriented languages, artificial intelligence tools, and other newly popular development tools can also help achieve this goal.

In fact, CASE might soon become the pivot around all, which all these technologies turn. CASE is promised to be the foundation technology for development approaches.

CASE transcends programming languages *i.e.*, it is a tool framework while languages are the material the tools operate on. Here is a good example:

> CASE is the tool bench, holding saws and drills and hammers; programming languages are the wood from which you carve the model.

CASE tools can potentially automate any kind of software development. They can be tailored to fit almost any existing development environment. However, only a few environments are popular enough to ensure substantial sales of CASE products.

For the next few years, until it is fully coupled to methodologies, CASE will target fairly standard system types and development platforms.

## STUDENT ACTIVITY

1. What are the different levels of CASE?

2. Explain the components of CASE architecture.

## SUMMARY

- Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features.
- Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code.
- Software re-engineering is a combination of two consecutive processes *i.e.*, Software reverse engineering and software forward engineering.
- CASE is a computer aided software engineering technology.
- A set of portability service provides a bridge between CASE tools and their integration framework and the environment architecture.
- Analysis and design tools enable a software engineer to create models of the system to be built.
- A CASE repository is a system developer database.
- CASE classifications help us understand the different types of CASE tools and their role in supporting software process activities.

- Code generators automate the preparation of computer software.
- The cost of outfitting every system developer with a preferred CASE tool kit can be quite high.

NOTES

## REVIEW QUESTIONS

1. What for software products are required to maintain?
2. What are the different types of maintenance that a software product might need? Why are these maintenance required?
3. What are the disadvantages associated with software maintenance?
4. What do you mean by the term software reverse engineering? Why is it required? Explain the different activities undertaken during reverse engineering.
5. What is legacy software product? Explain the problems one would encounter while maintaining a legacy product.
6. What are the different factors upon which software maintenance activities depend?
7. What do you mean by the term software re-engineering? Why is it required?
8. If the development cost of a software product is Rs. 10,000,000/-, compute the annual maintenance cost given that every year approximately 5% of the code needs modification. Identify the factors which render the maintenance cost estimation inaccurate.
9. Legacy software products are very difficult to maintain.
10. Legacy products are those products which have been developed long time back.
11. In the process of reverse engineering; we change the functionalities of an existing code.
12. List put documentation and also explains their purpose.
13. Discuss Building blocks for CASE.
14. What do you understand by OMS?
15. Explain some characteristics of CASE Tools.
16. Name the different Categories of CASE Tools. Also show it diagrammatically.
17. Why analysts use CASE repositories? Give some reasons.
18. What are the advantages and disadvantages of CASE Tools?

## FURTHER READINGS

1. **Software Engineering**, Bharat Bhushan Agarwal, Sumit Prakash Tayal, Firewall Media.
2. **Software Engineering**, D. Sunder, University Science Press.