

स्वाध्याय

स्वमन्थन

स्वावलम्बन

**UTTAR PRADESH RAJARSHI TANDON OPEN UNIVERSITY**  
(Established vide U.P. Govt. Act No. 10, of 1999)



Indira Gandhi National Open University



UP Rajarshi Tandon Open University

**BCA- 19**  
**Introduction to Software**  
**Engineering**

**FIRST BLOCK : Software Engineering Concepts**

**SECOND BLOCK : Software Quality Concepts and  
Case Tools**

**Shantipuram (Sector-F), Phaphamau, Allahabad - 211013**



Uttar Pradesh  
Rajarshi Tandon Open University

**BCA-19**  
**Introduction to Software  
Engineering**

Block

**1**

**Software Engineering Concepts**

<b>UNIT 1</b>	
<b>Introduction to Software Product, Component and Characteristics</b>	<b>5</b>
<b>UNIT 2</b>	
<b>Software Process Management</b>	<b>15</b>
<b>UNIT 3</b>	
<b>Project Planning and Control</b>	<b>28</b>
<b>UNIT 4</b>	
<b>Risk Management Concepts</b>	<b>39</b>

---

## **COURSE INTRODUCTION**

---

This course presents an overview of Software Engineering principles. The course covers different aspects on Software Product component, design and development, documentation and software life cycles, requirement analysis and specification, Issues relating to Human Resource Management, project planning and scheduling, conflicts and standards. This course is a complete courses and also covers risk management, technical planning, project tracking and scheduling, project metrics and case tools.

A separate block is devoted to software tools and environment and software quality concepts.

### **Further Readings**

1. Roger S. Pressman – Software Engineering (A Practitioner's Approach).

---

## **BLOCK INTRODUCTION**

---

In the early days, programming was viewed as an artistic form. The programmer often used hit and trial methods. With this tie between computers and solving problems, computer science examines all aspects of Problem Solving and the integration of Software into this process. This block covers different aspect of Software Engineering discipline. The first unit focuses on Software Product, Component and characteristics, Software Engineering concepts, Documentation and Software Process. The second unit reflects issues relating to Human Resource Management for developing Quality Software. The important issue of Project Planning and Scheduling, conflict and Standards are discussed in Unit 3. Finally, Risk Management Concepts, Technical Planning and Project Tracking concepts are discussed in Unit 4. The students of this course are also advised to go through at least one book on Software Engineering along with this material.

---

# **UNIT 1 INTRODUCTION TO SOFTWARE PRODUCT, COMPONENT & CHARACTERISTICS ENGINEERING**

---

## **Structure**

- 1.0 Introduction
- 1.1 Objective
- 1.2 Software Product, Components and Characteristics
- 1.3 Software Engineering Concepts
  - 1.3.1 Phases
  - 1.3.2 The Study Phase
  - 1.3.3 The Design Phase
  - 1.3.4 The Development Phase
  - 1.3.5 The Operation Phase
- 1.4 Documentation of the Software Product
- 1.5 Software Process and Models
  - 1.5.1 Software life Cycle
  - 1.5.2 Requirement Analysis and Specification
  - 1.5.3 Design and Specification
  - 1.5.4 Coding and Module Testing
- 1.6 Summary
- 1.7 Model Answer
- 1.8 Further Readings

---

## **1.0 INTRODUCTION**

---

The first unit focused on Software Product, Component and Characteristics. This unit discuss in details the evolution process of Software Engineering life cycle. A sample waterfall model is also discussed in this unit. Software Engineering concepts and its phases are also included in this unit. The documentation part is also included in this unit, Software Documentation is a continuous and parallel activity in development process. The students are advised to go through at least one standard book in Software Engineering along with this material.

---

## **1.1 OBJECTIVES**

---

After going through this unit, you should be able to:

- Define Software Product . Component and characteristics.
- Explain what is documentation of Software Product.
- Describe what is Software Process and what is Software Life Cycle.
- Describe a Generic View of Software Engineering.

---

## 1.2 SOFTWARE PRODUCT, COMPONENTS AND CHARACTERISTICS

---

We are entering an information age, one in which the management of the information resource of organization will be of vital importance. Business information systems are systems that use these resources to convert data into information in order to improve productivity. Business information systems usually are composed smaller systems, called subsystems. Computer hardware and software are important resources that support information systems and subsystems.

Systems analysis is a general terms that refers to an orderly, structured process for solving problems. This process, when applied to information systems, is called the life cycle-methodology. Four phases-study, design, development, and operation-make up the life cycle of computer-related business systems. A systems analyst is a person who performs systems analysis during any or all of the life-cycle phases. The systems analyst not only analyzes information system problems, but also synthesizes new systems to solve these problems.

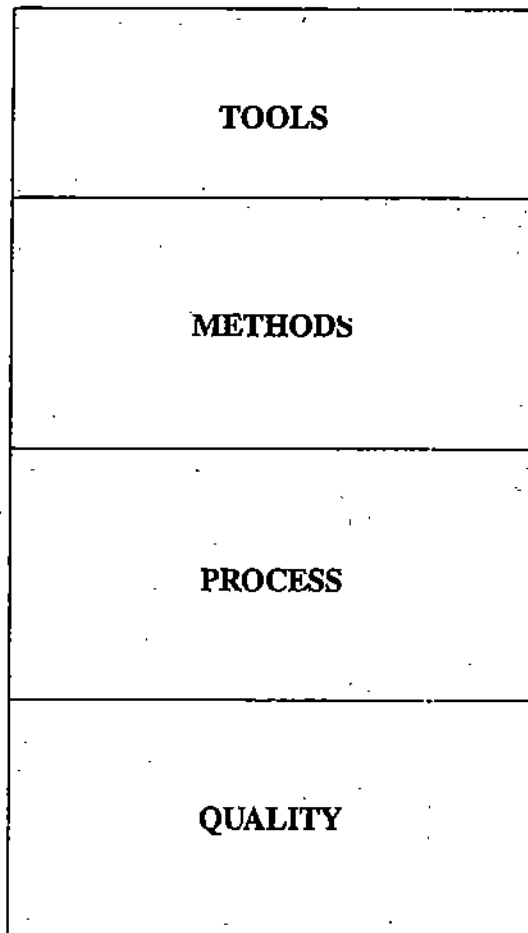
The four information eras are: the Early Era (1940-1955), the Growing Era (1955-1965), the Refining Era (1965-1980), and the Maturing Era (1980- ). The Early Era concentrated on hardware, and human-machine communication was very difficult. The growing Era improved this communication through the introduction of English-like programming languages; however, techniques for managing computer-related projects were lacking. During the Refining Era, explosive growth occurred in the development of large (midi and maxi) and small (micro and mini) computer systems and in their applications. Developments in microelectronics technology contributed significantly to this growth. Throughout most of the Refining Era, in spite of a proliferation of applications, difficulties were encountered in using computer to solve business problems. However, toward the end of this era, a structured system analysis process-called the life-cycle methodology-came into increasing use as a means of developing usable business information systems. Structured techniques for the analysis, design, and development of computer-related information systems will be enhanced in the maturing Era. These techniques will be used to develop information systems in applications areas such as distributed data processing, the automated office, and management-decision support. This will be an era in which information will be acknowledged as an important corporate resource. The systems analyst will assume an important role in managing the information resources of the corporation.

The computer-based business system also contains hardware components; however, its most significant characteristics is a software end-product. Software may be defined as a collection of programs or routines that facilitates the use of a computer. This definition includes operation systems, which facilitate the general use computers, and application programs, which are written to solve specific problems. The latter is the end-product associated with a computer-based information system. Software, in contrast with hardware, does not possess attributes that can readily be observed and measured from concept to end product. The software end - product is information. Although it may be stored or printed on a physical medium, such as a magnetic disk, a reel of tape, or a sheet of paper, information is transient and fragile compared with hardware.

Many of the past difficulties in developing effective computer-based business systems stemmed not only from belated efforts to apply management controls, but also from failure to recognize that techniques applicable to the development of hardware end-products could not be applied without modification to the development of software end-products. However, as a result of experience gained

from large government and commercial software projects in the latter part of the 1960s and throughout the 1970s, the concept of life-cycle management was adapted to fit the development of computer based business systems.

The key to modifying the life-cycle concept for the management of software projects was the recognition that, although supporting documentation accompanies a physical product throughout its development, documentation is the software product.



**Figure 1.1 : Software engineering layers**

---

## **1.3 SOFTWARE ENGINEERING PHASES**

---

**Life-Cycle Phases and the Life-Cycle Manager :**

The life cycle of a computer-based system exhibits distinct phases. These are:

### **1.3.1 Phases**

**Performance of the Cycle**

The life-cycle methodology for developing complex systems is modular, top-down procedure. In the study phases, modules that describe the major functions to be

performed by the system are developed. The procedure is called top-down because in successive phases the major modules are expanded into additional, increasingly detailed, phases the major modules are expanded into additional, increasingly detailed, modules. Powerful graphic tools have been developed to structure the *top-down* design and development phase activities in detail. For the present, we can summarize the principal tasks associated with each of the phases of the life cycle of a computer-based business system.

### 1.3.2 The Study Phase

This is the phases during which a problem is identified, alternate system solutions are studied, and recommendations are made about committing the resources required to design the system. Tasks performed in the study phase are grossly analogous to (1) determining that a shelter from the elements is needed, and (2) deciding that a two-bedroom house is a more appropriate shelter than a palace, a cave, or other possible selections.

### 1.3.3 The Design Phase

In this phase the detailed design of the system selected in the study phases is accomplished. This is analogous to drawing the plans for the two-bedroom home decided on in the study phase. In the case of a computer-based business system, design phase activities include the allocation of resources to equipment tasks, personnel tasks, and computer program tasks. In the design phase, the technical specifications are prepared for the performance of all allocated tasks.

### 1.3.4 The development Phase

This is the phase in which the computer-based system is constructed from the specifications prepared in the design phase. Equipment is acquired and installed during the development phase. All necessary procedures, manuals, software specifications, and other documentation are completed. The staff is trained, and the complete system is tested for operational readiness. This is analogous to the actual construction of our two-bedroom house from the plans prepared in its design phase.

### 1.3.5 The Operation Phase

In this phase, the new system is installed or there is a changeover from the old system to the new system. The new system is operated and maintained. Its performance is reviewed, and changes in it are managed. The operation phase is analogous to moving into and living in the house that we have built. If we have performed the activities of the preceding phases adequately, the roof should not leak.

All of the activities associated with each life-cycle phase must be performed, managed, and documented. Hence, we now define systems analysis as the performance, management, and documentation of the activities related to the four life-cycle phases of a computer-based business system. Similarly, we now can identify the system analyst as the individual who is responsible for the performance of systems analysis for all, or a portion of the phases of the life cycle of a business system. The analyst is, in effect, a life-cycle manager.



---

## 1.4 DOCUMENTATION OF THE SOFTWARE PRODUCT

---

The accumulation of documentation parallels the life-cycle performance and management review activities. Documentation is not a task accomplished as a "wind up" activity; rather, it is continuous and cumulative. The most essential documents are called baseline specification (that is, specifications to which change can be referred). There are three baseline specifications:

1. Performance specification : It is completed at the end of the study phase, and describing in the language of the user exactly what the system is to do. It is a "design to" specification.
2. Design specification : It is completed at the end of the design phase, and describing in the language of the programmer (and others employed in actually constructing the system) how to develop the system. It is a "build to".
3. System specification : It is completed at the end of the development phase and containing all of the critical system documentation. It is the basis for all manuals and procedures, and it is an "as built" specification.

The design specification evolves from the performance specification, and the system specification evolves from the design specifications. Since these documents are the only measurable evidence that progress is being made toward the creation of a useful software end-product, it is not possible to manage the life-cycle process without them. Thus, documentation is not only the "visible" software end-product, but also the key to the successful management of the life cycle of computer-based business systems.

---

## 1.5 SOFTWARE PROCESS AND MODELS

---

### 1.5.1 Software Life Cycle

From the inception of an idea for a software system, until it is implemented and delivered to a customer, and even after that, the system undergoes gradual development and evolution. The software is said to have a life cycle composed of several phases. Of these phases result in the development of either a part of the system or something associated with the system, such as a test plan or user manual. In the traditional life cycle model, called the "waterfall model," each phase has well-defined starting and ending points, with clearly identifiable deliverables to the next phase.

A sample waterfall life cycle model comprises the phases, similar to described in next sections.

### 1.5.2 Requirements analysis and specification

Requirements analysis is usually the first phase of large-scale software development project. It is undertaken after a feasibility study has been performed to define the precise costs and benefits of a software system. The purpose of this phase is to identify and document the exact requirements for the system. The customer, the developer, a marketing organization or any combination of the three may perform such study. In cases where the requirements are not clear e.g., for a system that has

never been defined, more interaction is required between the user and the developer. The requirements at this stage are in end-user terms. Various software engineering methodologies advocate that this phase must also produce user manuals and system test plans.

### 1.5.3 Design and specification

Once the requirements for a system have been documented, software engineers design a software system to meet them. This phase is sometime split into two sub-phases: architectural or high-level design and detailed design. High-level design

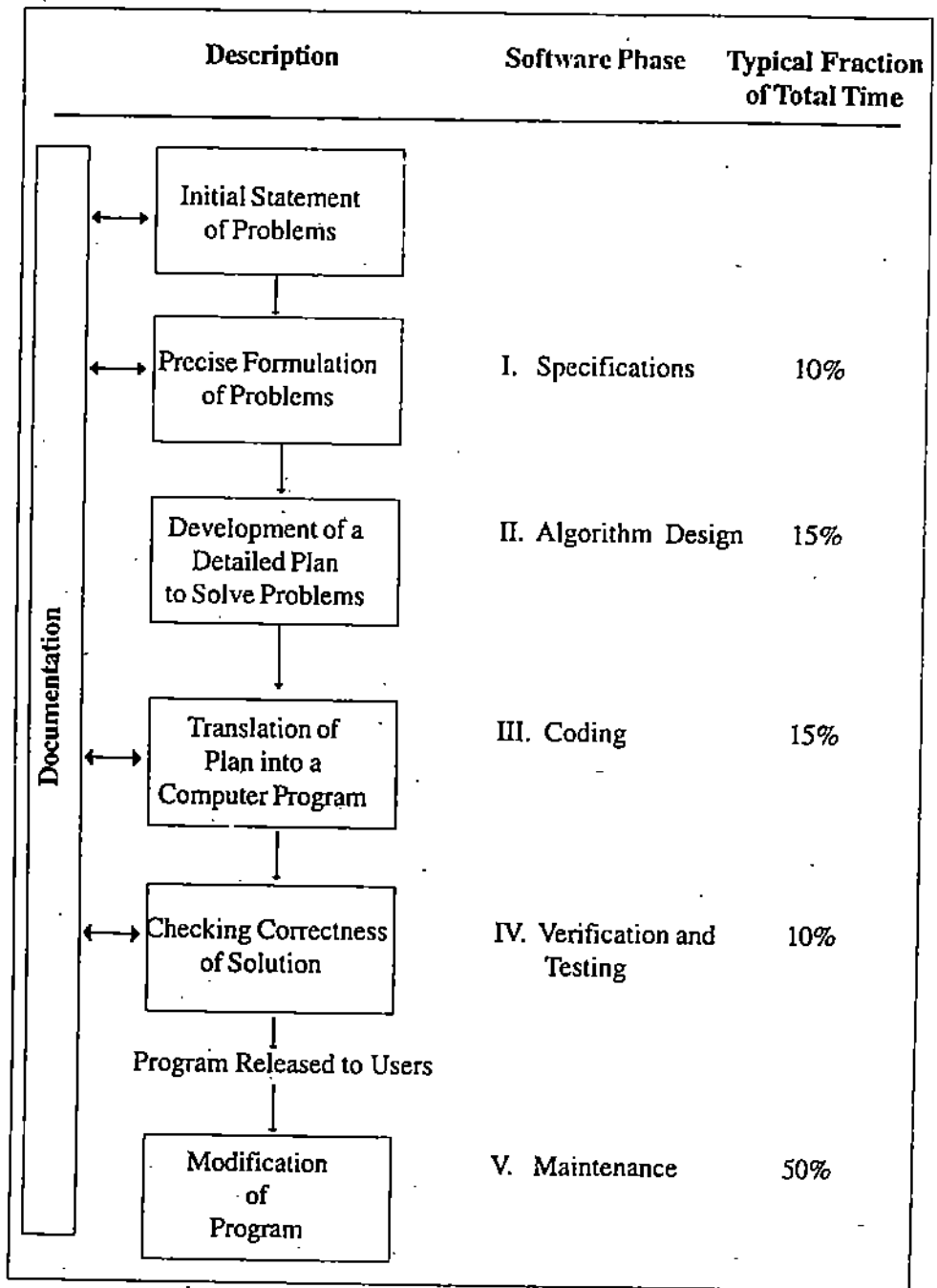


Figure 1.2 : Software Life Cycle

deals with overall module structure and organization, rather than the details of the modules. The high level design is refined by designing each module in detail (detailed design). Separating the requirements analysis phase from the design phase is instance of a fundamental "what/how" dichotomy that we encounter quite often in computer science. The general principle involves making a clear distinction between what the problem is and how to solve the problem. In this case, the requirement phase attempts to specify what the problem is. There are usually many ways that the requirements may be met, including some solutions that do not involve the use of computers at all. The purpose of the design phase is to specify a particular software system that will meet the stated requirements. Again there are usually many ways to build the specified system. In the coding phase, which follows the design phase, a particular system is coded to meet the design specification.

#### 1.5.4 Coding and module testing

This is the phase that produces the actual code that will be delivered to the customer as the running system. The other phases of the life cycle may also develop code, such as prototypes, tests, and test drivers, but these are for use by the developer. Individual modules developed in this phase are also tested before being delivered to the next phase.

- **Integration and system testing:** All the modules that have been developed before and tested individually are put together integrated-in this phase and tested as a whole system.
- **Delivery and maintenance :** Once the system passes the entire test, is delivered to the customer and enters the maintenance phase. Any modifications made to the system after initial delivery are usually attributed to this phase.

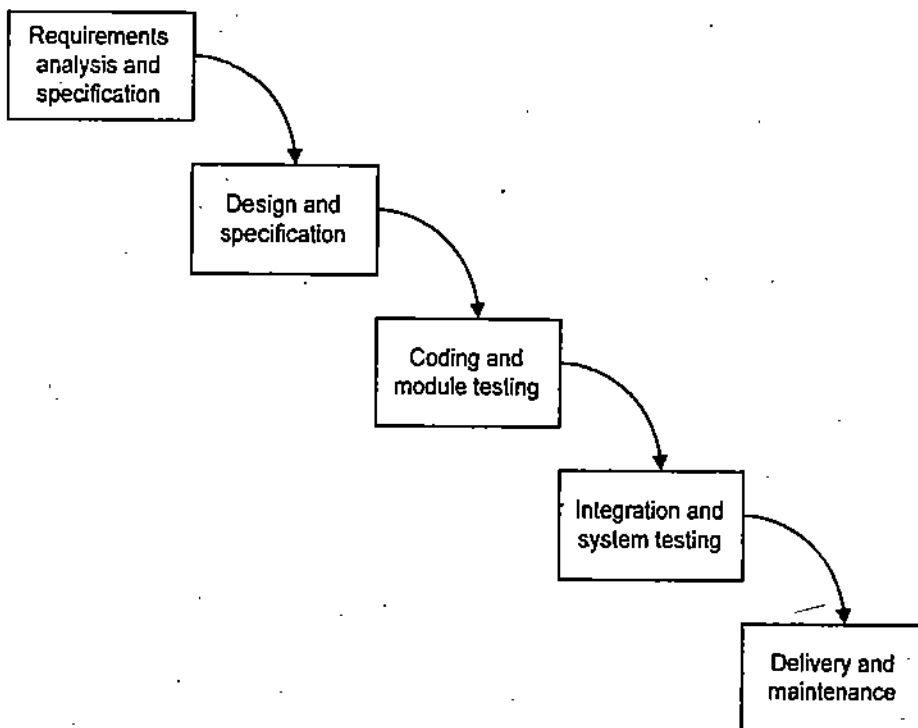


Figure 1.3 : Waterfall Model of Software life Cycle

## 1.6 SUMMARY

As presented above, the phases give a partial, simplified view of the conventional waterfall software life cycle. The process may be decomposed into a different set of phases, with different names, different purpose, and different granularity. Entirely different life cycle schemes may even be proposed, not based on a strictly phased waterfall development. For example, it is clear that if any tests uncover defects in the system, we have to go back at least to the coding phases and perhaps to the design phase to correct some mistakes. In general, any phases may uncover problems in previous phases this will necessitate going back to the previous phases and redoing some earlier work. For example, if the system design phase uncovers inconsistencies or ambiguities in the system requirements, the requirements analysis phase must be revisited to determine what requirements were really intended.

Another simplification in the above presentation is that it assumes that a phase is completed before the next one begins. In practice, it is often expedient to start a phase before a previous one is finished. This may happen, for example, if some data necessary for the completion of the requirement phase will not be available for some time. Or it, might be necessary because the people ready to start the next phases are available and have nothing else to do.

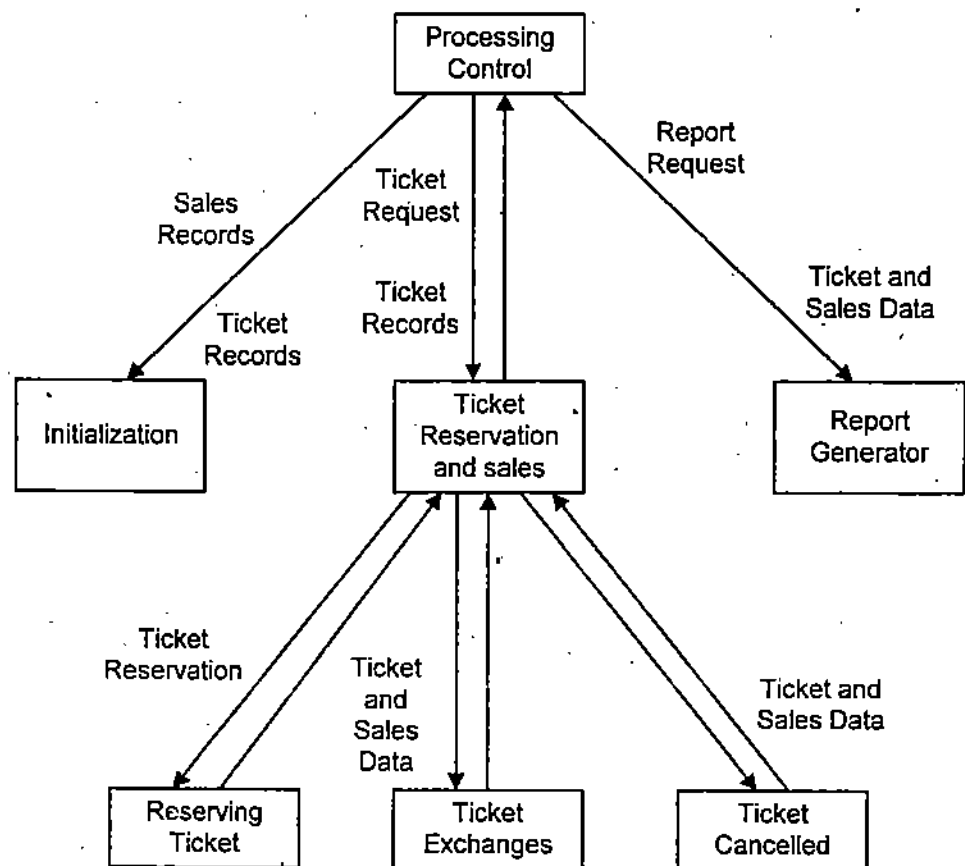


Figure 1.4 : An Structure Chart for Rail-Reservation System showing Data Flow

Most books on software engineering are organized according to the traditional software life cycle model, each, section or chapter being devoted to one phase. Once mastered, the software engineer can apply these principles in all phases of software development, and also in life cycle models that are not based on phased development, as discussed above. Indeed, research and experience over the past decade have shown that there is a variety of life cycle models and that no single one is appropriate for all software systems.

### Check Your Progress

1. Process is
  - (a) Program in High Level Language kept on disk;
  - (b) Contents of Main/mamory;
  - (c) a program in execution;
  - (d) a job in secondary memory;
2. Which of the following is not an example of program documentation?
  - (a) Source code
  - (b) Object code
  - (c) Specification
  - (d) Identifier Names
3. Which of the following is (are) among the legitimate purposes of software documentation?
  - I. To assist in maintaining and modification.
  - II. To describe the capabilities of the program.
  - III. To provide the use with instructions.
  - a) II only
  - b) II and III only
  - c) III only
  - d) I, II and III
4. A top down approach to programming calls for
  - (a) Working from the general to the specific.
  - (b) Postponing the minor decisions.
  - (c) A systematic approach
  - (d) Immediate coding of the problem.

---

## 1.7 MODEL ANSWERS

---

- Q1 - (c)  
Q2 - (b)  
Q3 - (d)  
Q4 - (c)

---

## 1.8 FURTHER READING

---

The students of this course are advised to go through at least one standard book on Software Engineering along with this material.

1. **Software Engineering - A Practitioner's Approach** by **ROGER S. PRESSMAN** :  
**McGraw Hill International Edition**

The students are also advised to search the World Wide Web for Software Engineering material, using standard search Engines.

---

## UNIT 2 SOFTWARE PROCESS MANAGEMENT

---

### Structure

- 2.0 Introduction
- 2.1 Objective
- 2.2 Software Process Management
- 2.3 Human Resource Management
  - 2.3.1 Software Process
  - 2.3.2 Team Leaders
  - 2.3.3 Problem Solving
  - 2.3.4 Influence and Team Building
- 2.4 The Software Team
  - 2.4.1 Democratic Decentralised
  - 2.4.2 Controlled Decentralised
  - 2.4.3 Controlled Centralised
- 2.5 Organisation, Information and Decision
- 2.6 Problem Identification
  - 2.6.1 Principles of Coupling, Cohesion and Information Hiding
  - 2.6.2 Problem Handling Guidelines
- 2.7 Software Crisis
  - 2.7.1 From Programmer's Point of View
  - 2.7.2 From User's Point of View
- 2.8 Role of a System Analyst
- 2.9 Model Answers

---

## 2.0 INTRODUCTION

---

This unit deals with the most important capital in any organisation which is Human Capital. The "people factor" is so important that the Software Engineering Science has developed a capability model to enhance the ability of Software organisations to undertake complex applications. Major Technology Fortune 500 companies attribute their success to their people. Even the profitability of a Tech Company is, if certain key people left the organisation. In this Unit you learnt about Software Process Management, Human Resource Management, the Software Team, Problem Handling Guidelines and Managing the Software crisis.

---

## 2.1 OBJECTIVE

---

After going through this unit, you should be able to:

Describe Software Process Management;

Able to Highlight the Human Management Concept in Software Development Process, and

Describe Problem Handling Guidelines in Software Development.

---

## 2.2 SOFTWARE PROCESS MANAGEMENT

---

Most design methodologies focus on designing a "fresh software". No design methodology really handles designing from an existing design, or changing a design to include new specifications. Change is an inherent property of software, and its strength. By having these methodologies, and people following them, the view is further strengthened that software should be developed as a "fresh product". This clearly is inconsistent with the reality of things and can be argued to hinder productivity growth, as developing a fresh product is likely to be more expensive than developing by reusing existing designs and implementations.

The separation of software process and software products is desirable and has helped software engineers understand both separately and give importance to the process also, which was neglected earlier. However, current process models and their implementations, and the current heavy emphasis on process with the belief that software process determines most properties of software.

Some of the undesirable consequences that we have mentioned are (a) Tendency to make process document heavy and consequently not liked by people in the process, (b) neglect of some of the important software product, like software design, (c) inability of process to model change, which is a fundamental property and strength of software, and (d) tendency to develop "processes" for creative activities like design.

However, some future trends in software engineering seem to be promising. There is an effort to build newer models that are more consistent with the properties of software. There is also an effort to develop more formal methods for software specification, design and verification and software reuse is being targeted as the future software engineering technology - a target that will force development of new methods for design, which use existing designs.

---

## 2.3 HUMAN RESOURCE MANAGEMENT

---

Effective software project management focuses on the three P's *people, problem, and process*. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor, will never have success in project management. A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the problem. Finally, the manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum.

The "people factor" is so important that the Software Engineering Science has developed a *people management capability maturity model (PM-CMM)* "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability".

The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

The PM-CMM is a companion to the software capability maturity model, which guides organizations in the creation of a mature software process.



Before a project can be planned, its objectives and scope should be established, alternative solution is considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost; an effective assessment of risk; a realistic breakdown of project tasks; or a manageable project schedule that provides a meaningful indication of progress.

The software developer and customer must meet to define project objectives and scope. In many cases, this activity begins as part of the system engineering process and continues as the first step in software requirement analysis. Objectives identify the overall goals of the project without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the problem, and more important, attempts to *bound these* characteristics in a quantitative manner.

Once the project objectives and scope are understood, alternative solutions are considered. The alternatives enable managers and professionals to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and other factors.

### 2.3.1 Software Process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of *framework activities* are applicable to all software projects, regardless of their size or complexity. A number of different *task sets* - tasks, milestones, deliverable, 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.

In a study, the engineering presidents of three major technology companies were asked the most important contributor to a successful software project. They answered in the following way:

- P1: If I had to pick one thing out that is most important in our Organisation, I'd say it's not the tools that we use, it's the people.
- P2: The most important ingredient that was successful on this project was having smart people.... very little else matters in my opinion. ... The most important thing you do for a project is selecting the staff. The success of the software development organizations is very much associated with the ability to recruit good people.
- P3: The only rule I have in management is to ensure I have good people-real good people-and that I grow good people - and that I provide an environment in which good people can produce.

Indeed, this is a compelling testimonial on the importance of people in the software engineering process. And yet, all of us, from senior engineering vice presidents to the lowest practitioner, often take people for granted. Managers argue (as the group above had done) that people are primary, but their actions sometime are lying their words.

Players who can be categorized into one of five constituencies populate the software process (and every software project).

- 1.- Senior managers, who define the business issues that often, have significant influence on the project.
2. Project (technical) managers, who must plan, motivate, organize and control the practitioner who do software work.
3. Practitioners, who deliver the technical skills that, are necessary to engineer a product or application.
4. Customers, who specify the requirements for the software to be engineered.
5. End users, who interact with the software once it is released for production use.

The players noted above to be effective, the project team must be organized in a way that maximizes each person's skills and abilities populate every software project.

### 2.3.2 Team Leaders

Project management is a people intensive activity. They simply don't have the right mix of people skills.

What do we look for when we select someone to lead a software project?

**Motivation:** The ability to encourage (by "push or pull") technical people to produce to their best ability.

The ability to encourage people to create and feel creative when they work within bounds established for a particular software product or application.

Successful project leaders apply a problem solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone the team know (by words, and far more important, by actions) that quality counts and that it will not be compromised.

### 2.3.3 Problem Solving

An effective software project manager can diagnose the technical and organizational issue, most relevant, systematically structure a solution or properly motivate other professionals to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.

**Managerial identity :** A good project manager must take care of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

**Achievement :** To optimize the productivity of a project team, a manager must reward initiative and accomplishment, and demonstrate through his own action that controlled risk taking will not be punished.

### 2.3.4 Influence and Team Building

An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high stress situations.

## 2.4 THE SOFTWARE TEAM

The organization of the people directly involve in a new software project is within the project manager's purview

The following options are available for applying human resources to a project that will require  $x$  people working for  $y$  years.

1.  $x$  individuals are assigned to  $m$  different functional tasks, relatively little combined work occurs; coordination is the responsibility of a software manager who may have six other projects to be concerned with.
2.  $x$  individuals are assigned to  $m$  different tasks ( $m < n$ ) so that informal "teams" are established; an ad hoc team leader may be appointed; coordination among team is the responsibility of a software manager.
3.  $x$  Individuals are organized into  $t$  teams; each team is assigned one or more functional task; each team has a specific structure that is defined for all teams working on a project; coordination is controlled by both the team and a software project manager.

Although it is possible to voice pros and con arguments for each of the above approaches, there is a growing body of evidence that indicates a formal team organization (option 3) is most productive.

The "best" team structure depends on the management style of an organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Three generic team organizations are:

### 2.4.1 Democratic Decentralized (DD)

This software engineering team has no permanent leader. Rather, "task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks". Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.

### 2.4.2 Controlled Decentralized (CD)

This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but the team leader partitions implementation of solutions among subgroups. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

### 2.4.3 Controlled Centralized (CC)

Top-level problem solving and internal team coordination are managed by a team leader. Communication between the leader and team members is vertical.

Seven projects factors that should be considered when planning the structure of software engineering teams are :

- The difficulty of the problem is to be solved
- The size of the resultant program(s) in lines of code or function points
- The time the team will stay together (team lifetime)

- The degree to which the problem can be modularized
- The required quality and reliability of the system to be built
- The rigidity of the delivery date
- The degree of sociability (communication) required for the project

The length of time the team will "live together" affects team morale. It has been found that DD team structure result in high moral and job satisfaction and are therefore good for long lifetime teams.

The DD team structure is best applied to problems with relatively low modularity because of the higher volume of communication that is needed. When high modularity is possible (and people can do their own thing), the CC or CD structure will work well.

CC and CD teams have been found to produce fewer defects than DD teams, but these data have much to do with the specific quality assurance activities that are applied by the team. Decentralized teams generalized require more time to complete a project than a centralized structure and at the same time are best when high sociability is required.

Four "organizational paradigms" for software engineering teams are:

1. A closed paradigm structures a team along a traditional hierarchy of authority (similar to a CC team). Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. The random paradigm structures a team, loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggler when "orderly performance" is required.
3. The open paradigm attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when suing the random paradigm. Work is performed col-

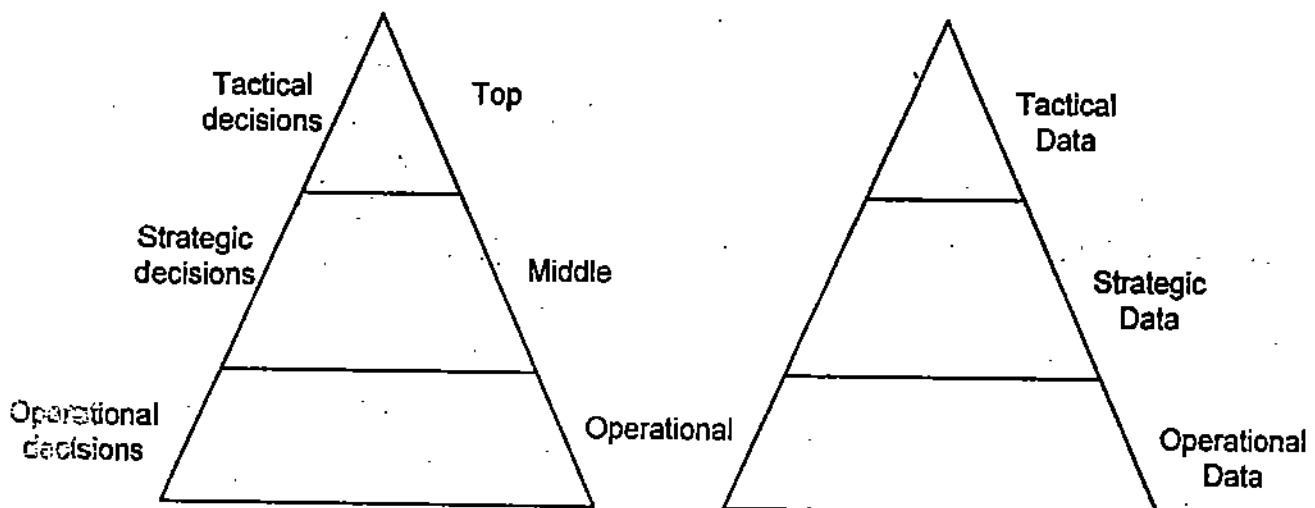


Figure 2.1 : Organizational Paradigms

laboratively with heavy communication and consensus-based decision making. Open paradigm team structures are well suited to the solution of complex problem, but may not perform as efficiently as other teams.

4. The synchronous paradigm relies on the natural compartmentalization of problem with little active communication among themselves.

---

## 2.5 ORGANISATION, INFORMATION AND DECISION

---

Successful development of information systems call for a deep understanding of the organisational structure and dynamics of the enterprise. Some organisations are goal oriented, the analyst must be clear as to what information exactly need to be collected, stored and analysed. Since every information must have a context, only operational information that ultimately has some decision making contribution must be collected. Second the information collected and processed must be consistent with the level of the organisation to which it is to be presented.

According to Anthony's classification, there are basically three levels of management, independent of the size of the enterprise: operational level, middle level and top level management. Operational decisions that call for large volumes of internal data (local to the enterprise). The middle management is concerned with medium range (tactical) decisions that call for much less information. The top management being concerned with long term (strategic) decisions calling for just a few vital internal information but a lot of external information as well. Any successful information system should take into account such a pattern of information needs by the management. This is generally pictorially displayed in the form of Management vs Information Pyramid.

The importance of information to management is further emphasized by the fact that much of management is primarily decision making. While there are several views of what constitute management, the generally accepted planning, organising, coordinating, directing and control are all concerned with decision making. In this text we take such a view of management and we perceive management information systems to support such managerial decision making. We also would like to emphasize that information systems should address clearly the situations of programmed decisions and non-programmed decisions by properly structuring the appropriate information. Failure to recognize intrinsic difference may led to a failure of the information system.

---

## 2.6 PROBLEM IDENTIFICATION

---

We have to reduced the initial, complex problem to a series of simple tasks, each of which can be solved fairly easily. Second, by working at several levels, we can move from an overall outline of a solution to various details, without having these details interfere with our overall understanding of the solution. Third, for large problems, we may be able to assign various parts of the overall solution to different people. Then, the overall structure will allow us to combine the various pieces into the final solution of the overall problem.

software engineering, each main section of the solution is called a module, and a module is a piece of a solution that performs a specific task or a collection of related operations.

2.6.1 Principles of Coupling, Cohesion, and information Hiding

Often there may be several reasonable ways to divide an overall solution into modules, and it may be difficult to determine which approach will work best. In any decomposition into modules, however, a few general principles should be

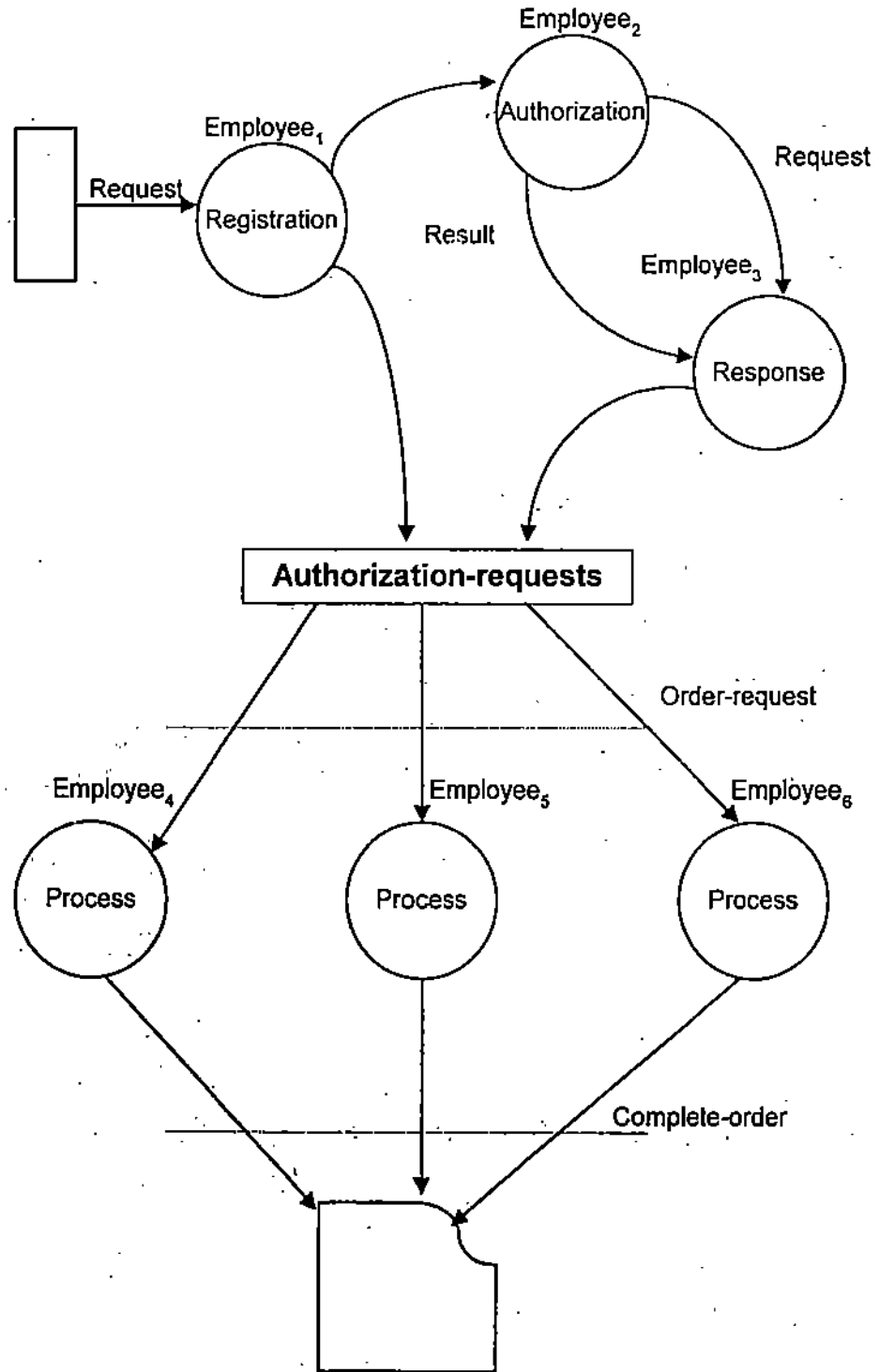


Figure 2.2 : Data flow diagram representing flow of work in an office

considered. First, we must try to keep the steps independent so that these steps do not interfere with each other. Each module should do a specific task, but we should try to limit the interaction among various modules. More formally, we can consider the coupling of modules in the problem design. Here coupling describes the amount that modules depend on each other, and the decomposition of large problems into pieces should minimize coupling.

In software engineering, the relating of various parts of a module to a central theme is called cohesion, and we should strive to write concise, cohesive modules for each part of a solution outline.

The third principle related to module decomposition involves the concept of abstraction, where we try to separate the details of a step from the way that step fits in the solution as a whole. Of course, at some point all details must be specified, but once they have been determined we do not want to worry about them when that task is performed in the overall solution. When we work with major steps in a solution, we should not be concerned about just how those steps are actually done. Rather, we concentrate on putting those individual tasks into an appropriate framework. This is the concept of information hiding, where the details of a task at one level are hidden from the use of that task at a higher level. From this standpoint, we normally should write modules for each levels of detail in a solution. For example, one main module often controls the overall running of a program, and that module calls on other modules to perform each major task.

Similarly, details about data storage, retrieval, and processing can be separated from the logical interactions involving that data. Information hiding, therefore, can be applied to both the structure of problems and the organization and manipulation of data.

Clearly, a program is correct and useful only if it helps to solve the problem at hand. The development of specifications, a general design, detailed algorithms, and programs, and each of these steps builds on the previous ones. Thus, any errors (such as omissions, inconsistencies) from one step normally will be reflected in subsequent work. If we can correct difficulties before we start to code, therefore we will not have to correct such trouble after the programs are written. More specifically, we should analyze and check each problem solving step for errors or potential problems.

### 2.6.2 Problem Handling Guidelines

- Software Specification.
- The Specification must be complete.
- We should not omit any cases.
- We must indicate what is required in each case.
- The specifications must state what is actually meant.
- Are the specifications contradictory?
- Do some cases overlap?
- Software Design.
- Broken out problem into logically.

- Independent pieces.
- The pieces must fit together.

---

## 2.7 SOFTWARE CRISIS

---

The transition of a familiarity with software into the development of useful application is not a straight forward task. This has led to the search for methods and techniques to be able to cope with the ever expanding demands for software. The present course, which is an attempt to teach the ingredients of a structured systems development methodology, and elsewhere in the programme there is a reference to the techniques of software engineering as well. Later on in the subsequent years of the MCA programme, you would also be exposed to a full course on Software Engineering.

However, it is still useful and desirable to have some feel for the kinds of problems which the programmer and the user faces and collectively perceive as the software crisis.

Software crisis can be broadly classified in the following major areas:

### 2.7.1 From Programmer's Point of View

The following types of problems may contribute in maximum cases to software crisis:

- Problem of compatibility.
- Problem of portability.
- Problem in documentation.
- Problem in coordination of work of different people where a team is initiating to develop software.
- Problems that arise during actual run time in the organisation. Some time the errors are not detected during sample run.
- Problem of piracy of software.
- Customers normally expand their specifications after program design and implementation has taken place.
- Problem of maintenance in proper manner.

### 2.7.2 From User's Point of View

There are many sources of problems that arise out of the user's end. Some of these are as follows:

- How to choose a software from total market availability ?
- How to ensure which software is compatible with his hardware specifications ?
- The customised software generally does not meet his total requirements.



- Problem of virus.
- Problem of software bugs, which comes to knowledge of customer after considerable data entry.
- Certain Softwares run only on specific operating system environment.
- The problem of compatibility for user may be because of different size and density of floppy diskettes.
- Problem in learning all the facilities provided by the software because companies give only selective information in manual.
- Certain software run and create files which expand their used memory spaces and create problem of disk management.
- Software crisis develops when system memory requirement of software is more than the existing requirements and/or availability.
- Problem of different versions of software.
- Security problem for protected data in software.

---

## 2.8 ROLE OF A SYSTEM ANALYST

---

### Who is Systems Analyst?

A systems analyst is a person who conducts a study, identifies activities and objectives and determines a procedure to achieve the objectives. Designing and implementing systems to suit organisational needs are the functions of the systems analyst. He plays a major role in seeing business benefit from Computer technology. The analyst is a person with unique skills. He uses these skills to coordinate the efforts of different type of persons in an organisation to achieve business goals.

### What a Systems Analyst does?

A system analyst carries out the following job:

- (a) The first and perhaps most difficult task of systems analyst is problem definition. Business problems are quite difficult to define. It is also true that problems cannot be solved until they are precisely, and clearly defined.
- (b) Initially a systems analyst does not know how to solve a specific problem. He must consult with managers, users and other data processing professionals in defining problems and developing solutions. He uses various methods for data gathering to get the correct solution of a problem.
- (c) Having gathered the data relating to a problem, the systems analyst analyses them and thinks of plan to solve it. He may not come up personally with the best way of solving a problem but pulls together other people's ideas and refines them until a workable solution is achieved.
- (d) Systems analysts coordinate the process of developing solutions. Since many problems have number of solutions, the systems analyst must evaluate the merit of such proposed solution before recommending one to the management.

- (e) Systems analysts are often referred to as planners. A key part of the systems analyst's job is to develop a plan to meet the management's objectives.
- (f) When the plan has been accepted, systems analyst is responsible for designing it so that management's goal could be achieved. Systems design is a time consuming, complex and precise task.
- (g) Systems must be thoroughly tested. The systems analyst often coordinates the testing procedures and helps in deciding whether or not the new system is meeting standards established in the planning phase.

#### Attributes of an effective Systems Analyst

Systems analyst must have the following attributes:

- (a) **Knowledge of people:** Since a systems analyst works with others so closely, he or she must understand their needs and what motivates them to develop systems properly.
- (b) **Knowledge of Business functions:** A systems analyst must know the environment in which he or she works. He must be aware of the peculiarities of management and the users at his installation and realize how they react to systems Analyst A working knowledge of accounting and marketing principles is a must since so many systems are built around these two areas. He must be familiar with his company's product and services and management's policies in areas concerning him.

#### Check Your Progress

1. A help system in an application program is used to:
  - (a) Make it easy to switch from one mode to another.
  - (b) Display menus to prompt the users with choices of available commands.
  - (c) Display Explanatory Information.
  - (d) All of the above
2. An Expert System
  - (a) simulates the reasoning of a Human Expert in a Particular Subject.
  - (b) is an application of Artificial Intelligence research.
  - (c) both (a) and (b)
  - (d) None of the above.
3. Vertical Market Application Programs Include
  - (a) Database Management Systems.
  - (b) Farm Management Programs
  - (c) Home Finance Programs
  - (d) All of the above.
4. An example of an expert system is
  - (a) The Internist, a medical diagnosis program.

- (b) A stock and bond Analysis program.
- (c) A Structural Analysis program
- (d) All of the above

5. A simulation Program

- (a) guides novices through the basics of using other computer programs.
- (b) teaches fact, such as arithmetic operations and spelling.
- (c) teaches by emulating the responses of the system being studied.
- (d) None of the above.

---

## 2.9 MODEL ANSWERS

---

Q1 - (c)

Q2 - (c)

Q3 - (b)

Q4 - (a)

Q5 - (c)

---

## **UNIT 3 PROJECT PLANNING AND CONTROL**

---

### **Structure**

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Project Planning and Control
- 3.3 Project Scheduling
- 3.4 Project Standards
  - 3.4.1 Project Conflicts
  - 3.4.2 Project Modifications
  - 3.4.3 Completing the Project
- 3.5 Project Outsourcing
- 3.6 Model Answers

---

### **3.0 INTRODUCTION**

---

The important aspects of Project Management are discussed in this Unit. An effective Manager is essential for successful project execution. It is important when organizing a project to ensure that every person knows his or her role in the project and is aware of corporate objectives. The Charting Techniques are discussed under Project Scheduling heading. A schedule has two primary functions, it is both a plan and a device for measuring progress. This Unit deals with Project Management concepts like project standards, conflicts, modifications, project outsourcing and completing the project.

---

### **3.1 OBJECTIVE**

---

After completing this Unit, you should be

- able to describe Important Issues of Project Management,
- describe project standards, conflicts and factors for Project Outsourcing, and
- describing the Task for the Software Project, Refinement of Major Tasks and the Project Plan.

---

### **3.2 PROJECT PLANNING AND CONTROL**

---

The planning, design, and installation of a system termed a project and is directed by a project leader who uses the available resources to produce a new and better system for the organizations.

In large companies, the installation of a computer system may take years and involve thousands of people. Planning for smaller projects also requires effective management controls to ensure the desired results. Thus, project planning for any company has four main steps.

1. Organizing the resources available for the project.
2. Scheduling the events in the project.

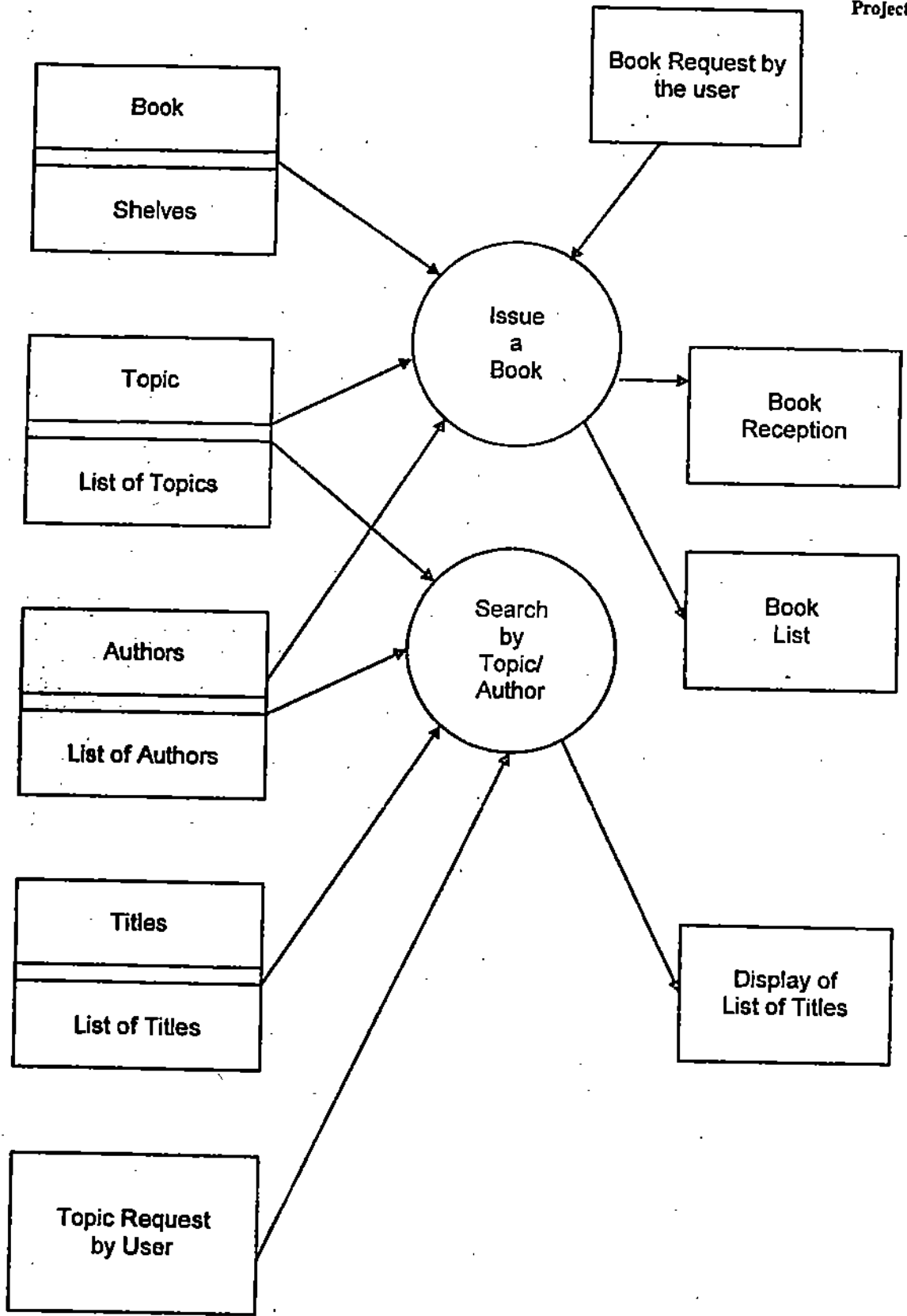


Figure 3.1 : Data Flow of Simplified Library Information System

3. Evaluating progress.
4. Establishing standards for the project.

An effective manager is essential for successful project planning. The techniques of project planning are not a substitute for good management, but merely a tool to be used by managers to achieve better results. Only effective management can complete the project on time, within budget, and with satisfactory results.

To achieve a goal, that goal must be kept clearly in mind; consequently, defining the objectives is the first action taken in any project. Along with defining objectives, corporate management must assign priorities to the various projects underway and clarify the relationship between systems projects and existing systems. A systems project requires extensive interaction between systems personnel and people in the user areas. User are, of course, preoccupied with day-to-day operations, and it cannot be assumed they will be enthusiastic about participating in a system study. Only when corporate management clearly defines the importance of user participation in systems development will be the necessary cooperation exist.

Many systems are designed and implemented through project teams headed by a project leader. The team may be relatively small during the feasibility stage—consisting of a few highly qualified systems people, users, and managers. During the design phase, when more detailed work is required, the size of the team normally is increased. A typical project team would have a senior systems analyst as project leader, supported by junior analysts, programmers and whatever key user personnel are required. When the project has been completed, the team is disbanded and each member is reassigned to a new project or returned to normal work.

To organize a project, the project leader must determine who is required for the project, when they are available, and for how long their services can be expected. The key people required in a systems project are often the key people in the day-to-day operations of an organization, and they probably will have to continue their normal routines as they participate in the systems project. In organizing their efforts, the project leader must avoid scheduling important project activities when the users are very busy with normal duties. For users who are "always busy", plans must be made to utilize overtime, shift personnel, or and personnel to free the key systems users for participation in the project.

It is important when organizing a project to ensure that every person knows his or her role in the project and is aware of corporate objectives. This is accomplished through formal training as well as informal conversations.

The project leader is solely responsible for the completion of a project, but obviously cannot do it alone. As the project is organized, responsibility is assigned and authority delegated for the completion of each phase of the project. Responsibilities must be defined precisely, and overlapping responsibilities avoided. When a phase of a project breaks down, is behind schedule, or is over budget, the leader of a well-organized project will be able to identify easily the responsible person who can provide information and, perhaps, the solution to the problem.

Besides organizing people, the project leader must budget money and order equipment. Acquainting people with their responsibilities and enabling them to discharge these responsibilities is the essence of organizing.

---

### 3.3 PROJECT SCHEDULING

---

The charting techniques are the scheduling tools of the project planner. Even the simplest project should be charted so that progress can be measured. The Gantt chart is effective in simple projects, especially when the interrelationships among events are not too complex. Complicated scheduling usually requires a PERT chart.

Included in the tasks to be scheduled in a normal data processing project are systems design, programming, file and data base creation, program and systems testing, conversion, documentation, and training. The project planner must anticipate problem areas that inevitably develop and allow for delays in obtaining approvals at key check-points in the project.

Projects are organized into modules, or segments, of related tasks. Modular planning has these advantages: it facilitates assigning responsibility and measuring process, and it further allows systems analysis to work in concentrated areas of projects so they can master every aspect of that portion of the system.

A schedule must be flexible because unexpected events occur that may alter materially the development of the system. Seldom do systems projects meet the original schedule at each milestone. This does not imply that schedules are made to be broken, but a schedule cannot be so rigid that when the unexpected occurs, subsequent events cannot be rescheduled.

A schedule has two primary functions, it is both a plan and a device for measuring progress. The key steps in a schedule are called milestones, or checkpoints. As the project progress, the date each milestone is completed is compared with the date for which it was projected. In any project, frequent progress reviews take place in which the status of events is reported and evaluated. If, in the original planning stage, the important milestones were anticipated correctly, reporting them as completed, late, ahead of schedule, or on time has significance to the status of the project. The status of fringe events is relatively insignificant. Here the value of the PERT network as a tool for determining the relative importance of milestones is apparent.

Status of projects is often reported in terms of percentage of completion. As a simplified reporting device, this is effective and allows easy communication with top management. The problem with percentage of completion reporting is that events on the critical path are not emphasized. A project may have 90 percent of its events complete, but if one of the incomplete events is on the critical path and is two years late, the project is in serious trouble.

Accurate scheduling requires extensive experience. The novice scheduler almost always does not allow enough time for activities. Even when estimates are carefully gathered, as in the preparation of a PERT network, some areas of delay are not apparent. For example, an inexperienced person may not realize that equipment or forms are often delivered late. Moreover, lead time must be provided for approvals in several areas, such as file design and input and output forms.

At the outset of the project, the project leader must determine the reporting format. Its status to be measured in days, weeks, and tenths of weeks, or percentage of each

job done? When are status reports to be made? Are reports to be made orally, in writing, or in chart form?

When a project is behind schedule, corrective steps must be taken. Establishing milestones is meaningless unless the project manager can enforce adherence to schedule. Enforcement is a normal managerial duty. If a project leader cannot enforce a schedule, someone else should be leading the project. If one area is consistently behind schedule, or over budget, the project leader must discuss the problem with the individual responsible and take corrective action. A variety of options are open to the project leader.

- Increase the budget
- Increase manpower in the form of overtime or additional people
- Add equipment
- Change priorities
- Replace the individual responsible.

The project leader must determine the real cause of unsatisfactory progress. Perhaps it is a budgetary or personnel problem. Too often, a major cause of apparently unsatisfactory performance is that the original schedule estimates were wrong, and that progress is as good as can be expected under the circumstances.

Projects have many target dates, but few deadlines. The project leader must distinguish one from the other. When target dates are missed, there may be some grumbling, but missed deadlines result in financial loss to the organization.

At the outset of a project, the project leader should not become committed to unreasonable target dates or deadlines. Unreasonable deadlines are costly because unnecessary effort is made to meet them. Moreover, failing to meet them often creates morale problems.

The project leader must remember that this is not the only project under way and delays will occur routinely simply because another, more important project may have to be tested first. Schedules are highly dependent upon priorities and should be planned accordingly.

**The Problem of Capacity:** The problem of capacity occurs when a system component is not large enough. Capacity problems are specially common in organisations that experience peak periods of business. During peak periods, inadequate processing capacity, transmission capacity, storage capacity, staff capacity, and the like may all exist. Capacity problems are also evident in rapidly growing organisations. With growth, smaller-capacity equipment soon becomes too small; smaller staff groups soon become overworked. In either case, some expansion is needed to handle the increasing volume of business.

Many system problems are directed at solving capacity problems. Because it is often difficult to justify the purchase of new equipment or the hiring of new staff, people tend to put off such decisions until the very last moment. Consequently, when the systems group is contacted, the problem of capacity is easy to spot; the difficulty, however, lies in knowing how to handle the problem. For example, an analyst might be forced to suggest a short-term solution to the problem. This is done to gain time toward the formulation of a longer-term solution. For instance, an analyst might recommend: "Let's hire five part-time employees to help us get through the peak period." When a short-term approach fails, the analyst may be tempted to implement a quick-fix computer-based solution. Unfortunately, this solution carries with it the associated danger of creating an even more severe system problem in the near future.



**The Problem of Throughput:** The problem of Throughput may be viewed as the reverse of the problem of capacity. Throughput deals with the efficiency of a system. If system capacity is high and production low, a problem of throughput occurs. Consider the following example.

Five programmers are assigned to a fairly straightforward programming assignment consisting of 10,000 lines of computer code. After thirty days of coding, the programming team is evaluated. It is discovered that they have completed 6000 usable lines of code. Now, if each programmer worked eight hours a day, a total of 1200 hours would have been expended on the project. Calculated differently, the average production rate for each programmer would be 5 lines of code per hour (6000 lines divided by 1200 hours). These findings might lead the analyst to conclude that there is a problem of throughput.

Similar to the problem of capacity, the problem of throughput may be much easier to spot than to treat. When repeated equipment breakdowns lead to low rates of production (and when the equipment has been purchased and cannot be returned), an organisation can badger the vendor into fixing the equipment but can achieve little more short of legal action. Likewise, when groups of people exhibit low rates of production, such as the five-person programming team, the problem becomes even more complicated. Badgering and threats may not work at all. Rather, a manager must be able to determine the root of the problem for any improvement in throughput.

#### Evaluating the Problem

Suppose that a problem has been identified. The next step is problem evaluation, which consists of asking the following questions: Why is it important to solve the problem? What are possible solutions to the problem? What types of benefits can be expected once the problem is solved? There will be times when an analyst will recommend that no project be started to resolve a problem, as the next example demonstrates.

Suppose that an analyst discovers that the real problem lies with the supervisor of an area. Because of mistakes made by this man, the throughput rate is 20 percent less than had been expected. However, suppose next that the supervisor is new to the job, is smart enough to realize where mistakes were made, and knows how not to repeat them in the future. Given this situation, the analyst might close the book on this project, recommending that no action be taken at this time.

Consider a different set of circumstances. Suppose that an analyst determines that a problem of low throughput can be traced to a computer printer. Suppose further that the problem must be corrected. Once the problem has been identified, the analyst would prepare a solutions table to list possible problem solutions and the expected benefits from each. Sometimes, the best solution is not at all evident. The analyst might recommend that further study is required to determine which of the possible solutions is best.

In this section, we have spent considerably more time examining how an analyst identifies a problem compared with how the problem is evaluated. This uneven split also occurs in practice. As a general rule, analysts spend 75 percent of the project-definition phase of analysis defining the problem and 25 percent evaluating and documenting their findings. Note also that we have limited our discussion to seven major types of system problems. Because of this limitation, you might ask, "What about the problems of communication? of group conflict? of management? of system security? Are these problems as well? Are these types of problems also evaluated by the analyst?" Although our discussion has been restricted to more technical system problems, individual or group problems also occur in a systems

environment and require identification and evaluation.

Still another limitation is the coverage given to determining the feasibility of taking some action to solve a problem. The concept of feasibility entails the joint questions of "Can something be done?" and, if so, "Should it be done given a particular set of circumstances?" For example, is it possible to climb a mountain when we have at our disposal only a forty - foot rope? If it is, a second question is well advised, namely, "Should we attempt such a climb given the size of our rope?" We will examine the question of project feasibility in more detail in the next unit of this block.

A final limitation is the coverage given to tools which the analyst can use to identify and evaluate system problems. These tools are needed when the problems are not self-evident.

Organisations face various types of problems during their course of operations and come across opportunities or situations which could be converted into profitable

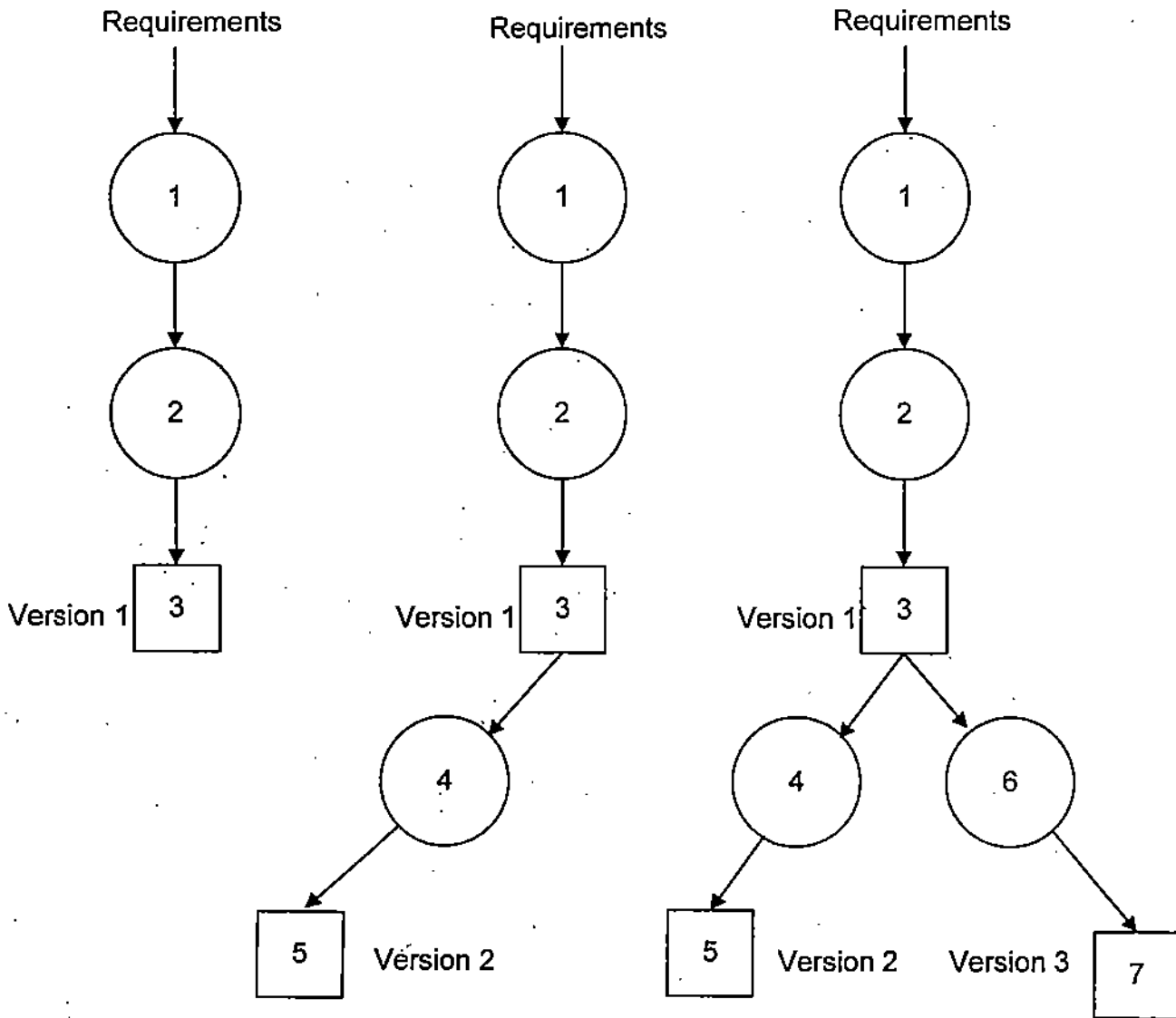


Figure 3.2 : Sequential Design of a Program Family

solutions. Whenever there is an opportunity and /or problem in the existing system of operations or when a system is being developed for the first time, the organisation considers designing a new system for information processing.

### Sources of Problem/Opportunity

Organisations usually face problems or have opportunity due to the following:

- a new product or plant or branch
- a new market or new process
- failure of an existing system
- inefficiency of an existing system
- structural error in the existing system, etc.

Thus a thorough analysis of the situation need to be required. Not only the above listed reasons but there exist some organisation based reasons too.

### Problem Identification and Definition

For identifying problems/opportunities, we scan the following:

- The performance of the system
- The information being supplied and its form
- The economy of processing
- The control of the information processing
- Security of Data and Software.

---

## 3.4 PROJECT STANDARDS

---

Initially, the project must establish the objectives of each phase of the project. Each phase must be of a controllable size, and every task within the phase spelled out. The project leader and the individual responsible for the phase must agree upon the human skills and other resources required to accomplish each task. They also must agree upon the expected outputs from these tasks. Ultimately, they must decide upon the measurable outputs that will be examined throughout the project to evaluate progress.

For each phase of a project, the status of time to complete tasks, personnel utilization, and unforeseen problems should be reported to the project leader. This report is then reviewed by the project leader and the managers concerned to evaluate progress. Also evaluated is the quality of work, as reflected in the outputs from each phase. This periodic review has four main tasks:

1. Review project progress.
2. Analyze the impact of delays on the entire project.
3. Examine any problems existing in the quality of the data.
4. Anticipate developing problems.

### 3.4.1 Projects Conflicts

Disagreements are inevitable in most systems projects. Lack of clearly defined

objectives and standards for acceptable performance are major causes of disagreement. A project leader should be sure to have these definitions in writing before undertaking the project.

The project manager is responsible for settling disputes that arise within the scope of the project. This is usually done by calling together all concerned parties of the talking the matter out. When the parties cannot reach agreement, the project manager must intervene.

Implied in this procedure is that all the parties are aware of what authority the project leader has. Too often, this may be vague. In general, however, the project manager must have the backing of whoever originally requested the project. When top management is the requester, the authority of the project leader is rarely questioned. When the project has been requested by a specific department, branch, or individuals representing areas of equal or greater status than the project leader, substantial challenges to the project leader's authority may occur. Any irony regarding systems projects is that large projects are usually more successful than smaller ones because the requester comes from higher up in the corporate structure.

### 3.4.2 Project Modifications

For a variety of reasons, changes must sometimes be made in a project while it is under way. Requests for changes must be evaluated carefully, according to several criteria.

1. The impact on the present schedule.
2. The impact on the resources available for the project.
3. The cost
4. The effect on the deadlines for the system

Poor planning is the primary cause for changes in current projects. It has four other consequences as well

**Obsolescence** : The systems in an organization tend to become obsolete quickly.

**Poor Follow up** : The responsibility for follow-up falls upon corporate executives instead of project leaders.

**Inflexibility** : Systems become inflexible, so that minor modifications result in extensive program writing.

**Lack of Documentation** : Procedure writing and documentation are, in general, neglected.

### 3.4.3 Completing the Project

Projects often run on schedule and within budget for most of their existence, only to fall behind during the final stages. Perhaps the last 10 percent of a project is the most difficult to complete because enthusiasms wanes at that point and people look forward to newer challenges. Often, too, the final phases of a project are the least interesting. Documentation must be updated and completed, detailed problems solved and annoyances cleared up. Discipline on the part of the project team is required to do the final stages of a project in a professional manner and turn over a complete and functioning system to the user staff.

## 3.5 PROJECT OUTSOURCING

Sooner or later, every company that develops computer software asks a fundamental question: "Is there a way that we can get the software and systems that we need at a lower price?" The answer to this question is not a simple one, and the emotional discussions that occur in response to the question always lead to a single word: "outsourcing".

In concept, outsourcing is extremely simple. Software engineering activities are contracted to a third party who does the work at lower cost, and hopefully, higher quality. Software work conducted within a company is reduced to a contract management activity.

If you have been brought up in a culture that glorifies the American software industry as a world leader, I ask simply that you remember that it was only a few years ago that we had the same opinion of our automobile industry... [Software development may well move out of the U.S. into software factories in a dozen countries whose people are well educated, less expensive, and more passionately devoted to quality and productivity. A strong statement! But one that is already becoming a reality.

The decision to outsource can be either strategic or tactical. At the strategic level, business managers consider whether a significant portion of all software work can be contracted to others. At the tactical level, a project manager determines whether part or all of a project can be best accomplished by subcontracting some portion of the software work.

Regardless of the breadth of focus, the outsourcing decision is often a financial one.

On the positive side, cost savings can usually be achieved by reducing the number of software people and the facilities (e.g., computers, infrastructure) that support them. On the negative side, a company loses some control over the software that it needs. Since software is a technology that differentiates its systems, services, and products, a company runs the risk of putting the fate of its competitiveness into the hands of a third party.

The trend toward outsourcing will undoubtedly continue. The only way to blunt the trend is to recognize that software work in the twenty-first century will be extremely competitive at all levels. The only way to survive is to become as competitive as the outsourcing vendors themselves.

### Check Your Progress

1. Which of the following languages is not well suited for computation?
  - (a) Java
  - (b) C++
  - (c) C
  - (d) COBOL
2. Repeated Execution of Simple Computation may cause compounding of -
  - (a) Round off Errors
  - (b) Syntax Errors

- (c) Run Time Errors
  - (d) Logic Error
3. Use of Modern Control Technology in automation systems.
- (a) Reduces cost
  - (b) Increases yield
  - (c) Improves Reliability
  - (d) All of the above
4. The Preliminary evaluation of a Top down design before programs are written is referred to as an:
- (a) Informal Design Review
  - (b) Structured Walk through
  - (c) Formal Design Review
  - (d) Scheduled Review

---

### 3.6 MODEL ANSWERS

---

- Q.1 - (d)
- Q.2 - (a)
- Q.3 - (d)
- Q.4 - (a)

---

## UNIT 4 RISK MANAGEMENT CONCEPTS

---

### Structure

- 4.0 Introduction
- 4.1 Objective
- 4.2 Introduction and Risk Management Concepts
  - 4.2.1 Managing Risk
  - 4.2.2 Typical Management Risk in Software Engineering
  - 4.2.3 Technical Planning
  - 4.2.4 Project Tracking
  - 4.2.5 Delivery Timings
  - 4.2.6 Partial Recovery
- 4.3 Benchmark testing
- 4.4 Model Answers
- 4.5 Further Readings

---

## 4.0 INTRODUCTION

---

Risk Analysis and control is a crucial factor for any Software project and it is a topic of management. Several standard techniques for identifying project risk, assessing their impact, monitoring and controlling them are discussed in this Unit. This Unit is dedicated to Risk Managing Concepts and Tools. This Unit discusses typical risks in software developing process, technical planning, project tracking, delivery timings and partial recovery. Software performance and the benchmarked versions are discussed in this Unit.

---

## 4.1 OBJECTIVE

---

After completing this Unit, you should be able to:

- Describe Risk Management Concepts and strategies to manage such Risks.
- What is Risk Monitoring, Technology Risk, Risk Components and Drivers, Customer Related Risks.
- Knowledge of Basic Concepts of Project Scheduling and Tracking.
- Knowledge of Software Testing Fundamentals.

---

## 4.2 INTRODUCTION AND RISK MANAGEMENT CONCEPTS

---

Experimental assessment of different organizational structures is difficult. It is clearly impractical to run large software development projects using two different types of organization, just for the purpose of comparing the effectiveness of the two structures. While cost estimation models can be assessed on the basis of how well they predict actual software costs, an organizational structure cannot be assessed so easily, because one cannot compare the results achieved with those one would have achieved with a different organization.

Experiment have been run to measure the effects of such things as teams size and

task complexity on the effectiveness of development teams. In the choice of team organization however, it appears that we must be content with the following general considerations:

Just as no life cycle model is appropriate for all projects, no team organization is appropriate for all tasks.

Decentralized control is best when communication among engineers is necessary for achieving a good solution.

Centralized control is best when speed of development is the most important goal and the problem is well understood.

An appropriate organization tries to limit the amount of communication to what is necessary for achieving project-goals no more and no less.

An appropriate organization may have to take into account goals other than speed of development. Among these other important goals are: lower life cycle costs, reduced personnel turnover, repeatability of the process, development of junior engineers into senior engineers and widespread dissemination of specialized knowledge and expertise among personnel.

#### 4.2.1 Managing Risk

An engineering project is expected to produce a reliable product, within a limited time, using limited resources. Any project, however runs the risk of not producing the desired product, overspending its allotted resource budget, or overrunning its allotted time. Risk accompanies any human activity.

Risk analysis and control is a topic of management theory. Several standard techniques exist for identifying project risk, assessing their impact, and monitoring and controlling them. Knowledge of these technique allows a project manager to apply them when necessary to increase the chance of success of a project.

We have already seen in previous sections, many examples of software development problems that can be viewed from a risk analysis point of view. For example, we have discussed the difficulties of specifying product requirements completely. Given these difficulties, a project runs the risk of producing the wrong product or having the requirements change during development. An effective approach for reducing this risk is prototyping or incremental delivery. A different type of approach to handling the risk of late changes in the requirements is to produce a modular design so that such changes can be accommodated by actual changes to the software. Of these approaches, prototyping tries to minimize changes in the requirements, while modular design tries to minimize the impact of changes in the requirements. choosing between the two alternatives, or deciding to use both, should involve a conscious and systematic analysis of the possible risk, their likelihood, and their impact.

At different levels of an organization, different levels of risk can be tolerated. For example, a project manager at the beginning of his career may not want to tolerate any delay in the schedule (to minimize risk to his personal career), while his supervisor might be more concerned with the reliability of the product. A project manager might not be able to tolerate the risk of running over budget by more than 10% while a higher level manager who is more aware of the value of early time to market, and therefore more concerned with the time taken by the project might be willing to overspend the budget by much more if the product can be produced sooner. To complicate matters even more, different people have different tolerances of risk based



on their personal nature, as can be evidenced by observing people at a gambling casino.

#### 4.2.2 Typical Management Risks in Software Engineering

By examining the difficulties that raise in software engineering, we can identify typical areas of risk that a software engineering project manager must address. We have already discussed the example of changes in requirements. Another important risk is in not having the right people working on the project. Since there is great variability among the abilities of software engineers, it makes a big difference whether a project is staffed with capable or mediocre engineers. If key positions in a project are staffed with inappropriate people, the project runs the risk of delaying deliveries or producing poor quality products, or both.

Risk Details	Risk Management Techniques
(a) Individual shortfalls	Staffing with top talent; job matching; teambuilding; key-personnel agreements; cross-training; pre-scheduling key people.
(b) Unrealistic schedules and budgets	Detailed multisource cost and schedule estimation; design to cost; incremental development; software reuses, requirements scrubbing.
(c) Developing the wrong software functions	Organization analysis; mission analysis; concept formulation; user surveys; prototyping early user's manuals.
d) Developing the wrong user interface	Prototyping; scenarios; tasks analysis; user characterization (functionally, style workload)
e) Gold plating	Requirements scrubbing; prototyping; cost-benefit analysis; design to cost.
f) Continuing stream of requirements changes	High change threshold; information hiding; incremental development (defer changes to later increments)
g) Shortfalls in externally furnished components	Benchmarking; inspections; reference checking; compatibility analysis.
h) Shortfalls in externally performed task	Reference checking; per-ward audits; award-fee contract; competitive design or prototyping; teambuilding
i) Real-time performance shortfalls	Simulation; benchmarking; modeling; prototyping instrumentation; tuning
j) Computer science capabilities	Technology analysis; cost benefit analysis prototyping; reference checking.

schedule overrun risks can be reduced by limiting dependencies among tasks. For example, if many tasks cannot be started until a given task is completed, delay in that one task can delay the entire project. Imagine a computer system project where hardware and software are being developed concurrently. If all software development is scheduled to start after the hardware is completed, any delay in completion of the hardware translates directly into a delay in the entire project. A

way control this schedule risk is to produce a simulation version of the hardware so that software development can be carried on even if the hardware is delayed.

PERT charts can help a manager identify schedule bottlenecks immediately-even mechanically: a node with many outgoing arcs is a sign of trouble, and the manager should try to reschedule activities to avoid it. Such a node should be rescheduled especially if it happens to be on the critical path. One way to reschedule the activity is to break it up into smaller activities. Examining the risk items in column I of the table, we can see that they overlap the items that are used in software cost estimation models. If a factor has a high multiplier in cost estimation, it represents a risk that must be managed carefully.

While we have only talked about management in the large, that is, management of a group of engineers that must cooperate to produce a common product, many techniques we have discussed can be used by the individual engineer as well, that is in the small. Indeed, each engineer must carefully plan, monitor, and execute the plan for his or her own work. Staffing and directing are the only two management function that PERT chart can help individual engineers on nontrivial activities.

While we have discussed the difficulties in measuring software productivity, we have also stated the importance of defining and collection such metrics. A software existing projects, and validate the metrics in order to be able to apply management principles to guide the planning, decision making, and monitoring of future project. In the absence of metrics, there is no way to measure whether progress is being made and, if so, at what rate.

In addition to the technical aspects of management that we have discussed, the manager is involved in resolving conflicts among competing goals. For example:

In assigning tasks to people, should the experienced engineer be assigned to do all the difficult jobs and get them done fast, or should S/he work with less experience engineers to have them trained for the further?

Large software system exhibit what have been called progressive and antiregressively components in their evolution. A software evolving progressively when features are being added and functionality is increasing. But after adding to the software for a long time, its structure becomes so difficult to deal with, that an effort must be undertaken to restructure it to make it possible to make further additions later. Reengineering, which does not add any functionality, is an antiregressive component of software because its goal is to stop the software from regressing beyond hope. The decision that the manager must make is when it is time to undertake antiregressive activities. In its logical extreme, this decision amounts to whether a software system must be retired and a new one developed.

A common conflict is known as the mythical man-month conflict. In some disciplines, people and time are interchangeable, that is, the same task can be accomplished by two people in half the time that it takes a single person. In software engineering, as we have seen adding more people increase the overhead of communication on each engineer, preventing a linear increase in productivity with additional people. In fact, after a certain point in the project, and beyond a certain number of people adding more people to the project and delay the project rather than speed it up. The difficult task of the manager is to determine when those limits have been reached. The cost estimation models that we have discussed are the beginning of foundation for allowing such decision to be made quantitatively.

Will the approached that worked on one project work on another project? One of the painfully observed phenomena in software engineering is that many techniques do not scale up; that is, a method that works on smaller projects does not necessarily work on large projects. Our emphasis throughout the earlier or design, was in fact

motivated by this inability to scale up in software engineering. A corollary of this observation is that it is not in general possible to derive precise scheduling information from a throwaway prototype. For example, if the prototype demonstrates a tenth of the functionality of the final product, the product will not take ten times the development time that the prototypes took.

Should engineers be encouraged to reuse existing software in order to reduce schedule time? While software reuse reduces coding time, it may cause difficulties in other phases of the life cycle. If the modules that are reused do not supply the exact interfaces suitable to the design and functionality of the product, they cause the engineer to go through extra effort just to match the interface, and worse, they lessen the evolvability of the product. These problems point out the immaturity of software reuse technology, rather than an inherent flaw in the idea of software reuse. Whatever the reasons, however, the manager is left with a difficult set of compromises to consider.

Finally, we must recognize that there are no panaceas to software engineering problems. For example, using the latest process—an incremental, prototype-oriented, life cycle model—or the latest technology—an advanced tool for configuration management—or the latest methodology—object oriented analysis and designing—will not solve all software production problems. In truth, software engineering is a difficult intellectual activity, and there are no easy solutions to difficult problems. Using the right process, the right technology, the right methodology, and the right tool will certainly help to control the complexities of software engineering, but it will not eliminate them altogether. In practice, because software engineering is such a difficult task at times, and because costs are rising rapidly, managers tend to grasp at any solution that comes along which promises to solve their very real problems.

Panaceas do not exist, however, and a manager is best advised to accept the difficulties of the job and carefully evaluate the impact of any newly offered proposed panaceas.

### 4.2.3 Technical Planning

At the start of the project, it would have been wise to write clear and precise documents stating the requirements for the new product. These documents should have been based on a careful and organized interaction with potential users, paying close attention to choosing a representative sample of the user population.

One might argue that it would have been difficult, or even impossible, to write such documents, since the desirable features of the system were not clear in the first place. A possible solution to this problem would have been to put a limited effort into the development of a system that would act as a prototype. The prototype system would then help assess the most critical issues and derive firm requirements by observing user's reactions when using the system. Unfortunately, the designers did not even realize that a problem existed, and therefore, they did not even consciously choose between the first alternative—specifying requirements carefully—and the second—developing a fast exploratory prototype.

Similarly, careful planning of resource allocation should have started, both from the point of view of work assignment to designers and programmers and from the point of view of physical resources management (e.g., hardware acquisition and office space). But instead, just the opposite happened. An amusing but dangerous "Game" started between the designers and the very few representative clients. In this case, of course, the designers included the company leaders, since they were technical people and they had originated the idea of the product in the first place.

In fact, everybody was excited with the innovative and challenging features of the

product, but nobody paid much attention to fairly obvious but critical details. For instance a true programming language was designed to allow the sophisticated user to define his or her own document composition rules. Very sophisticated and expensive word-processing facilities were included without measuring their cost effectiveness.

For a long time, nobody paid attention to the definition of suitable user interfaces to facilitate the interaction of nontechnical people—a lawyer or secretary—with the system. Similarly, sophisticated features for the automatic computation of invoices on the basis input data (people time, service value, travel expenses, etc.), were designed, but no attention was paid standard office operations such as the filing of large numbers of documents (e.g., records of automobile sales in some offices, several hundreds of such documents are produced every day).

From a technical view point, many typical mistakes were made:

No analysis was performed to determine whether all the product features were needed by all users; or whether it would be better to restructure the functionality of the product based on different classes of users. More generally, no effort was put into determining which qualities of the product were not critical for its success. For instance, in the choice of the hardware and of the development software (the operating system, programming language, etc.) little or no attention was paid to the evolution, and no effort was made to prepare for possible changes to them.

No "Design for change" was done, i.e., no design decision was influenced by any analysis of which parts of the product were likely to change during the product's lifetime (e.g., how might possible changes in the law affect product requirements?)

Strong pressure was applied to have some (any) code running as early as possible.

No precautions were taken to minimize damages due to personnel turnover.

What is perhaps worth pointing out is that everybody in the company was, of course aware of such classical mistakes in software engineering. This awareness notwithstanding, the mistakes were made. This remark shows that knowing the difficulties is not enough: it is also necessary to have the technical and organization ability and willingness to face them, even at the cost of doing something that does not appear immediately attractive and productive.

#### 4.2.4 Project Tracking

After a while (about six months after the start of development) some mistakes became apparent both from a technical and from a management point of view. For instance, the lack of a clear definition of the product's functionality caused some initial misunderstanding between the potential users—the ones with whom the early contacts have been established—and the designers. It was realized that some features that have been neglected at the beginning were actually quite important.

Also, getting in touch with other potential users showed that not all of them needed the same features. Thus, a modular architecture would have been preferable, even from the user point of view, allowing the product to be customized for different classes of users, just as the same "skeleton" of a car can be sold with different computer can be sold with many optional features e.g., or the same personal can be sold with many option features etc., a color monitor or a floating point coprocessor.

Finally, it became apparent that the original cost estimates were off by an order of magnitude. This invalidated the initial economic and financial plans.

The reaction of this situation was even worse than the problem itself: the impact of the mistakes-both technical and non-technical-was again underestimated. In general, the attitude was of the following type "OK, we made a few mistake but now we are almost done. So let's put in a little more effort, and we will complete the product soon and will start earning money. " That is, no critical and careful analysis of the mistakes was made, nor was a serious re-planning and redesign of the whole project attempted. There was only a generic claim of an intuitive confidence is being close to the end.

The consequences of this attitude were disastrous. Under the pressure of "being almost done and close to delivering the product," the design focused more and more on the very end product i.e., machine code. Classical "patches" on object code were made wildly, no systematic error and correction logs were kept, and communications among designers occurred almost exclusively orally in an attempt to save time.

The same attitude prevailed on the managerial and financial front: since "we are almost done" "we just need a little more financing and can accept almost any terms.

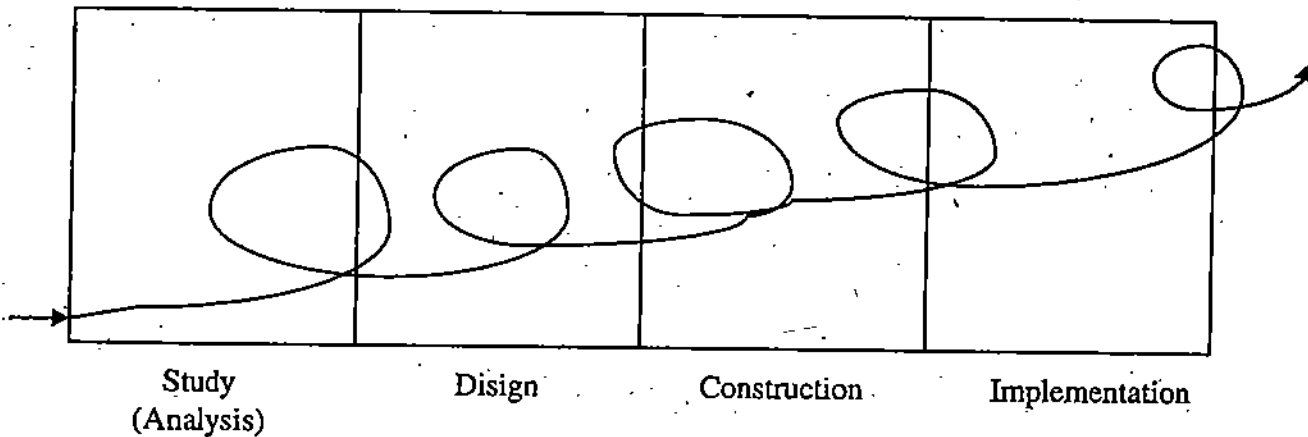
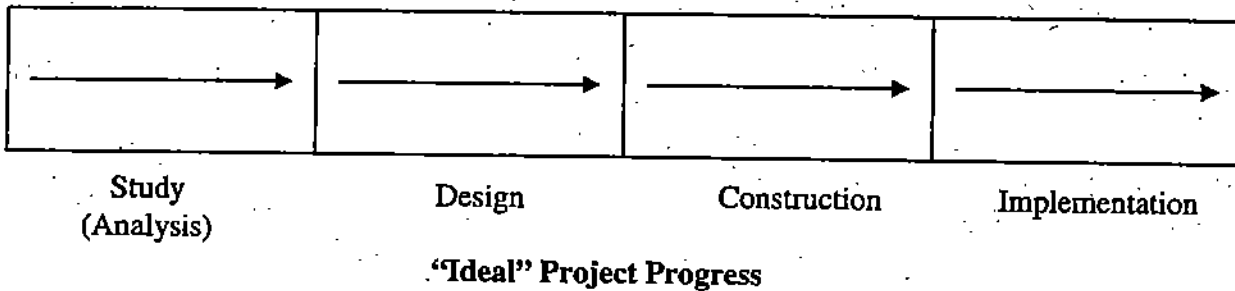


Figure 4.1 : "Reality of Project Progress

#### 4.2.5 Delivery Timings

It was decided that income could be generated as soon as the company started delivering the first versions of the product or as soon as new construct could be signed. In turn, the customers would be good references for the product.

This decision, too, turned out to be a big mistake. In act. new dimension was added to the already critical technical and management problems. Since the rule was "sign the contract anyway", while the product was not clearly defined and was only partially developed, early customers had many problems with the product. This caused a lot of internal problems also, because it was not clear whether some activity fell under product development or user assistance; nor was it clear who had to do what. After marketing, some development, some user assistance, some hardware acquisition, etc., according to an unpredictable flow of events.

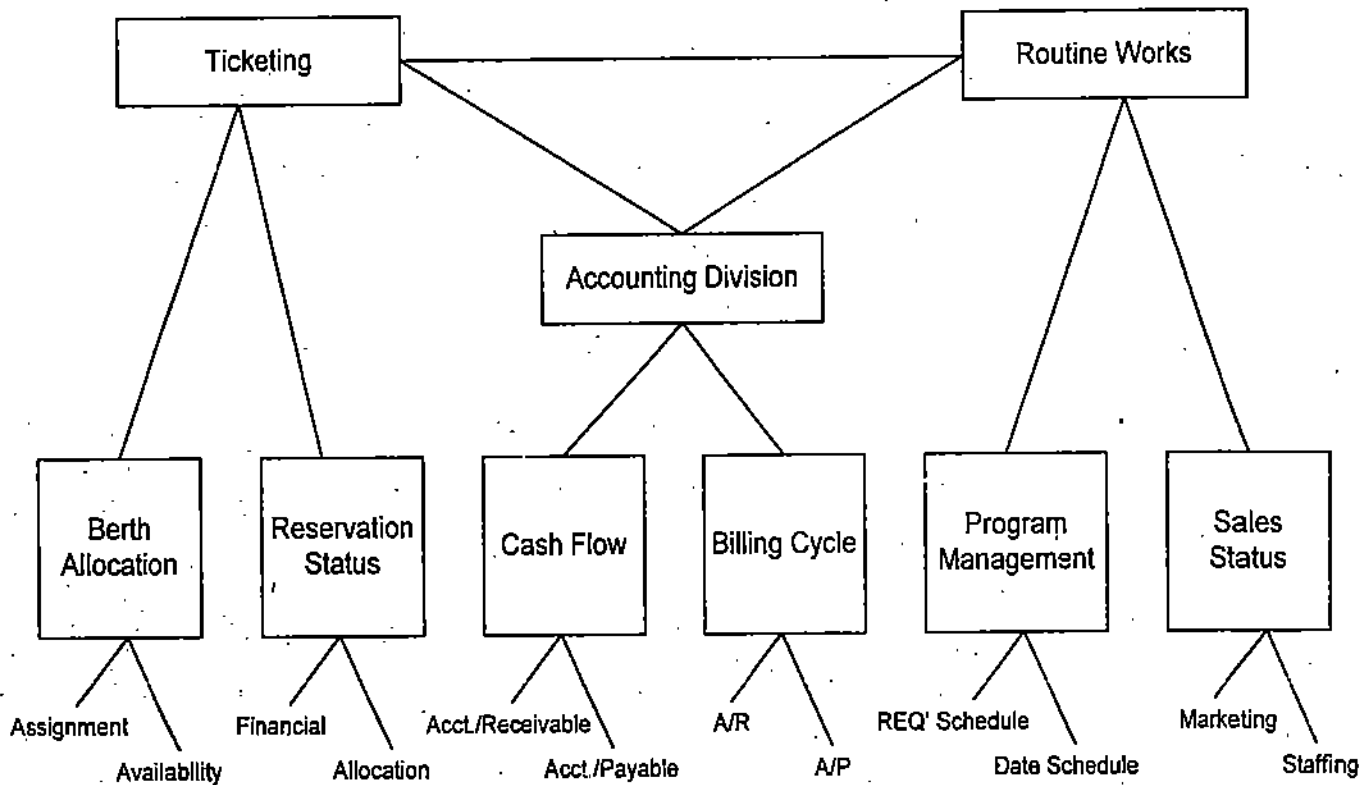


Figure 4.2 : A Simple Technical Design for Rall Ticketing System

#### 4.2.6 Partial Recovery

Eventually, it was realized that the naïve way the firm was managing the project was leading to disaster. Thus, a real effort was made, first to define responsibilities clearly (who was responsible for the design, and also for the distribution, etc.) and second, to achieve a clear picture of the state of the product, of its weaknesses, of the effort required to fix them, etc. this was done even at the expenses of slowing down the project, increasing costs, and reducing sales. Thus, people had to resist an initial feeling that the restructuring of the project was impeding "real" progress.

After a while, however, the improvement became apparent, so that eventually, the product really existed and full documentation was available. The company actually started to ship the product and earn money from its, although far less than expected

initially, mainly because the delayed introduction of the product caused it to enter a more competitive market than anticipated.

### 4.3 BENCHMARK TESTING

The term "benchmark" was derived from the days when the machinist in a factory would use measurements at each bench to determine if the parts he was machining were satisfactory. In the computing field, to compare one system with another, you would run the same set of "benchmark" programs, through each system.

A benchmark is a sample program specially designed to evaluate the performance of different computers and their software. This is necessary because computers will not generally use the same instructions, words of memory or machine cycle to solve particular problem. As regard, evaluation of software, benchmarking is mainly concerned with validation of vendor's claims in respect of following points:

- ▶ minimum hardware configuration needed to operate a package.
- ▶ time required to execute a program in an ideal environment and how the performance of own package and that of other programs under execution is affected, when running in a multi-programming mode.

The more elaborate the benchmarking, the more costly is the evaluation. The user's goals must be kept in mind. Time constraints also limit how thorough the testing process can be. There must be a compromise on how much to test while still ensuring that the software (or hardware) meets its functional criteria.

Benchmarks can be run in almost all type of systems environment including batch and on-line jobs streams and with the users linked to the system directly or through telecommunications methods.

Common benchmarks test the speed of the central processor, with typical instructions executed in a set of programs, as well as multiple streams of jobs in a multiprogramming environment. The same benchmark run on several different computers will make apparent any speed and performance differences attributable to the central processor.

Benchmarks can also be centered around an expected language mix for the programs that will be run, a mix of different set of programs and applications having widely varying input and output volumes and requirements. The response time for sending and receiving data from terminals is an additional benchmark for the comparison of systems.

#### Check Your Progress

Question 1 A stock and bond analysis program that focuses on technical analysis will

- (a) Allow to establish a database.
- (b) Analyze each security's market price and volume statistics.
- (c) both (a) and (c)
- (d) None of the above.

Question 2 A single Integrated Program may contain

- (a) Programs that take care of all the basic accounting systems used by a business.
- (b) Word Processing, Spread Sheet processing, graphics and data management.
- (c) An operating system and an application program.
- (d) both (a) and (b)

Question 3 A Structural Program

- (a) can be reduced to control structures.
- (b) is generally more complicated than non-structured program.
- (c) can only modified by the developer who wrote it.
- (d) all of the above

Question 4 The case that is hardest to fix in Software Requirement Specification is:

- (a) How user friendly the system should be
- (b) How fast the software should run.
- (c) What the software system is to do.
- (d) How accurate the outputs should be.

---

#### 4.4 MODEL ANSWERS

---

- Q1. - (c)
- Q.2 - (d)
- Q.3 - (a)
- Q.4 - (c)

---

#### 4.5 FURTHER READING

---

SOFTWARE ENGINEERING : ROGER S. PRESSMAN, TATA MCGRAW HILL PUBLISHING HOUSE





Utter Pradesh  
Rajarshi Tandon Open University

## BCA-19 Introduction to Software Engineering

Block

# 2

### Software Quality Concepts and Case Tools

---

#### UNIT 1

Software Performance 5

---

#### UNIT 2

Quality Concepts 17

---

#### UNIT 3

Software Methodology and Object Oriented Concepts 27

---

#### UNIT 4

Case Tools 51

---

---

## FACULTY OF THE SCHOOL

---

Prof. Mohan Lal  
Director (I/c)

Pro M. M. Pant

---

Shri Sharhi Bhushan  
Reader

Shri Akshay Kumar  
Reader

---

Shri P. V. Suresh  
Lecturer

Shri V. V. Subrahmanyam  
Lecturer

---

Course Co-ordinator :

P. V. Suresh  
IGNOU

---

Block Writer:

Pankaj Goel  
Consultant

---

Print Production

H. K. Som (IGNOU)

---

May, 2001

ISBN-81-266-0152-3

© Indira Gandhi National Open University

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Indira Gandhi National Open University.

Further information on the Indira Gandhi National Open University courses may be obtained from the University's Office at Maidan Garhi, New Delhi-110068.

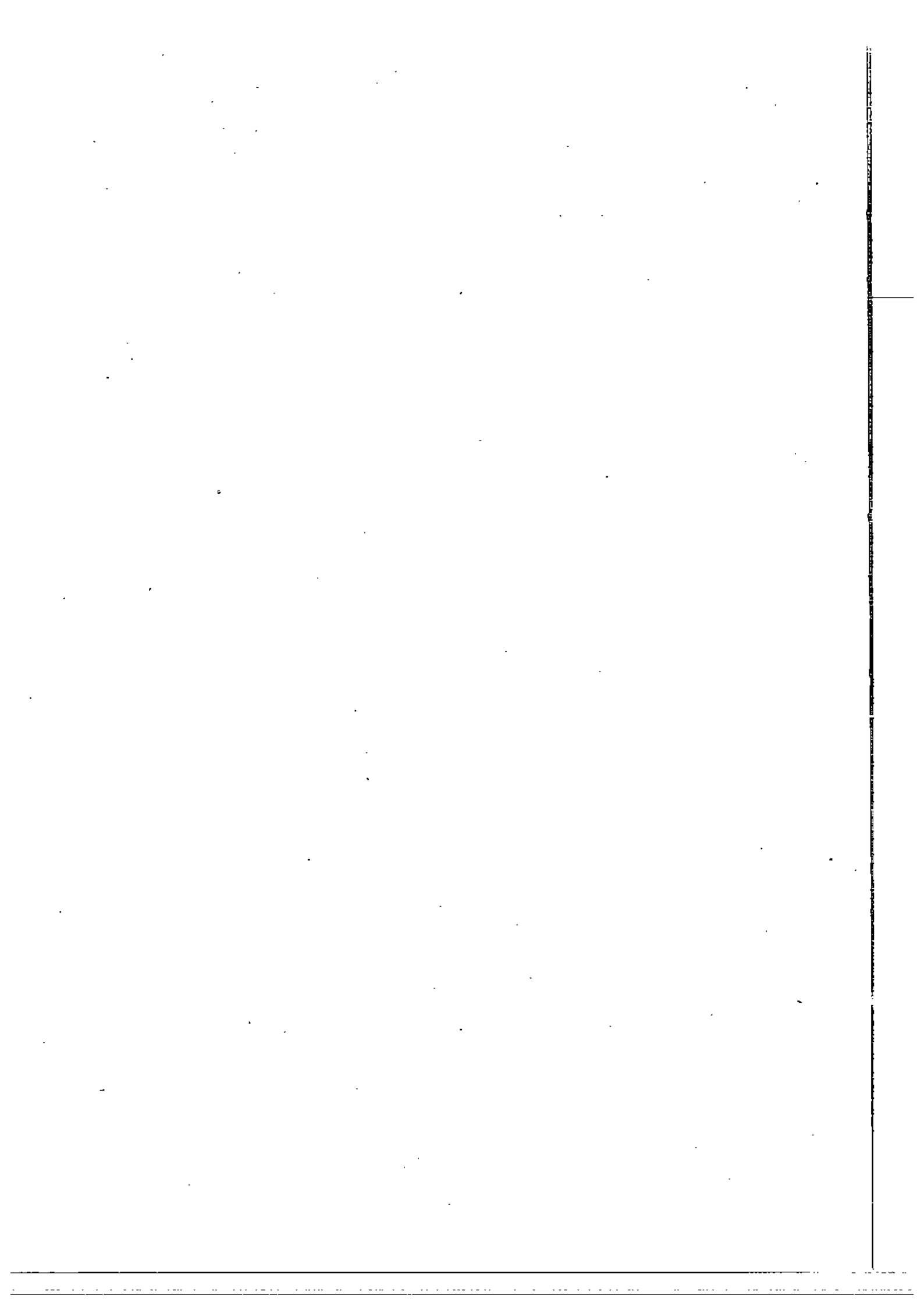
---

## **BLOCK INTRODUCTION**

---

Software engineering is one of the fields of computer science that deals with the design of a complex software system. It examines all aspects of problem solving and the integration of software into this process. This block covers different aspects of software Engineering discipline. The unit 1 focuses on various features like Software Reliability, customer friendliness, software Toolkits and Programming Environment. Unit 2 describes various qualities of software Product. (The important influencing factors of software development, how to select the right methodology is discussed in Unit 3). Unit 4 describes case tools. The students of this course are also advised to go through at least one reference book on software Engineering along with this material.

The process of selecting the right developmental methodology is discussed in Unit 3.



---

# UNIT 1 SOFTWARE PERFORMANCE

---

## Structure

- 1.0 Introduction
- 1.1 Objective
- 1.2 Customer Friendliness
- 1.3 Software Reliability
- 1.4 Software Reviews
- 1.5 Software Upgradation
- 1.6 Software Tools and Environment
- 1.7 Software Libraries and Toolkits
- 1.8 Software Modules
- 1.9 Reapplication of Software Modules
- 1.10 Development Tools
  - 1.10.1 Code Generators
  - 1.10.2 Debuggers
- 1.11 Model Answers
- 1.12 Further Readings

---

## 1.0 INTRODUCTION

---

The Software development team works towards a single goal of producing high quality software. Software Quality Assurance (SQA) is a Broad activity that comprises of:

- (a) A quality management approach.
- (b) Effective methodology and use of appropriate tools.
- (c) Technical Reviews.
- (d) Modulewise testing strategies.
- (e) Quality standards.

The software manufacturer is not responsible for any damages due to product errors. Software engineering can truly be called an engineering discipline only when we can achieve software reliability comparable to the reliability of other products.

The functional requirement specification must captures all the desirable properties of the application and no undesirable properties should be specified in it. The set of all reliable programs includes the set of correct programs, but not vice versa.

Unfortunately, things are different in practice. In fact, the specification is a model of what the user wants, but the model may or may not be an accurate statement of the user's needs and actual requirements. All that the software can do is to meet the specified requirements of the model, it cannot ensure the accuracy of the model.

A program is reliable if it behaves "reasonable", even in circumstances that were not anticipated in the requirements specification - for example, when it encounters incorrect input data or some hardware malfunction (say, a disk crash). A program that assumes perfect input and generates an unrecoverable run-time error as soon as the user inadvertently types an incorrect command would not be robust. It might be correct, though, if the requirements specification does not state what the action should be upon entry of an incorrect command. Obviously, robustness is a difficult-to-define quality; after all, if we could state precisely what we should do to make an application robust, we would be able to specify its "reasonable" behavior completely. Thus, robustness would become equivalent to correctness of reliability.

An analogy with bridges is instructive. Two bridges connecting two sides of the same river are both "correct" if they each satisfy the stated requirements. If, however, during an unexpected, unprecedented, torrential rain, one collapses and the other one does not, we can call the latter more robust than the former. Notice that the lesson learned from the collapse of the resistance to torrential rains is a correctness requirement. In other words, as the phenomenon under study becomes more and more known, we will approach the ideal case where specifications capture expected requirements.

The amount of code devoted to robustness depends on the application. For example, a system written to be used by novice computer users must be more prepared to deal with ill-formatted input than an embedded system that receives its input from a sensor. If the embedded system is controlling the space shuttle or some life critical devices, then extra robustness is advisable.

In conclusion, we can see that robustness and correctness are strongly related, without a sharp dividing line between them. If we put a requirement in the specification, its accomplishment becomes an issue of correctness: if we leave it out of the specification, it may become an issue of robustness. The border line between the two qualities is the specification of the system. Finally, reliability comes in because not all incorrect behaviors signify equally serious problems; some incorrect behaviors may actually be absorbed.

Correctness, Robustness, and Reliability also apply to the software production process. A process is robust, for example, if it can accommodate unanticipated changes in the environment, such as a new release of the operating system or the sudden transfer of half the employees to another location. A process is reliable if it consistently leads to the production of high-quality products. In many engineering disciplines, considerable research is devoted to the discovery of reliable processes.

Any engineering product is expected to meet a certain level of performance. Unlike other disciplines, in software engineering we often equate performance with efficiency. We will follow this practice here. A software system is efficient if it uses computing resources economically.

Performance is important because it affects the usability of the system. If a software system is too slow, it reduces the productivity of the users, possibly to the point of not meeting their needs. If a software system uses too much disk space, it may be too expensive to run. If a software system uses too much memory, it may affect the other applications that are run on the same system, or it may run slowly while the operating system tries to balance the memory usage of the different applications.

Underlying all of these statements-and also what makes the efficiency issue difficult-are the changing limits of efficiency as technology changes. Our view of what "too expensive" is constantly changing as advances in technology extend the limits. The computers of today cost orders of magnitude less than computers of a few years ago, yet they provide order of magnitude more power.

Performance is also important because it affects the scalability of a software system. An algorithm that is quadratic may work on small inputs but not work at all for larger inputs. For example, a compiler that uses a register allocation algorithm whose running time is the square of the number of program variables will run slower and slower as the length of the program being compiled increases.

There are several ways to evaluate the performance of a system. One method is to measure efficiency by analyzing the complexity of algorithms. An extensive theory exists for characterizing the average of worst case behavior of algorithms, in terms of significant resource requirements such as time and space, or-less traditionally-in terms of number of message exchanges (in the case of distributed system).

Analysis of the complexity of algorithms provides only average of worst case information, rather than specific information, about a particular implementation. For more specific information, we can use techniques of performance evaluation. The three basic approaches to evaluating the performance of a system are measurement, analysis, and simulation. We can measure the actual performance of a system by means of monitors that collect data while the system is working and allow us to

discover bottlenecks in the system. Or we can build a model of the product and analyze it. Or, finally, we can even build a model that simulates the product. Analytic models—often based on queuing theory—are usually easier to build but are less accurate while simulation models are more expensive to build but are more accurate. We can combine the two techniques as follows: at the start of a large project, an analytic model can provide a general understanding of the performance-critical areas of the product, pointing out areas where more thorough study is required; then we can build simulation models of these particular areas.

In many software development projects, performance is addressed only after the initial version of the product is implemented. It is very difficult—sometimes even impossible to achieve significant improvements in performance without redesigning the software. Even a simple model, however, is useful for predicting system performance and guiding design choices so as to minimize the need for redesign.

In some complex projects, where the feasibility of the performance requirements is not clear, most effort is devoted to building performance models. Such projects start with a performance model and use it initially to answer feasibility questions and later in making design decisions. These models can help resolve issues such as whether a function should be provided by software or a special-purpose hardware device.

The notion of performance also applies to a process, in which case we call it productivity. Productivity is important enough to be treated as an independent quality.

---

## 1.1 OBJECTIVES

---

After going through this unit, you should be able to:

- Describe Software Reliability, Upgradation and Customer Friendliness;
  - Explain Software Libraries and Toolkits
  - Describe Software modules and Reapplication of Software Modules.
- 

## 1.2 CUSTOMER FRIENDLINESS

---

A software system is user friendly if its human users find it easy to use. This definition reflects the subjective nature of user friendliness. An application that is used by novice programmers qualifies as user friendly by virtue of different properties than an application that is used by expert programmers. For example, a novice user may appreciate verbose messages, while an experienced user detects and ignores them. Similarly, a nonprogrammer may appreciate the use of menus, while a programmer may be more comfortable with typing a command.

The user interface is an important component of user friendliness. A software system that presents the novice user with a window interface and a mouse is friendlier than the one that requires the user to use a set of one-letter commands. On the other hand, an experienced user might prefer a set of commands that minimize the number of keystrokes rather than a fancy window interface through which he was to navigate to get to the command that he knew all along he wanted to execute. There is more to user friendliness, however, than the user interface. For example, an embedded software system does not have a human user interface. Instead, it interacts with hardware and perhaps other software systems. In this case, the user friendliness is reflected in the ease with which the system can be configured and adapted to the hardware environment.

In general, the user friendliness of a system depends on the consistency of its user and operator interfaces. Clearly, however, the other qualities mentioned above—such as correctness and performance—also affect user friendliness. A software system that produces wrong answers is not friendly, regardless of how fancy its user interface is. Also, a software system that produces answers more slowly than the user requires is not friendly even if the answers are displayed in color.

User friendliness is also discussed under the subject "human factors". Human factors or human engineering plays a major role in many engineering disciplines. For example, automobile manufacturers devote significant effort to deciding the position of the various control knobs on the dashboard. Television manufacturers and microwave oven makers also try to make their products easy to use. User-interface decision in these classical engineering fields are made, not randomly by engineers, but only after extensive study of user needs and attitudes by specialists in fields such as industrial design or psychology.

Interestingly, ease of use in many of these engineering disciplines is achieved through standardization of the human interface. Once a user knows how to use one television set, he or she operates almost any other television set.

---

### 1.3 SOFTWARE RELIABILITY

---

If an organization depends on a software for its functions then it is reliable software. Reliability of a software program is an important factor of its overall quality. Software Reliability factor can be measured and estimated. In statistical terms, the probability of failure free operation of a Software Program in a particular environment is defined as Software Reliability. Software Reviews form an important part of Software Quality Assurance Activity. The Quality Assurance team must collect data about Software Engineering Process, evaluate the data and disseminate. The ability to ensure quality is the measurement of a mature engineering discipline.

Correctness is an absolute quality: any deviation from the requirements makes the system incorrect, regardless of how minor or serious is the consequence of the deviation. The notion of reliability is, on the other hand, relative: if the consequence of a software error is not serious, the incorrect software may still be reliable.

Engineering products are expected to be reliable. Unreliable products, in general, disappear quickly from the marketplace. Unfortunately, software products have not achieved this enviable status yet. Software products are commonly released along with a list of "Known Bugs". Users of software take it for granted that "Release 1" of a product is "buggy". This is one of the most striking symptoms of the immaturity of the software engineering field as an engineering discipline.

In classic engineering disciplines, a product is not released if it has "bugs". You do not expect to take delivery of a automobile along with a list of shortcomings.

Current research and development activity in the area of standard user interfaces for software systems will lead to more user-friendly systems in the future.

---

### 1.4 SOFTWARE REVIEWS

---

A software system is verifiable if its properties can be verified easily. For example, the correctness or the performance of a software system are properties we would be interested in verifying. Verification can be performed either by formal analysis methods or through testing. A common technique for improving verifiability is the use of "software monitors" that is, code inserted in the software to monitor various qualities such as performance or correctness.

Modular design, disciplined coding practices, and use of an appropriate programming language all contribute to verifiability.

Verifiability is usually an internal quality, although it sometimes becomes an external quality also. For example, in many security-critical applications, the customer requires the verifiability of certain properties. The highest level of the security standard for a "trusted computer system" requires the verifiability of the operating system kernel.

Software maintenance is commonly used to refer to the modifications that are made to a software system after its initial release. Maintenance used to be viewed as merely "bug fixing," and it was distressing to discover that so much effort was being spent on



maintenance is in fact spent on enhancing the product with features that were not in the original specifications or were stated incorrectly.

"Maintenance" is indeed not the proper word to use with software. First, as it is used today, the term covers a wide range of activities, all having to do with modifying an existing piece of software in order to make an improvement. A term that perhaps captures the essence of this process better is "software evaluation." Second, in other engineering products, such as computer hardware or automobiles or washing machines, "maintenance" refers to the upkeep of the product in response to the gradual deterioration of parts due to extended use of the product. For example, transmissions are oiled and air filters are dusted and periodically changed. To use the word "maintenance" with software gives the wrong connotation because software does not wear out. Unfortunately, however, the term is used so widely that we will continue using it.

There is evidence that maintenance cost exceeds 60% of the total costs of software. To analyze the factors that affect such costs, it is customary to divide software maintenance into three categories: corrective, adaptive and perfective maintenance.

Corrective maintenance has to do with the removal of residual errors present in the product when it is delivered as well as errors introduced into the software during its maintenance. Corrective maintenance accounts for about 20 percent of maintenance cost.

Adaptive and perfective maintenance are the real sources of changes in software; they motivate the introduction of evolvability as a fundamental software quality and anticipation of change as a general principle that should guide the software engineer. Adaptive maintenance accounts for nearly another 20 percent of maintenance costs while over 50 percent is absorbed by perfective maintenance.

Adaptive maintenance involves adjusting the application to changes in the environment, e.g., a new release of the hardware or the operating system or a new data-base system. In other words, in adaptive maintenance the need for software changes cannot be attributed to a feature in the software itself, such as the presence of residual errors or the inability to provide some functionality required by the user. Rather, the software must change because the environment in which it is embedded changes.

Finally, perfective maintenance involves changing the software to improve some of its qualities. Here, changes are due to the need to modify the functions offered by the application, add new functions, improve the performance of the application, make it easier to use, etc. The requests to perform perfective maintenance may come directly from the software engineer, in order to improve the status of the product on the market, or they may come from the customer, to meet some new requirements.

We will view maintainability as two separate qualities: reparability and evolvability. Software is repairable if it allows the fixing of defects; it is evolvable if it allows changes that enable it to satisfy new requirements.

The distinction between reparability and evolvability is not always clear. For example, if the requirements specifications are vague, it may not be clear whether we are fixing a defect or satisfying a new requirement.

A software system is repairable if it allows the correction of its defects with a limited amount of work. In many engineering products, reparability is a major design goal. For example, automobile engines are built with the parts that are most likely to fail as the most accessible. In computer hardware engineering, there is a subsection called reparability, availability, and serviceability (RAS).

In other engineering fields, as the cost of a product decreases and the product assumes the status of a commodity, the need for reparability decreases: it is cheaper to replace the whole thing, or at least major parts of it, than to repair it. For example, in early television sets, you could replace a single vacuum tube. Today, a whole board has to be replaced.

In fact, a common technique for achieving reparability in such products is to use standard parts that can be replaced easily. But software parts do not deteriorate. Thus,

while the use of standard parts can reduce the cost of software production, the concept of replaceable parts does not seem to apply to software repairability. Software is also different in this regard because the cost of software is determined, not by tangible parts, but by human design activity.

Repairability is also affected by the number of parts in a product. For example, it is harder to repair a defect in a monolithic automobile body than if the body were made of several regularly shaped parts. In the latter case, we could replace a single part more easily than the whole body. Of course, if the body consisted of too many parts, it would require too many connections among the parts, leading to the probability that the connections might need repair.

An analogous situation applied to software: a software product that consists of well-designed modules is much easier to analyze and repair than a monolithic one. Merely increasing the number of modules, however, does not make a more repairable product. We have to choose the right module structure with the right module interfaces to reduce the need for module interconnections. The right modularization promotes repairability by allowing errors to be confined to few modules, making it easier to locate and remove them. We can examine several modularization techniques, including information hiding and abstract data types.

Repairability can be improved through the use of proper tools. For example, using a high-level language rather than an assemble language leads to better repairability. Also, tools such as debuggers can help in isolating and repairing errors.

A product's repairability affects its reliability. On the other hand, the need for repairability decreases as reliability increases.

---

## 1.5 SOFTWARE UPGRADATION

---

Like other engineering products, software products are modified over time to provide new functions or to change existing functions. Indeed, the fact that software is so malleable makes modifications extremely easy to apply to an implementation. There is, however, a major difference between software modification and modification of other engineering products. In the case of other engineering products, modification starts at the design level and then proceeds to the implementation of the product. For example, if one decides to add a second storey to a house, first one must do a feasibility study to check whether this can be done safely. Then one must do a design, based on the original design of the house. Then the design must be approved, after assessing that it does not violate the existing regulations. And, finally, the construction of the new part may be commissioned.

In the case of software, unfortunately, people seldom proceed in such an organized fashion. Although the change might be a radical change in the application, too often the implementation is started without doing any feasibility study, let alone a change in the original design. Still worse, after the change is accomplished, the modification is not even documented a posteriori; i.e., the specifications are not updated to reflect the change. This makes future changes more and more difficult to apply.

On the other hand, successful software products are quite lived. Their first release is the beginning of a long lifetime and each successive release is the next step in the evolution of the system. If the software is designed with care, and if each modification is thought out carefully, then it can evolve gracefully.

---

## 1.6 SOFTWARE TOOLS AND ENVIRONMENT

---

Up to this point we have reviewed the various phases in the software life cycle, and we have seen how to apply some principles of program design and modularity in the actual writing of programs. A natural next step is to consider how some of this work can be streamlined or automated through the use of computers. Any programs or other automated aids for the development of software are called tools, and it is appropriate to consider what kinds of software tools might increase the productivity of developments or the effectiveness of the final product.

Some of the most obvious areas for computer support of software development arise within the programming phase. At an elementary level, programming requires code to be written, compiled, run, modified, and corrected. This suggests access to an editor that is easy to use, a compiler that runs quickly and identifies errors in an understandable and helpful way, and a means to run programs conveniently with a variety of data sets.

More generally, provisions for editing, compiling, and running programs are part of a general programming environment, which includes all the features available on a particular computer to aid the programming process. A description of such an environment includes statements about what capabilities are available for the programmer and considerations of what the programmer must do to take advantages of these capabilities. For example, on small microcomputers, a programming environment might contain an editor stores on one floppy disk, a compiler on another, and programs themselves on a third disk. If the microcomputer has only one or two disk drives, programmers must swap disks each time they move from editing to compiling or back.

In contrast, another environment might include an editor and compiler in our package so that a programmer can compile and run a program at any time within an editing environment. In this case, the programmer need not spend any extra time moving from editing to compiling or running a program.

Beyond this basic integration of programming operators, more sophisticated environments may assist further with some of these tasks. For example, some modern syntax-directed editor scan a program as it is being entered or revised, and this may help automate program indenting and format. These editors may add the keyword End on a following line whenever a programmer types Begin. Similarly, these editors may insert the general form for procedures (including braces, { }, for initial comments and a Begin-End block) whenever the programmer starts a new procedure.

Other programming environments may include incremental compilers, which allow a programmer to work on one part of a program at a time. These compilers recognize what section of code has been revised recently and recompile only these procedures that have been changed. When developing large programs, this capabilities can speed up compilation considerably, since only a few lines may need to be compiled from one test to the next if the overall program contains thousands of lines of code.

Once code is first written, its tracing and debugging can be assisted by the use of a debugger, which controls the execution of the program. With debuggers, program execution may be stopped at designated points, values of variables printed or changed, and the order of procedure calls clarified. Individual procedures or groups of procedures also may be written and tested by themselves, before they are placed within an overall program. A programmer may supply initial values for parameters and then run a module to test the results before these small pieces of code are combined into larger units.

When available, such capabilities may eliminate the need to write separate driver programmes for testing.

Other advances in automating the software development process extend beyond the programming phase. Some tools assist specific phases of the process, and other tools automate parts of several phases. This gives rise to general software development environments, which may include a collection of automated aids to assist in the development process.

Both the specifications and design phases of software development have the characteristics that data must be collected organized, checked for completeness and consistency, and presented clearly for review. Several computer packages with these common features have been developed to help store informat'ion about a problem so that it can be stored and manipulated with relatively little overhead. More sophisticated systems may allow some automated checking of pieces for consistency and some possible omissions. The systems also may format parts of the specifications of designs in appropriate data flow diagrams or structure charts.

In any of these environments, automation can help streamline a particular part of the software development process, and time may be saved in the transition from one phase of the work to the next. Information and decisions within one phase can be checked for completeness and consistency, and work at one phase can draw easily on activity that has taken place earlier. In addition, this recording of information and decisions through the development of a software package can assist in the maintenance of the package since maintenance programmers will know how software is structured, what each module does, and how potential modifications will affect a module.

---

## 1.7 SOFTWARE LIBRARIES AND TOOLKITS

---

Our discussion of automated aids for software development has focussed on the development of new specifications, designs and programs. Another helpful technique involves cataloging existing modules and placing tested modules in libraries. With careful planning, it may not be necessary to spend time and effort on the development of new systems if one can reuse materials that already perform appropriate tasks.

Any attempt to take advantage of earlier work relies on at least three conditions. First, we must have a clear idea of what is needed in the current project so that we can identify the appropriate capabilities. Second, there must be a well-organized catalogue of existing software so that we can locate any modules that might be available to do pieces of the current project. Third, each existing module must be clearly documented, with carefully stated pre and post-conditions.

Although these conditions may seem obvious, it is essential that each of these points is satisfied if existing software is to be reused. For example, the existence of even one extra undocumented, Write statement in a module may Change what appears on a user's terminal. As a result a programmer may waste much time revising code, since initially the plans for some new code anticipated that the existing module behaved differently.

---

## 1.8 SOFTWARE MODULES

---

Since the purpose of problem solving is to find solutions to specific problems, it is good practice to survey existing software before planning for new code. For example, many statistical applications can often use general purpose packages, such as SPSS, BMDP, Minitab, RS/I, or S, for entering and editing data and running appropriate tests. In such instances, the use of an existing software package may save hours of years of development time.

In other cases, an existing package may do most of the work needed in an application, but some adjustment may be necessary to complete the required task. Here, it may be possible to change or enhance the existing software to meet the current needs. Alternatively, one might construct a revised flow of data for which most of the work is done with existing software but some additional work is done before or after this main step. Such a data flow is shown at a high level in Figure 1. In this Figure, the data as initially presented in the problem may not be in a form that can be used by an existing package. Thus, we may write a (Short) program or pre-processor to modify the form of the raw data so that the revised form will be appropriate for the existing package. Then, after that package has performed much of the required processing, another (short) post-processor may be needed to complete the job. In this case, Solving the initial problem may require the writing of two short programs to handle some details of processing. However, this work still may be substantially shorter and less error-prone than ignoring the existing package the writing and entire application.

More generally, much work can be saved by dividing a larger processing task into several phases, as shown in Figure 2. Here each phase performs some transformations on the data which contribute to the desired results. This approach is particularly powerful if a computing environment allows such steps to be combined in an easy way. For example, the Unix operating system support a capability called piping, in which the output of one program can be fed directly into the input of the next. This

allows long sequences of programs to be placed together easily. In such an environment, many applications may require little programming at all; rather, solutions may be developed by combining various modules in the same way that jigsaw puzzles are completed by putting individual pieces together.

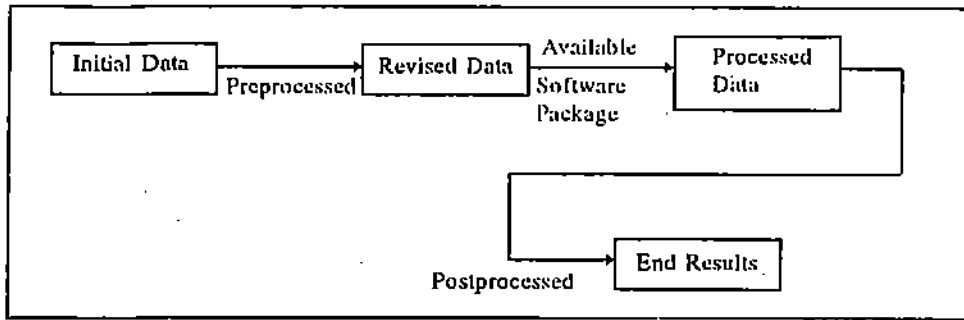


Figure 1: Modifying Data to use Existing Software

## 1.9 REAPPLICATION OF SOFTWARE MODULES

A second way to reduce the amount of code that must be written in any application is to identify tasks that must be done in several places and write one low-level module to perform the common task. As a simple example, in designing a wagon, we might start from the general concept of "wagon". Once we decide that a wagon must be supported in four places, we will not try to invent the wheel four times! In a programming context, we might determine the details of a task just once, document of pre-conditions and post-conditions for this task and how it can be used, and then refer to that common task from several places.

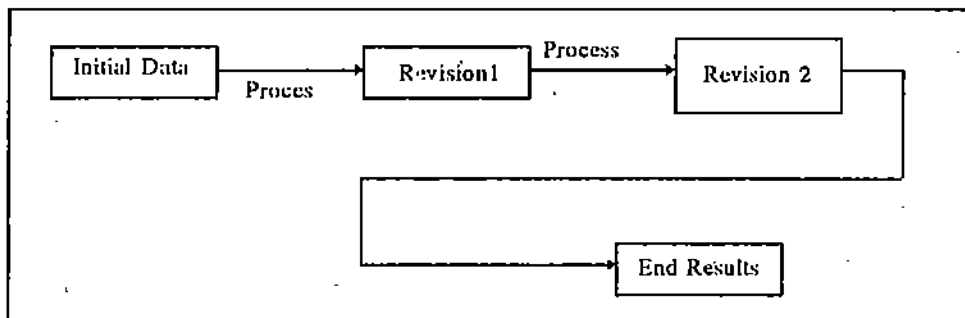


Figure 2: Data Flow Using Several Process

For example, the same types of input or output may be needed in several places within a program, and we may write some procedures to perform standard input and output tasks. In Pascal Language, we regularly rely on the Read and Write procedures in virtually every program. A similar approach can be applied to many programs when we want to read data and place them in a standard form in a particular field of a data record on array. One data entry procedure can ensure that all terminal or file input will be handled in a consistent manner throughout the program.

Between the high level use of entire programs and the low-level definition of common procedures for a specific application, we can develop a collection of these commonly used procedures and functions that we can use without further effort in a variety of applications. Such a collection of procedures is called a library, and we had experiences using such libraries.

## 1.10 DEVELOPMENT TOOLS

There are many development tools for development Environment, which include editors. Since software is ultimately a more or less complex collection of documents

requirements specifications, module architecture descriptions, programs, etc. editors are a fundamental software development tool.

With respect to the classification of the previous section, we can place editors in different categories:

- They can be either textual or graphical.
- They can follow a formal syntax, or they can be used for writing informal text or drawing free-form pictures. For instance, a general purpose graphical editor such as Apple's MacDraw could be used to produce any kind of diagrams, including formal diagrams such as DFDs or Petri nets, but cannot perform any check on their syntactic correctness (say, an incorrupt Petri net and connecting two transitions). Similarly, we may use a word processor to write programs in any programming language, but the tool cannot help in finding missing keywords, ill formed expressions, undeclared variables, etc. in order to perform such checks, one should use tools that are sensitive to the syntax - and, possible, the semantics - of the language.
- They can be either monolingual (e.g. and Ada syntax - directed editor) or polylingual (e.g., a general syntax - directed editor that is driven by the specific syntax of a programming language). A conventional word processor is intrinsically polylingual.  
  
A mode may be driven by the syntax tables of particular language, allowing simple checks and standard indentation to be performed by the editor.
- They may be used not only to produce, but also to correct or update documents. Thus, editors should be flexible (e.g., able to be run interactively or in batch) and easy to integrate with other tools. We can integrate an editor with a debugger, in order to support program correction during debugging.

Linkers are tools that are used to combine mutually objectcode fragments into a larger system. Thus, they can be both monolingual (when they are language specific) or polylingual (when they can accept modules written in different languages).

Basically, a linker binds names appearing in a module to external names exported by other modules. In the case of language-specific linkers, this may also imply kind of intermodule type checking, depending on the nature of the language. A polylingual linker may perform only binding resolution, leaving all language - specific activities to other tools.

The concept of a linker has broader applicability than just to programming language. Typically, if one deals with a modular specification language, a linker for the language would be able to perform checking and binding across various specification modules.

The concept of a linker has broader applicability than just to programming languages. Typically, if one deals with a modular specification language, a linker for that language would be able to perform checking and binding across various specification modules.

An interpreter executes actions specified in a formal notation - its input code. At one extreme, an interpreter is the hardware that executes machine code. However, it is possible to interpret even specifications, if they are written in a formal language. In this case, an interpreter behaves as a simulator or a prototype of the end product and can help detect mistakes in the specifications even in the early stages of the software process.

We already have observed that requirements specifications often occurs incrementally, hand in hand with the analysis of the application domain. Even in such cases, it would be useful to check the requirements by suitable execution of an impleately system. For example, initially one might decide to specify only the sequence of screen panels through which the end user will interact with the system, leaving out the exact specification of the functions that will be invoked in response to the user input. This decision might be dictated by the fact that, in the application under development, user interfaces are the most critical factor affecting the requirements. Thus, we would decide to check with the end user whether the interaction style we intend to provide corresponds to his or her expectations, before starting any development. This implies

that the interpreter of the specifications should be able to generate screen panels and should allow us to display sequences of screen panels in order to demonstrate the interactive sessions with the application. The interpreter should tolerate the incompleteness of specifications e.g., when no functions are provided in response to the various commands that might be entered in the field of the screen panels. In essence, the interpreter operates like a partial prototype, allowing experimentation with the look and feel of the end product.

In other cases, the result of interpreting the requirements is more properly called requirements animation; what we provide on the screen is a view of the dynamic evolution of the model, which corresponds to the physical behaviour of the specified system. For example, one might easily animate a finite state machine that is used to model the evolution of a state - changing dynamic system. If the state - changing system is a plant controlled by a computer, and a finite state machine - displayed on the terminal describes the states entered by the plant as a consequence of commands issued by the computer, we may achieve animation by blinking the states to the finite state machine as the corresponding state of the plant is entered. The control signals may be simulated by pressing any key on the terminal.

Usually, interpreters operate on actual input values. It is possible, however, to design symbolic interpreters, which operate on symbolic input values. A symbolic execution corresponds to whole class of executions on input data. Thus, a symbolic interpreter can be a useful verification tool and can be used as an intermediate step in the derivation of test data that caused execution to follow certain paths.

### 10.1 Code Generators

The software construction process is a sequence of steps that transform a given problem description called a specification into another description called an implementation. In general, the later description is more formal, more detailed, and lower level than the former; it is also more efficiently executable. The transformation process eventually ends in machine - executable code. As mentioned in the previous section, even intermediate step may be executable. The reason we decide to proceed through additional transformation step is that interpreters of intermediate are, in general, slower than the interpreter of the final product (which is the computer itself).

Derivation steps may require creativity and may be supported by tools to varying degrees. A simple and fully automatic step is the translation from source code into object code. This is performed by one of the oldest and best known software tools. The transformation may be recorded, and even controlled, by a suitable tool, but the choice of which lower level modules to use to implement a given higher level module is the designer's responsibility and cannot be automated fully.

With reference to the transformation based life cycle model, the optimizer tool is essentially a translator supporting the stepwise transformation of specifications into an implementation. As we discussed, the optimizer is only partially automated. The clerical job of recording the transformation steps is automated in order to support later modifications. We also envisioned the case where the optimizer plays the role of an intelligent assistant. The difficult and critical steps, however, cannot be automated; thus, even in this case, most of the creative tasks are the software engineer's responsibility.

Moving from a formal specification of a module to an implementation may also be viewed as a transformation that involves creative activities such as designing data structure of algorithm. Again, and clerical parts of such a transformation can be supported by automatic tools. Examples of generalized code generators are provided by several so-called fourth-generation tools, which automatically generate code for higher level language, a fourth-generation or manipulating data in the database and querying the database. Also, reports may be automatically generated from the database definition. In this case, the user can choose among several options to define report formats.

### 10.2 Debuggers

Debugging may be viewed as a kind of interpreter. In fact, they execute a program with the purpose of helping to localize and fix errors. Modern debuggers give the user the following major capabilities.

- To inspect the execution state in a symbolic way. (Here, "symbolic" means "referring to symbolic identifiers of program object", not that the debugger is a symbolic interpreter).
- To select the execution mode, such as initiating step-by-step execution or setting breakpoints symbolically.
- To select the portion for the execution state the execution points to inspect without manually modifying the source code. This not only makes debugging simpler, but also avoids the risk of-introducing foreign code that one may forget to remove after debugging.
- To check intermediate assertions.
- A debugger can also be used for other reasons than just locating and removing defects from a program. A good symbolic debugger can be used to observe the dynamic behaviours of a program. By animating the programs this way, a debugger can be a useful aid in understanding programs written by another programmer, thus supporting program modification and reengineering.

### Check Your Progress

1. What is the purpose of looping statements in a programming language?  
.....  
.....
2. Software Quality Assurance (SQA) is a broad activity that comprises of:
  - a) A Quality Management Approach.
  - b) Effective Methodology and use of Appropriate Tools..
  - c) Modulewise Testing Strategies.
  - d) All of the above.

---

### 1.11 MODEL ANSWERS

1. Looping statements are provided in a programming language to support execution of statements repeatedly.
2. (d)

---

### 1.12 FURTHER READINGS

The students of this course are advised to go through at least one standard book on Software Engineering along with this material.

1. Software Engineering - A Practitioner's Approach by ROGER S. PRESSMAN: McGraw Hill International Edition.



---

## UNIT 2 QUALITY CONCEPTS

---

### Structure

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Important Qualities of Software Product and Process
  - 2.2.1 Correctness
  - 2.2.2 Reliability
  - 2.2.3 Robustness
  - 2.2.4 User Friendliness
  - 2.2.5 Verifiability
  - 2.2.6 Maintainability
  - 2.2.7 Reusability
  - 2.2.8 Portability
  - 2.2.9 Data Abstraction
  - 2.2.10 Modularity
- 2.3 Principles of Software Engineering
  - 2.3.1 High-quality Software is Possible
  - 2.3.2 Give Products to Customers Early
  - 2.3.3 Determine the Problem Before Writing the Requirements
  - 2.3.4 Evaluate Design Alternatives
  - 2.3.5 Use an Appropriate Process Model
  - 2.3.6 Minimize Intellectual Distance
  - 2.3.7 Good Management is more Important than Good Technology
  - 2.3.8 People are the Key to Success
  - 2.3.9 Follow with Care
  - 2.3.10 Take Responsibility
- 2.4 Summary
- 2.5 Model Answers

---

## 2.0 INTRODUCTION

---

The goal of any engineering activity is to build a product. For example, the aerospace engineer builds an air plane. The product of software engineer is a software system. But the difference between software product and other product is that it is modifiable. This quality makes software quite different from other products such as cars. In this unit we will first examine the Important software qualities and then discuss software engineering principles

---

## 2.1 OBJECTIVES

---

After going through this unit, you will be able to:

- List various qualities of software product.
- DISCUSS various qualities of software product.
- Explain principles of software engineering.

---

## 2.2 IMPORTANT QUALITIES OF SOFTWARE PRODUCT AND PROCESS

---

There are many important qualities of software products. Some of these qualities are applicable both to product and to the process used to produce the product. The user

wants the software product to be reliable and user-friendly. The designer of the software want it to use maintainable portable and extensible. In this unit, we will consider all these qualities.

### 2.2.1 Correctness

A program is functionally correct if it behaves according to the specification of the functions it should provide (called functional requirements specifications). It is common simply to use the term correct rather the functionally correct; similarly, in this context, the term specification implies functional requirements specifications. We will follow this convention when the context is clear.

The definition of correctness assumes that a specification of the system is available and that it is possible to determine unambiguously whether or not a program meets the specifications. With most current software systems, no such specification exists. If a specification does exist, it is usually written in an informal style using natural language. Such a specification is likely to contain many ambiguities. Regardless of these difficulties with current specifications, however, the definition of correctness is useful. Clearly, correctness is a desirable property for software systems.

Correctness is a mathematical property that establishes the equivalence between the software and its specification. Obviously, we can be more systematic and precise in assessing correctness depending on how rigorous we are in specifying functional requirements. Correctness can be assessed through a variety of functional requirements, Correctness can be assessed through a variety of methods, some stressing an experimental approach (e.g. testing), others stressing an analytic approach (e.g. formal verification of correctness). Correctness can also be enhanced by using appropriate tools such as high-level languages, particularly those supporting extensive static analysis. Likewise, it can be improved by using standard algorithms or using libraries of standard modules, rather than inventing new ones.

### 2.2.2 Reliability

Informally, software is reliable if the user can depend on it. The specialised literature on software reliability defines reliability in terms of statistical behaviour—the probability that the software will operate as expected over a specified time interval.

Correctness is an absolute quality; any deviation from the requirements makes the systems incorrect, regardless of how minor or serious is the consequences of the deviation. The nation of reliability is on the other hand, relative; if the consequence of a software error is not serious, the incorrect software may still be reliable.

Engineering products are expected to be reliable. Unreliable products, in general, disappear quickly from the marketplace. Unfortunately, software products have not achieved this enviable status; yet, software products are commonly released along with a list of known bugs. Users of software take it for granted that Release I of a product is buggy. This is one of the most striking symptoms of the immaturity of the software engineering field as an engineering discipline.

In classic engineering disciplines, a product is not released if it has bugs. You do not expect to take delivery of an automobile along with a list of shortcomings or a bridge with a warning not to use the railing. Design errors are extremely rare and worthy of news headlines. A bridge that collapses may even cause the designers to be prosecuted in court.

On the contrary, software design errors are generally treated as unavoidable. Far from being surprised with the occurrence of software errors, we expect them. Whereas with all other products the customer receives a guarantee of reliability, with software we get a disclaimer that the software manufacturer is not responsible for any damages due to product errors. Software engineering can truly be called an engineering discipline only when we can achieve software reliability comparable to the reliability of other products.

### 2.2.3 Robustness

A program is robust if it behaves reasonably, even in circumstances that were not anticipated in the requirements specification - for example, when it encounters

incorrect input data or some hardware malfunction (say, a disk crash). A program that assumes perfect input and generates an unrecoverable run-time error as soon as the user inadvertently types an incorrect command would not be robust. It might be correct, though, if the requirements specification does not state what the action should be upon entry of an incorrect command. Obviously, robustness is a difficult-to-define quality; after all, if we could state precisely what we should do to make an application robust, we would be able to specify its reasonable behaviour completely. Thus, robustness would become equivalent to correctness.

The amount of code devoted to robustness depends on the application area. For example, a system written to be used by novice computer users must be more prepared to deal with ill-formatted input than an embedded system that receives its input from a sensor - although, if the embedded system is controlling the space shuttle or some life-critical devices, then extra robustness is advisable.

In conclusion, we can see that robustness and correctness are strongly related without a sharp dividing line between them. If we put a requirement in the specification, its accomplishment becomes an issue of correctness; if we leave it out of the specification, it may become an issue of robustness. The border line between the two qualities is the specification of the system. Finally, reliability comes in because not all incorrect behaviours signify equally serious problems; some incorrect behaviours may actually be tolerated.

Correctness, robustness, and reliability also apply to the software production process. A process is robust, for example, if it can accommodate unanticipated changes in the environment, such as a new release of the operating system or the sudden transfer of half the employees to another location. A process is reliable if it consistently leads to the production of high-quality products. In many engineering disciplines, considerable research is devoted to the discovery of reliable processes.

#### 2.2.4 User Friendliness

A software system is user friendly if its human users find it easy to use. This definition reflects the subjective nature of user friendliness. An application that is used by novice programmers qualifies as user friendly by virtue of different properties than an application that is used by expert programmers. For example, a novice user may appreciate verbose messages, while an experienced user grows to detest and ignore them. Similarly, a nonprogrammer may appreciate the use of menus, while a programmer may be more comfortable with typing a command.

The user interface is an important component of user friendliness. A software system that presents the novice user with a window interface and a mouse is friendlier than one that requires the user to use a set of one-letter commands. On the other hand, an experienced user might prefer a set of commands that minimize the number of keystrokes rather than a fancy window interface through which he has to navigate to get to the command that he knew all along he wanted to execute.

There is more to user friendliness, however, than the user interface. For example, an embedded software system does not have a human user interface. Instead, it interacts with hardware and perhaps other software systems. In this case, the user friendliness is reflected in the ease with which the system can be configured and adapted to the hardware environment.

In general, the user friendliness of a system depends on the consistency of its user and operator interfaces. Clearly, however, the other qualities mentioned above such as correctness and performance - also affect user friendliness. A software system that produces wrong answers is not friendly, regardless of how fancy its user interface is. Also, a software system that produces answers more slowly than the user requires is not friendly even if the answers are displayed in colour.

User friendliness is also discussed under the subject **human factors**. Human factors or human engineering plays a major role in many engineering disciplines. For example, automobile manufacturers devote significant effort to deciding the position of the various control knobs on the dashboard. Television manufacturers and microwave oven makers also try to make their products easy to use. User-interface decisions in these classical engineering fields are made, not randomly by engineers, but only after

extensive study of user needs and attitude by specialists in fields such as industrial design or psychology.

Interestingly, ease of use in many of these engineering disciplines is achieved through standardisation of the human interface. Once a user knows how to use one television set, he or she can operate almost any other television set. The significant current research and development activity in the area of standard user interface for software systems will lead to more user - friendly systems in the future.

### 2.2.5 Verifiability

A software system is verifiable if its properties can be verified easily. For example, the correctness or the performance of a software system are properties we would be interested in verifying. Verification can be performed either by formal analysis methods or through testing. A common technique for improving verifiability is the use of software monitors, that is, code inserted in the software to monitor various qualities such as performance or correctness.

Modular design, disciplined coding practices, and the use of an appropriate programming language all contribute to verifiability.

Verifiability is usually an internal quality, although it sometimes becomes an external quality also. For example, in many security-critical applications, the customer requires the verifiability of certain properties. The highest level of the security standard for a trusted computer system requires the verifiability of the operating system kernel.

### 2.2.6 Maintainability

The term software maintenance is commonly used to refer to the modification that are made to a software system its initial release. Maintenance used to be viewed as merely bug fixing, and it was distressing to discover that so much effort was being open on fixing defects. Studies have shown, however, that the majority of time spent on maintenance is in fact spent on enhancing the product with features that were not in the original specifications or were stated incorrectly.

**Maintenance** is indeed not the proper word to use with software. First, as it is, used today, the term covers a wide range of activities, all having to do with modifying an existing piece of software in order to make an improvement. A term that perhaps captures the essence of this process better is software evolution. Second, in other engineering products, such as computer hardware or automobiles or washing machine, **maintenance** refers to the upkeep of the product in response to the gradual deterioration of parts due to extended use of the product. For example, transmissions are oiled and air filters are dusted and periodically changes. To use the word **maintenance** with software gives the wrong connotation because software does not wear out. Unfortunately, however, the term is used so widely that we will continue using it.

There is evidence that maintenance costs exceed 60% of the total costs of software. To analyse the factors that affect such costs, it is customary to divide software maintenance into three categories; corrective, adaptive and perfective maintenance.

### 2.2.7 Reusability

Reusability is akin to evolvability. In product evolution, we modify a product to build a new version of that same product; in product reuse, we use it - perhaps with minor changes - to build another product. Reusability appears to be more applicable to software components than to whole products but it certainly seems possible to build products that are reusable.

A good example of a reusable product is the UNIX shell. The UNIX shell is a command language interpreter; that is, it accepts user commands and then executes. But it is designed to be used both interactively and in batch. The ability to start a new shell with a file containing a list of shell commands allows us to write programs -scripts - in the shell command language. We can view the program as a new product that uses the shell as a component. By encouraging standard interfaces, the UNIX environment in fact supports the reuse of any of its commands, as well as the shell, in building powerful utilities.

Scientific libraries are best known reusable components. Several large FORTRAN libraries have existed for many years. Users can buy these and use them to build their own products, without having to reinvent or recode well-known algorithms. Indeed, several companies are devoted to producing just such libraries.

Another successful example of reusable packages is the recent development of windowing systems such as X windows or Motif, for the development of user interface

Unfortunately while reusability is clearly an important tool for reducing software production costs, example of software reuse in practice are rather rare.

Reusability is difficult to achieve a posteriori, therefore, one should strive for reusability when software components are developed. One of the more promising techniques is the use of **object-oriented design**, which can unify the qualities of evolvability and reusability.

So far, we have discussed reusability the framework of reusable components, but the concept has broader applicability: it may occur at different levels and may affect both product and process. A simple and widely practiced type of reusability consists of the reuse of people, i.e. reusing their "specific knowledge of an application domain, of a development or target environment, and so on. This level of reuse is unsatisfactory, partially due to the turnover of software engineers: knowledge goes away with people and never become a permanent asset.

Another level of reuse may occur at the requirements level. When a new application is conceived, we may try to identify parts that are similar to parts used in a previous application. Thus we may reuse parts of the previous requirements specification instead of developing an entirely new one.

As discussed above, further levels of reuse may occur when the applications is designed, or even at the code level. In the latter case, we might be provided with software components that are reused from a previous application. Some software experts claim that in the future new applications will be produced by assembling together a set of ready-made, off-the-shell components. Software companies will invest in the development of their own catalogues of reusable components so that the knowledge acquired in developing applications will not disappear as people leave, but will progressively accumulate in the catalogues. Other companies will invest their efforts in the production of generalised reusable components to be put on the marketplace for use by other software producers.

Reusability applies to the software process as well. Indeed, the various software methodologies can be viewed as attempts to reuse the same process for building different products. The various life cycle models are also attempts at reusing higher level processes. Another example of reusability in a process is the replay approach to software maintenance. In this approach, the entire process is repeated when making a modification. That is, first the requirements are modified, and then the subsequent steps are followed as in the initial product development.

Reusability is a key factor that characterizes the maturity of an industrial field. We see high degrees of reusability in such mature areas as the automobile industry and consumer electronics. For example, in the automobile industry, the engine is often reused from model to model. Moreover, a car is constructed by assembling together many components that are highly standardised and used across many models produced by the same industry. Finally, the manufacturing process is often reused. The low degree of reusability in software is a clear indication that the field must evolve to achieve the status of a well-established discipline.

### 2.2.8 Portability

Software is portable if it can run in different environments. The term **environment** can refer to a hardware platform or a software environment such as a particular operating system. With the proliferation of different processors and operating systems, portability has become an important issue for software engineers.

More generally, portability refers to the ability to run a system on different hardware platforms. As the ratio of money spent on software versus hardware increases.

portability gains more importance. Some software systems are inherently machine specific. For example, an operating system is written to control a specific computer, and a compiler produces code for a specific machine. Even in these cases, however, it is possible to achieve some level of portability. Again, UNIX is an example of an operating system that has been ported to many different hardware systems. Of course, the porting effort requires months of work. Still, we can call the software portable because writing the system from scratch for the new environment would require much effort than porting it.

For many applications, it is important to be portable across operating systems. Or, looked at another way, the operating system provides portability across hardware platforms.

### 2.2.9 Data Abstraction

Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details. Thus, abstraction is a special case of separation of concerns wherein we separate the concern of the important aspects from the concern of the unimportant details.

The programming languages that we use are abstractions built on top of the hardware: they provide us with useful and powerful constructs so that we can write (most) programs ignoring such details as the number of bits that are used to represent numbers or the addressing mechanism. This helps us concentrate on the problem to solve rather than the way to instruct the machine on how to solve it. The programs we write are themselves abstractions. For example, a computersed payroll procedure is an abstraction over the manual procedure it replaces; it provides the essence of the manual procedure, not its exact details.

Data abstraction is a concept which encapsulate (collect) data structure and well defined procedure/function in a single unit. This encapsulation forms a wall which is intended to shield the data representation from computer uses. There are two requirements for data abstraction facilities in programming language.

- (i) Data structure and operations as described is a single semantic unit.
- (ii) Data structure and internal representation of the data abstractions are not visible to the programmer, rather the programmer is presented with a well defined procedural interface. Today most of the object oriented programming language support this feature.

### 2.2.10 Modularity

A complex system may be divided into similar pieces called modules. A system that is composed of modules is called modular. The main benefit of modularity is that it allows the principle of separation 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 relationship in order to integrate them into a coherent system. If the two phases are temporarily executed in the order mentioned, then we say that the system is designed bottom up; the converse denotes top-down design.

Modularity is an important property of most engineering processes and products. For example, in the automobiles 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.

We will emphasise modularity in the context of software design in the next chapter. Modularity, however, not only in a desirable design principle, but permeates the whole of software production. In particular, there are three goals that modularity tries to achieve in practice: capability of decomposing a complex system of composing it from existing modules, and of understanding the system in pieces.

The decomposability of a system is based on dividing the original problem top down into sub problems and then applying the decomposition to each sub problem

recursively. This procedure reflects the well-known Latin motto *divide et impera* (divide and conquer), which describes the philosophy followed by the ancient Romans to dominate other nations: divide and isolate them first and conquer them individually.

The composability of a system is based on starting bottom up from elementary components and proceeding to the finished system. As an example, a system for office automation may be designed by assembling together existing hardware components such as personal workstations, a network, and peripherals; system software such as the operating system; and productivity tools such as document processors, data bases and spreadsheets. A car is another obvious example of a system that is built by assembling components. Consider first the main subsystems into which a car may be decomposed; the body, the electrical system, the power system, the transmission system, etc. Each of them, in turn, is made out of standard parts; for example, the battery, fuses, cables, etc. from the electrical system. When something goes wrong, defective components may be replaced by new ones.

Ideally, in software production we would like to be able to assemble new applications by taking modules from a library and combining them to form the required product. Such modules should be designed with the express goal of being reusable. By using reusable components, we may speed up both the initial system construction and its fine-tuning. For example, it would be possible to replace a component by another that performs the same function but differs in computational resource requirements.

The capability of understanding each part of a system separately aids in modifying a system. The evolutionary nature of software is such that the software engineer is often required to go back to previous work to modify it. If the entire system can be understood only in its entirety, modifications are likely to be difficult to apply, and the result unreliable. When the need for repair arises, proper modularity helps confine the search for the source of malfunction to single components.

To achieve modular composability, decomposability, and understanding, modules must have high cohesion and low coupling.

A module has high cohesion if all of elements are related strongly. Elements of a module (e.g. statement, procedures, and declarations) are grouped together in the same module for a logical reason, not just by chance; they cooperate to achieve a common goal, which is the function of the module.

Whereas cohesion is an internal property of a module, coupling characterises a module's relationship to other modules. Coupling measures the interdependence of two modules (e.g. module A calls a routine provided by module B or accesses a variable declared by Module B). If two modules depend on each other heavily, they have high coupling. Ideally, we would like modules in a system to exhibit low coupling, because if two modules are highly coupled, it will be difficult to analyse, understand, modify, test, or reuse them separately.

Module structures with high cohesion and low coupling allow us to see modules as black boxes when the overall structure of a system is described and then deal with each module separately when the module's functionality is described or analysed. This is just another example of the principle of separation of concerns.

---

## 2.3 PRINCIPLES OF SOFTWARE ENGINEERING

---

Engineering disciplines have principles based on the laws of physics, biology, chemistry or mathematics. Principles are rules to live by, they represent the collected wisdom of many dozens of people who have learned through experience.

Because the product of software engineering is not physical, physical laws do not form a suitable foundation. Instead, software engineering has had to evolve its principles based solely on observation of thousands of projects. The following are probably the more important ones. A customer will not tolerate a poor-quality product, regardless of how you define quality. Quality must be quantified and mechanisms put into place to motivate and reward its achievement. It may seem politically correct to deliver a product on time, even though its quality is poor, but this

is correct only in the short term, it is suicide in the middle and long term. There is no trade-off to be made here. The first requirement must be quality.

However, there is no one definition of software quality. To developers, it might be elegant design or elegant code. To users, it might be good response time or high capacity. For cost-conscious managers, it might be low development cost. For some customers, it might be satisfying all their perceived and no-yet-perceived needs. The dilemma is that these definitions may not be compatible.

### **2.3.1 High-quality Software is Possible**

Although our industry is saturated with examples of software systems that perform poorly, are full of bugs, or otherwise fail to satisfy user needs, there are counter examples. Large software systems can be built with very high quality but they carry a steep price tag - on the order of \$1,000 per line of code. One example is IBM's on-board flight software for the space shuttle: three million lines of code with less than one error per 10,000 lines.

Techniques that have been demonstrated to increase quality considerably include involving the customer, prototyping (to verify requirements before full-scale development), simplifying design, conducting inspections, and hiring the best people.

### **2.3.2 Give Products to Customers Early**

No matter how hard you try to learn user's needs during the requirements phase, the most effective way to ascertain real needs is to give users a product and let them play with it. The conventional waterfall model delivers the first product after 99 percent of the development resources have been expended. Thus, the majority of customer feedback on need occurs after resources are expended. Contrast this with an approach that you deliver a quick-and-dirty prototype early in development, gather feedback, write a requirements specification, and then proceed with full-scale development. In this scenario, only five to twenty percent of development resources have been expended when customers first see the product.

### **2.3.3 Determine the Problem Before Writing the Requirements**

When faced with what they believe is a problem, most engineers rush to offer a solution. If the engineer's perception of the problem is accurate, the solution may work. However, problems are often elusive. The occupants in high-rise buildings always complain of long waits for an elevator. Is this really the problem? And whose problem is it? From the occupants' perspective, the problem might be that the wait is a waste of time. From the building owner's perspective, the problem might be that long waits will reduce occupancy (and thus rental income). The obvious solution is to increase the speed of the elevators. But you could also add elevators, stagger working hours, reserve some elevators for express service, increase the rent, or refine the homing algorithm so elevators go to high-demand floors when they are idle. The range of costs, risks, and time associated with these solutions is enormous. Yet any one could work, depending on the situation. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution.

### **2.3.4 Evaluate Design Alternatives**

After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use an architecture simply because it was used in the requirements specification. After all, that architecture was selected to optimize the understandability of the system's external behavior. The architecture you want is the one that optimizes conformance with the requirements.

For example, architectures are generally selected to optimize constructability, throughput, response time, modifiability, portability, interoperability, safety, functional requirements. The best way to do this is to enumerate a variety of software architectures, analyze (or simulate) each with respect to the goals, and select the best alternative. Some design methods result in specific architectures, so one way to generate a variety of architectures is to use a variety of methods.



### 2.3.5 Use an Appropriate Process Model

There are dozens of process models: waterfall, throwaway prototyping, incremental, spiral, operational prototyping, and so on. There is no such thing as a process model that works for every project. Each project must select a process that makes the most sense for that project, on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well-understood.

Study your project's characteristics and select a process model that makes the most sense. When building a prototype for example, choose a process that minimizes protocol, facilitates rapid development and does not worry about checks and balances. Choose the opposite when building a life-critical product.

### 2.3.6 Minimize Intellectual Distance

Edsger Dijkstra defined intellectual distance as the distance between the real-world problem and the computerized solution to the problem. Richard Fairley has argued that the smaller the intellectual distance, the easier it is to maintain the software. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure. This is the primary motivation for approaches such as object-oriented design and Jackson System Development. But you can minimize intellectual distance using any design approach. Of course, the real-world structure can vary as Jawed Siddiqi points out (*Challenging Universal Truths of Requirements Engineering*, Mar. 1994, pp. 18-19). Different humans perceive different structures when they examine the same real world and thus construct quite different realities.

### 2.3.7 Good Management is More Important than Good Technology

The best technology will not compensate for poor management, and a good manager can produce great results even with meager resources. Successful software start-ups are not successful because they have great process or great tools (or great products for that matter!). Most are successful because of great management and great marketing.

Good management motivates people to do their best, but there are no universal right styles of management. Management style must be adapted to the situation. It is not uncommon for a successful leader to be an autocrat in one situation and a consensus-based leader in another. Some styles are innate, others can be learnt.

### 2.3.8 People are the Key to Success

Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and process will succeed. The wrong people with appropriate tool, languages and process will probably fail (as will the right people with insufficient training or experience). When interviewing prospective employees, remember that there is no substitute for quality. Don't compare two people by saying, *Person x is better than person y but person y good enough and less expensive*. You can't have all superstars, but unless you truly have an overabundance, hire them when you find them!

### 2.3.9 Follow with Care

Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping - all these, might increase quality, decrease cost, and increase user satisfaction.

However, only those organization that can take advantage of them will reap the rewards. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal. You can't afford to ignore a new technology. But don't believe the inevitable hype associated with it. Read carefully. Be realistic with respect to payoffs and risks. And run experiments before you make a major commitment.

### 2.3.10 Take Responsibility

When a bridge collapses we ask, *what did the engineers do wrong?* When software fails we rarely ask this. When we do, the response is, *I was just following the 15 steps*

of this method, or My manager made me do it or The schedule left in sufficient time to do it right. The fact is that in any engineering discipline the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs.

There are no excuses. If you develop a system, it is your responsibility to do it right. Take that responsibility. Do it right, or don't do it at all.

### Check Your Progress

1. What are the requirements of data abstraction facilities in a language ?

.....  
.....

2. How reusability is supported in object oriented programming language?

.....  
.....

---

## 2.4 SUMMARY

Software engineering deals with the applications of engineering principles to the building of software products. To arrive at a set of engineering principles, one has to select a set of qualities that characterise the products. In this unit we presented a set of qualities for software product. At the end, we discussed several principles which should be applied in designing software products that achieve these qualities.

---

## 2.5 MODEL ANSWERS

1. (i) The data structure and operations are described in a single unit, and  
(ii) The data structures and internal representation of the data observation are not visible to the programmer; rather the programmer is presented with a well-defined procedural interface.
2. It is supported through inheritance feature.

---

## UNIT 3 SOFTWARE METHODOLOGY: AN OBJECT ORIENTED CONCEPTS

---

### Structure

- 3.0 Introduction
- 3.1 Objectives
- 3.2 The Evolving Role of Software
- 3.3 An Industry Perspective
- 3.4 Some Initial Solutions
- 3.5 Structured Methodologies
- 3.6 Major Influencing Factors
  - 3.6.1 Evolution of End-user Computing
  - 3.6.2 Emergence of Case Tools
  - 3.6.3 Use of Prototyping and 4GL Tools
  - 3.6.4 Relational databases
  - 3.6.5 Object Oriented Programming
  - 3.6.5 Graphical User Interfaces
- 3.7 Using the Methodology
- 3.8 Choosing the Right Methodology
- 3.9 Implementing a Methodology
- 3.10 Which Tools are you Most Likely to Use
- 3.11 Current Generation of Software Developing Tools
  - 3.11.1 Fourth Generation
  - 3.11.2 Fifth Generation
- 3.12 4GLs
  - 3.12.1 What is a 4GL
  - 3.12.2 End User Computing
  - 3.12.3 Prototyping
  - 3.12.4 Non-Procedural
- 3.13 Considerations in Applications Development
  - 3.13.1 Problems in Application Development
  - 3.13.2 How 4GLs Help to Solve Problems
  - 3.13.3 Limitation of 4GLs
  - 3.13.4 Impact of 4GLs
  - 3.13.5 What to Look for in a 4GL
- 3.14 Summary
- 3.15 Model Answers

---

### 3.0 INTRODUCTION

---

During the first three decades of the computing era, the primary challenge was to develop computer hardware that reduced the cost of processing and storing data. Throughout the decade of the 1980s, advances in microelectronics resulted in more computing power at an increasingly lower cost. Today, the problem is different. The primary challenge during the 1990s is to improve the quality (and reduce the cost) of computer-based solutions - solutions that are implemented with software.

The power of a 1980s-era mainframe computer is available now on a desk top. The awesome processing and storage capabilities of modern hardware represent computing potential. Software is the mechanism that enables us to harness and tap this potential.

---

## 3.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss major influencing factors on s/w development
- Select right methodology for software development
- List current generation of s/w development tools
- Discuss features of 4 GL.

---

## 3.2 THE EVOLVING ROLE OF SOFTWARE

---

The context in which software has been developed is closely coupled to almost five decades of computer system evolution. Better hardware performance, smaller size and lower cost have precipitated more sophisticated computer-based systems. We have moved from vacuum tube processor to microelectronic devices that are capable of processing 200 million instructions per second.

During the early years of computer system development, hardware underwent continual change while software was viewed by many as an afterthought. Computer programming was a art for which few systematic methods existed. Software development was virtually unmanaged - until schedules slipped or costs began to escalate. During this period, a batch orientation was used for most systems. Notable exceptions were inactive systems such as the early American Airlines reservation system and real-time defense oriented systems. For the most part, however, hardware was dedicated to the execution of a single program that in turn was dedicated to a specific application.

During the early years, general-purpose hardware became common-place Software, on the other hand, was custom-designed for each application and had a relatively limited distribution. Product software (i.e., programs developed to be sold to one or more customers) was in its infancy. Most software was developed and ultimately used by the same person or organization. You wrote it, you got it running, and if it is failed, you fixed it. Because job mobility was low, managers could rest assured that you would be there when bugs were encountered.

Because of this personalized software environment, design was an implicit process performed in one's head, and documentation was often nonexistent. During the early years we learned much about the implementation of computer-based systems, but relatively little about computer system engineering. In fairness, however, we must acknowledge the many outstanding computer-based systems that were developed during this era. Some of these remain in use today and provide landmark achievements that continue to justify admiration.

The second era of computer system evolution spanned the decade from the mid-1960s to the late 1970s. Multiprogramming the multi-user systems introduced new concepts of human-machine interaction. Interactive techniques opened a new world of applications and new levels of hardware and software sophistication. Real-time systems could collect, analyze and transform data from multiple sources, thereby controlling processes and producing output in milliseconds rather than minutes. Advances in on-line storage led to the first generation of database management system.

The second era was also characterized by the use of product software and the advent of software houses. Software was developed for widespread distribution in a multidisciplinary market. Programs for mainframes and minicomputers were distributed to hundreds and sometimes thousands of users. Entrepreneurs from industry, government, and academia broke away to develop the ultimate software package and earn a bundle of money.

As the number of computer-based systems grew, libraries of computer software began to expand. In-house development projects produced tens of thousands of programs

source statements. Software products purchased from the outside added hundreds of thousands of new statements. A dark cloud appeared on the horizon. All of these program-all of these source statements-had to be corrected when faults were detected, modified as user requirements changed, or adapted to new hardware that was purchased. These activities were collectively called software maintenance. Effort spent on software maintenance began to absorb resources at an alarming rate.

Worse yet, the personalized nature of many programs made them virtually unmaintainable. A software crisis loomed on the horizon.

The third era of computer system evolution began in the mid-1970s and continues today. The distributed system-multiple computers, each performing functions concurrently and communicating with one another-greatly increased the complexity of computer-based systems. Global and local area networks, high-bandwidth digital communications, and increasing demands for 'instantaneous' data access put heavy demands of software developers.

The third era has also been characterized by the advent and widespread use of microprocessors, personal computers, and powerful desk top workstations. The microprocessor has spawned a wide array of intelligent products-from automobiles to microwave ovens, from industrial robots to blood serum diagnostic equipment. In many cases, software technology is being integrated into products by technical staff who understand hardware but are often novices in software development.

The personal-computer has been the catalyst for the growth of many software companies. While the software companies of the second era sold hundreds of copies of their programs, the software companies of the third era, sold more than hundreds of thousands of copies. Personal computer hardware is rapidly becoming a commodity, while software provides the differentiating characteristic. In fact, as the rate of personal computer sales growth flattened during the mid-1980s, software product sales continued to grow. Many people in industry and at home spent more money on software than they did to purchase the computer on which the software would run.

The fourth era in computer software is just beginning. Object-oriented technologies are rapidly displacing more conventional software development approaches in many application areas. Authors predict that fifth-generation computers, with radically different computing architectures, and their related software will have a profound impact on the balance of political and industrial power throughout the world. Already, fourth generation techniques for software development are changing the manner in which some segments of the software community build computer programs. Expert systems and artificial intelligence software has finally move from the laboratory into practical application for wide-ranging problems in the real world. Artificial neural network software has opened exciting possibilities for pattern recognition and human-like information processing abilities.

As we move into fourth era, the problems associated with computer software continue to intensify.

1. Hardware sophistication has outpaced our ability to build software to tap hardware's potential.
2. Our ability to build new programs cannot keep pace with the demand for new programs.
3. Our ability to maintain existing programs is threatened by poor designs and inadequate resources.

In response to these problems, software engineering practices are being adopted throughout the industry.

---

### 3.3 AN INDUSTRY PERSPECTIVE

---

In the early days of computing, computer-based systems were developed using hardware-oriented management. Project managers focused on hardware because it was the single largest budget item for system development. To control hardware costs,

managers instituted formal controls and technical standards. They demanded thorough analysis and design before something was built. They measured the process to determine where improvements could be made. Stated simply, they applied the controls, methods, and tools that we recognize as hardware engineering. Sadly, software was often little more than an afterthought.

In the early days, programming was viewed as an art form. Few formal methods existed and fewer people used them. The programmer often learned his craft by trial and error. The jargon and challenges of building computer software created a mystique that few managers cared to penetrate. The software world was virtually undisciplined - and many practitioners of the day loved it.

Today, the distribution of costs for the development of computer-based systems has changed dramatically. Software, rather than hardware, is often the largest single cost item. For the past decade managers and many technical practitioners have asked the following questions:

- Why does it take so long to get programs finished?
- Why are costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we have difficulty in measuring progress as software is being developed.

These, and many other questions, are a manifestation of the concern about software and the manner in which it is developed - a concern that has led to the adoption of software engineering practices.

The efforts that go into the design and development of computerised applications are enormous: the detailed studies need to be done for analysing the minute nuances of the manual system, defining the functions of the new system, designing, coding, testing and finally implementing it live.

But if we look closely at any computerisation exercise, we find that the same set of activities need to be performed even if the system is a run-of-the-mill. Of course, when the system is implemented, the end-users invariably find certain things not happening the way they want, or find errors and bugs. So, the software enters the Maintenance phase when these problems are tackled by the developers. These activities, from Problem Definition to Implementation and Evaluation, constitute the so-called System Development Life Cycle (SDLC).

The expectations of end-users from these miracle machines have multiplied manifold. With so much hype surrounding computers and their capabilities, users are often not clear on what to expect from computers and how soon; thus if the computerised system fails to perform as per their dreams, the users get disillusioned with it very soon! Also with their penchant for jargon, the systems personnel are often unable to communicate properly with the end-users and rarely come to a mutual understanding of the problem on hand and the deliverables expected by the user.

With the task defined as ambiguously, it is no wonder that neither of the parties agree upon the final product and the result is the endless maintenance phase and cost and time overruns that computerisation projects are notorious for. Fire-lighting takes centre-stage, relegating systematic ways of working to the back seat. Documentation, it always never keeps pace with the system.

### 3.4 SOME INITIAL SOLUTIONS

One of the early solutions proposed for these problems was Structured Programming. But soon it was found to be inadequate since merely writing programs in a more disciplined way did not seem to improve things drastically. The key apparently lay in having a more holistic picture of the problem; giving enough thought to the way the program modules are interfaced with each other; and in minimising the effect of changes in one module on other modules. This gave birth to Structured Design with its

principle of Top-Down design, minimum coupling between modules, maximum cohesion within the module, and so on.

But It still does not help, if you have an excellent solution that addresses the wrong problem. Or for a problem that is itself ill understood. The real solution, hence, was found in Structured Analysis which was to come first in the chain of interrelated activities of the SDLC. This would help in understanding the existing system well and defining users' problems and requirements. Thus, having clearly understood the

'What of the problem, Structured Design would look for the 'How' of the solution and Structured Programming would carry out this sound design.

Along with these principles, came a host of techniques like the ones for modelling the system's functions (Data Flow Diagrams or DFDs); understanding the relationships between the system's data groups (Entity Relationship Diagrams or ERDs); designing optimal data groups (Normalisation); designing optimal program modules (Structure Charts or Program Structure Diagrams), etc. Used in isolation with the above limited objectives, this set of discrete tools again fell short somewhere, and the search continued.

---

### 3.5 STRUCTURED METHODOLOGIES

---

The next quantum jump come in form of integrated structured methodologies which specify:

- a set of activities that to be carried out in developing an information system;
- the sequence and interrelationships of these activities;
- the tools and techniques that will be used for accomplishing these activities; and
- the way to record the results of these tasks.

The proposed system is detailed as a Structured Specification and has the properties of being graphical, top-down partitionable (moving from an interview to the details), and clearly separates the Physical Model (system as it is implemented from the logical model (devoid of physical details retaining only the functionality or essence) of the system. This specification forms the basis for the Structured Design phase and the Program Specifications derived at the end of this later phase lead into the coding and so on.

A methodology integrates the activities of the SDLC and the completion of the earlier phase is a pre-condition to the start of the later phases. Of course, some amount of controlled iteration is also allowed within or across phases in order to correct errors or to accommodate changes owing to better understanding of the system as the cycle proceeds.

Thus, structured methodologies go well beyond merely clubbing a set of tools, They have a well defined scope of coverage, prescribing standards and conventions and providing a reference framework applicable to all projects. They enforce simultaneous documentation and define the deliverables from the system clearly. Estimates can be made easily and clear work breakdown structure facilitates Project Management.

Structured methodologies are thus beneficial to a whole host of constituents like the management, end-users, systems management, analysts, programmers, Q.A. personnel, and auditors.

---

### 3.6 MAJOR INFLUENCING FACTORS

---

Some of the recent developments that influence the way Methodologies are being looked at are:

1. Evolution of End-user computing
2. Emergence of CASE tools

3. Use of Prototyping and 4GL tools
4. Relational Data Bases
5. Object Oriented Programming
6. Graphical User Interfaces.

### 3.6.1 Evolution of End-User Computing

This has brought in a mixture of good and bad influences on systemisation of the development process. Since end-users are now more familiar with computer technology and have first-hand exposure to it, they are more aware of its potential and its limitations. They can be involved more in the systems development process and interact more fruitfully with systems personnel. On the other hand, based on their interaction with small PC-based systems of limited power, they may imagine systems analysis, and proper problem definition to be redundant, and expect coding to start on-line on the PC! The importance of documentation may again be lost on them.

### 3.6.2 Emergence of CASE Tools

As in other application areas, more and more work in systems development is also being transferred to the computer itself, while human beings retain only the control element. Computer Aided Software Engineering (CASE) tools are based on the above philosophy and some of their feature are: Diagramming support, screen painting, data dictionary maintenance, documentation support, etc. Certain CASE tools provide methodology-specific support and many of them are multi-user tools.

Some advantages of CASE tools are : Integration of the activities of the SDLC, automatic standardisation, guaranteed correctness and level of quality, removal of monotony, self-documentation, reduce cost and time of development, etc.

RDBMS engineers use the CASE system to:

- Assist them in gathering the initial requirements from end-user
- Analyze these requirements and determine their feasibility
- Design the system's general algorithms.
- Design an actual detailed implementation in terms of the target environment (hardware and operating system, specific RDBMS, etc.)
- Check their designs for completeness and consistency, and for contravention of specific RDBMS naming conventions.
- Automatically generate the RDBMS (tables, indices and forms) from the design.
- Maintain their existing system by reverse engineering the original databases from their host machines to Delt.
- Control their development efforts through the medium of our configuration management tools.

CASE methods employ the structured approach to software engineering and comprise various methods in which one draws diagrams or models of the computer system to be built. The models each portray a certain aspect of the system, with four views required to adequately model a system that uses an RDRMS as the data repository. These four views are: Data Flow Diagrams (DFSs), Entity Relationship Diagrams (ERDs), Program Structure Diagrams (PSDs) and Form/Report templates.

Some disadvantages of CASE Tools could be : Reinforcement of the tendency of systems personnel to work only with the machine (and not with human beings, e.g., users in Analysis phase, or colleagues in Design phase); loss of data due to improper security or corruption, high cost for reasonable level of sophistication, etc. There may be a belief that the CASE tool is a panacea for all ills.



1. What are the advantages of CASE Tools?

.....  
 .....  
 .....

2. What are the disadvantages of CASE Tools?

.....  
 .....  
 .....

**3.6.3 Use of Prototyping and 4GL Tools**

Prototypes can be useful starting in developing a model of the proposed system, help in establishing the requirements more clearly. They may be constructed to simulate the business functionality of the system, its scope of coverage, ease of use and suitability to the organisation's way of working, etc. Once the initial vagueness about the users' expectations and functionality of the system has been cleared by prototyping, a more formal systems analysis and design can refine the prototype further. Thus, by using prototyping as a complement to the use of a Methodology, the advantages of both approaches can be retained.

**3.6.4 Relational Databases**

The advent of relational data base technology opened up the use of data bases directly by the end users with minimal programming skills. Today there are many RDBMS available under different group of hardware platforms. Some of the databases and the hardware platforms they are available on are tabulated below:

H/W Group	RDBMS
PCs	Focus, Ingres, Oracle, etc.
Minis	Focus, Ingres, Oracle, Sybase, Informix, Unify, etc.
Main Frames	Ingres, Oracle, RDB, DB2, BAS/S+, etc.

From the above table, it is clear that most of the databases available do not meet the first requirement viz. Availability across divergent hardware platforms. Only Oracle and Ingres are available under different platforms i.e., on PCs, minis and mainframes. Most of the other databases have interface to import data from and export data to different databases on other hardware platforms. But such interfaces do not give optimum performance. As a result developers get bogged down by performance issues. Interfaces available with other databases are not user-oriented, and hence, they are not very user-friendly.

Two databases, viz. Oracle and Ingres, can run across different platforms. They can also transfer data from various hardware platforms without any conversions of programs. They, thus, satisfy the feature of having Open architecture and Distributed data management capability.

These databases have industry standard SQL and report writer which are 4GL tools. Screen-oriented development tools for painting entry screens and menus is an inherent feature of both Oracle and Ingres. This not only facilitates faster development, but also enhances Professional productivity.

Oracle and Ingres have a query optimizer. The main function of a query optimizer is to determine automatically the fastest method in which a database request can be handled. As a result of this, programmers and end-users do not need any additional

training to obtain good RDBMS performance. These databases have servers or data managers which minimizes both memory and CPU resource utilization. This ensures high performance during transaction processing.

None of the above mentioned databases have the feature of compound document handling. RDBMS have always had robust tools for fixed-length alphanumeric data, and tracking of applications. However, they have not been able to handle unstructured text. Their only mechanism for searching words and phrases in a text field is through sequential string match over the entire database. Some databases have text retrieval applications on top of their RDBMS, but their server programs are not designed for large text transaction. As a result they suffer from poor response time and lack of text navigation features.

Thus, a compound document having collection of separate data objects like text, graphics, images, logical structures, layout structures, voice and annotations that can be edited, formatted or otherwise processed as a whole cannot be stored and retrieved by a RDBMS.

The databases of the future should not only be able to handle different objects (Object Oriented Database) but also have the following additional features to achieve, if not paperless office, at least less-paper office.

- Accept document from disparate sources
- Handle several document structures
- Provide from complete document identification
- Dynamic and deferred updating of different data structures
- Thesaurus-based concept searching for text
- Stopword and title control while indexing
- Context searching for text
- Device independent searching
- Soundex and plural control
- Reproduction of stored voice after successful search
- Window based interface for end-users.

### 3.6.5 Object Oriented programming

As the sophistication and capabilities of new computer hardware have grown by leaps and bounds, it has become evident that new software development techniques are needed. Users anxiously await more software that harnesses and capabilities of sophisticated hardware such as the Macintosh II and the PS/2 family of computers. In addition, users' expectations of software quality have increased. They are no longer satisfied with applications that are either not user-friendly or having numerous bugs which must be worked around.

A recent approach to Software development, called object-oriented programming, attempts to completely alter traditional software development methods, and many computer professionals believe it has the potential for satisfying some of the enormous demand for more sophisticated software. Object-oriented programming requires a new way of programming, one more closely related to how we actually think.

We typically regard the computer as a machine and data as the raw material that the computer processes. The programmer is the technical who controls the machine. The program lists all the steps the computer must take to obtain the needed output from the input. However, an object-oriented program defines the data and the set of operations that can act on that data as one unit called as object. The object is thought of as an actor with a specific set of skills. The programmer is the director of the show. The programmer no longer has to tell each actor exactly what steps to take but instead simply explains the ultimate task to be performed.

Critical to understanding object-oriented programming is the concept of inheritance. Objects can be defined and then used to build a hierarchy of descendant objects, each of which inherits access to methods used by the ancestors' objects. Objects can be reused. Even when a new object is needed, an old one can usually be modified to meet the new needs. The new object inherits the characteristics of the old object.

For example, a horse is a subclass of mammals. It inherits the characteristics of a mammal (body hair, live birth, nursing its young and so on). However, it also has characteristics that distinguish it from other mammals, such as its size and shape, the way it moves, and the kinds of sounds it makes. When a programmer creates a new object, it is necessary only to add its new features: the inherited ones are already there.

To see how this makes the programmer's job easier, assume you were writing a space-war game. Both sides - the Federation and the Ferengi - have space ships but of slightly different types. In addition, each side has non-lightning ships, such as space shuttles and cargo barges. The object ship has certain characteristics: X- Y coordinates, Shields, warp speeds, and loyalty (Federation or Ferengi). The object lightning ship has everything the ship has plus photon torpedoes. The object shuttlecraft has everything the ship has except shield and warp speeds.

Many programmers agree that object-oriented programming can greatly reduce the time needed to implement new software. In addition, because new software builds heavily on existing objects, the code is more likely to be reusable and error-free. Several object-oriented languages are currently available. The first commercially available language was Xerox's Smalltalk. Other languages include C++ (an object-oriented version of C), and Borland's Turbo Pascal, Version 5.5. The next few years will determine whether the full potential of object-oriented programming is as great as many computer professionals believe. If it is, we should see tremendous improvements in the speed of software development and the quality of the final product.

### Check Your Progress 2

1. What are the important features of Oracle and Ingres?

.....  
.....  
.....  
.....

2. What will be the features of the future database?

.....  
.....  
.....  
.....

3. What is object-oriented programming?

.....  
.....  
.....  
.....

### 3.6.6 Graphical User Interfaces

Graphical user interfaces (GUIs) offer a standard look and feel to application thus reducing development and learning time. A GUI is an application environment that can work with graphical objects. Microsoft Windows, a typical example, has the following components:

- Menus (including placement and names of options, and style such as pulldown or popup)
- Icons (for identifying applications and resources)
- Tiled windows (for views of multiple programs of data or for multiple views of a single program of data block)
- Dialog boxes for selecting files, options, and settings; when an option is selected, the one previously selected is turned off.
- Checklists from which the user can make multiple selections, as in specifying print of file attributes,
- Support for a pointing device, typically a mouse (especially to select and drag screen elements)
- Scroll bars along the edges of windows to show the relative position of the contents (such as the end or beginning of the text) or to move to a different position (such as another part of a spreadsheet).

A GUI enforces consistency by restricting developers - they must support features for the program to run under the GUI. In addition, suggestions from the GUI's creators (such as the arrangement of menu options) often become de facto standards for applications.

We have described how consistency simplifies the learning of a new application. One benefit of a GUI is that the user can learn a second application faster because he or she is familiar with the environment. Note that the benefit comes with succeeding applications.

Consistency and familiarity help produce shorter learning curves. Users generally prefer the interface style they know, whether it be Macintosh, Microsoft Windows, or Lotus 1-2-3. The consistency offered by a GUI trades on the user's familiarity with environment.

Drawing and CAD programs are the best-suited to GUIs since, by their nature, they manipulate objects, lines and curves, and fill closed areas with colour. For database programs such manipulation is not as useful. However, they can use GUI's effectively to :

- Specify data field when setting up reports
- Select sort keys
- Transfer data to or from other applications (such as a spreadsheet).

The last point is particularly important. Database application often must transfer data to or from spreadsheets, word processors, desktop publishing programs business or presentation graphics programs, statistics programs, or project management software. GUIs generally have data exchange features, such as Microsoft Window's Dynamic Data Exchange (DDE), to handle such transfers.

Hewlett-Packard's New Wave extends the direct manipulation approach. The user can, for example, drag file icons to the printer without loading the applications that created them. New-Wave also handles the merging of application data with links that are transparent to the user. To add to the general confusion, note that New Wave is a GUI that runs under another GUI (Microsoft Windows).

A final benefit of a GUI is that it lets you see the final product before you print it. What You See Is What You Get (WYSIWYG) is a feature essential to desktop publishing and drawing applications, and useful in database application (so you can inspect reports to see that all data fits on the page).

However, there are drawbacks associated with using GUI. The costs include the expense of graphics cards, pointing devices (such as mice), and extra memory. Because GUIs run in graphics mode, screen refresh is usually slower as well. If speed is important, a GUI's consistency may not be sufficient compensation.

---

### 3.7 USING THE METHODOLOGY

---

Very often, as compared to organisations with systems departments that cater only to in-house development requirements, many IT consultancy organisations have some kind of in-house standards and procedures in planning, designing and developing their clients' systems. The extent to which these 'methods' are formal may vary from organisation to organisation and be influenced by the environment in which the consultant operates (e.g., more formal if addressing the international market or if the consultant has some business tie-up with a multinational corporation). Some of these are proprietary and not available for public usage.

An illustrative list of some of the more popular methodologies includes offerings from: DeMarco, Gane and Sarson, James Martin (Information Engineering), Ken Orr Association (Data Structured Systems Development), LBMS (LSDM), Michael Jackson, N.C.C. (SSADM), Yourdon, etc.

The scope and complexity of these methodologies vary considerable depending on the objectives that the methodology seeks to achieve and the tools and techniques they prescribe. A methodology critic sees a parallel between the proliferation of methodologies and religious sects, and in his words, **Despite the fact that they are 95 per cent agreed in their aims and their broad areas of getting there they nevertheless manage to stay separate. Each sect religiously guards its own style and magic ingredients...**

Some of the areas in which current day methodologies are deficient are in their support to the testing and maintenance phases. Usual response to this are to adapt them or to dovetail them with in-house standards governing these areas. Sometimes, it is said that following a methodology too closely (with its insistence on formalising the whole process) may lead to development of 'systems of Yesterday'. Also, expecting miracles of Day 1 of introducing the methodology and believing in it as a panacea may lead to disillusionment.

As the saying goes, **A methodology does not replace a good System Analyst, it only makes a good Systems Analyst better.** The ultimate aim of an organisation is to build better systems, not to follow an excellent Methodology.'

---

### 3.8 CHOOSING THE RIGHT METHODOLOGY

---

Given the above scenario in the methodology market, how does one go about evaluating the various options, to decide which methodology is suitable for one's organisation? Well, there are no ready-made answers, and there is obviously no one 'best solution' that is suitable for all organisations and for all its projects. Depending on the problems faced by individual organisations and the setting in which they do business, the objectives in adopting a methodology may vary. Thus, what seems to work very well in one place may introduce more chaos in another. However, the following questions may be asked about each methodology before deciding on adopting it as a standard:

a. Scope and level of detail:

- Is it applicable to the various types of systems that your organisation builds (e.g., transaction-oriented, process-oriented, real time, small vs. Large)?
- What is its scope and what are its boundaries? Does it cover business planning, IT strategy, feasibility, analysis, design and maintenance phases? In what detail does it cover these?
- Does it facilitate cross-reference between products of various techniques (e.g., DFDs with ERDs).

b. Integration with other tools:

- Does it have enough flexibility to integrate prototyping?
- What is the data dictionary scheme it supports?

- What is the documentation standard it proposes? Is it easily created, referred and maintained?
  - What output of platforms does it provide to facilitate project management?
  - What does it recommend as quality assurance mechanisms?
- c. Ease of learning and use:
- What level of training does it require for use?
  - What is the support available from the vendor for training and consultancy while practicing the methodology?
  - What is the feedback from other users? Are there any user groups active?

---

### 3.9 IMPLEMENTING A METHODOLOGY

---

Recognizing the need for 'good' methodology and appreciating its benefits may just be the first step, and a lot of hard work still remains to be done before a methodology can be successfully introduced in an organisation.

Some of the important steps are:

- Study the type of application systems your organisation develops (Mix of transaction processing, decision support, on-line, batch, etc.).
- Understand the problems you face more often and which ones are of more concern to you.
- Look around the methodology market and select the one(s) which are right for you.
- Involve senior management in the whole process and educate the personnel directly or indirectly involved (systems staff and end-users),
- Train systems staff on the intricacies of the methodology.
- Apply the methodology on a pilot project (small, non-critical, low priority application system) to get the feel of it.
- Ensure you get feedback on its effectiveness.

Usual resistance to the introduction of a methodology is that it is looked at as 'Old wine in new bottles' and that it takes away the freedom and creativity in doing one's work. The path of the 'golden mean' has never been more apt.

#### **What Tools are Available for Development Software ?**

Just as many tools exist for building a house many tools are available for creating or writing software. These tools comprise different types of programming languages, each of which consists of a number of different commands that are used to describe the type of processing to be done, such as multiplying two numbers together. Software development tools can best be categorized as falling into one of five generations of programming languages. The languages in each successive generation represent an improvement over those of the prior generation—just as the electric saw was an improvement over the manual one. Languages of later generations are easier to learn than earlier ones, and they can produce results (software) more quickly and more reliably. But just as a builder might need to use a manual saw occasionally to cut a tricky corner, professional programmers still need to use early generation languages (except machine language, which we'll explain shortly) to create software. Each of the five language generations will be described in detail in this chapter.)

Compared with later generations, the early generation programming languages (first, second, and third) require the use of more complex vocabulary and syntax to write software; they are, therefore, used primarily by computer professionals. The term syntax refers to the precise rules and patterns required for the formation "of the

programming language sentences, or statements, that tell the computer what to do and how to do it. Programmers must use a language's syntax - just as you would use the rules of German, not French, grammar to communicate in German - to write a program in that language. Because more efficient software development tools are available, programmers do not create software using machine language anymore, and few use assembly language, except for programs with special processing requirements. However, third generation language are still in wide use today,

Fourth generation language still require the user to employ a specific syntax, but the syntax is easy to learn. In fact, fourth-generation programming languages are so much easier to use than those in prior generations that the non-computer professional can create software after about a day of training.

- 1) Natural Languages enabled. Processing Language currently under development — will constitute the fifth generation of languages. With this type of language, the user will be able to specify processing procedures using statements similar to idiomatic human speech - simple statements in English (or French, German, Japanese, and so on). The use of natural language will not require the user to learn a specific syntax.

In addition to the five generations of programming languages, some microcomputer software packages (such as electronic spreadsheet and database management software) are widely used for creating software. Although these packages generally cannot be categorized into one of the five generations, many people consider some of the database management systems software used on microcomputers, such as dBASE IV, to fall into the fourth generation category.

---

### 3.10 WHICH TOOLS ARE YOU MOST LIKELY TO USE?

---

The decision about which software development tool to use depends on what processing procedures you need to perform. Developing software is like building a house: The work will go much faster if you have a plan and the right tools. However, the tools have little value if you do not know how to use them; consequently, one of the most important steps towards effective and efficient software development is the selection of the right development tool.

As most of the Applications that exist in market today are developed using Third Generation Programming Language, the software that you buy off the shelf of a computer store has been created by a computer specialist using one of these languages. Also computer specialists need to know how to use these languages in order to update, or maintain, this existing software to accommodate new processing and output requirements.

For the user who is not a computer specialist the most popular tools for developing software will be the fourth-generation programming language and existing off the shelf software packages such as electronic spreadsheets and database management systems software, because one does not have to be an experienced computer professional to use them. The user who is working with these tools can create specialized software applications, such as keeping track of a company's expenses by department (a good application for a spreadsheet package), or maintaining a comprehensive customer file used in a clothing store for billing, marketing, and checking customer credit status (a good application for a database package).

---

### 3.11 CURRENT GENERATION OF SOFTWARE DEVELOPMENT TOOLS

---

Over the past 40 years, the programming languages used to develop software have been steadily improving in terms of ease of use, the time it takes to develop software, and the reliability of the finished product. Hence, we describe the major characteristics of current generation of languages, or software development tools, and pay special attention to the tools you will likely be using in the business environment.

### 3.11.1 Fourth Generation

Also known as very-high level languages, fourth-generation languages (4GLs) are as yet difficult to define, because they are defined differently by different vendors: sometimes these languages are tied to a software package produced by the vendor, such as a database management system. Basically, 4GLs are easier for programmers and user to handle than third-generation languages. Fourth-generation languages are non-procedural languages, so named because they allow programmers and user to specify what the computer is supposed to do without having to specify how the computer is supposed to do it, which, as you recall, must be done with third-generation, high-level (procedural) languages. Consequently, fourth-generation languages need approximately one tenth the number of statements that a high-level language needs to achieve the same result. Because they are so much easier to use than third-generation languages, fourth-generation languages allow users, or noncomputer professionals, to develop software. It is likely that, in the business environment, you will at some time use a fourth-generation language. Five basic type of language tools fall into the fourth-generation category: (1) query language, (2) report generators, (3) applications generators, (4) decision support systems and financial planning language and (5) some microcomputer applications software.

Query languages allow the user to ask questions about, or retrieve information from, database file by forming requests in normal human-language statements (such as English). Query languages do have a specific grammar, vocabulary, and syntax that must be mastered (like third-generation languages), but this is usually a simple task for both user and programmers. For example, a manager in charge of inventory may key in the following questions of a database:

How many items in inventory have a quantity-on-hand that's is less than the reorder point?

The query language will do the following to retrieve the information:

1. Copy the data for items with quantity-on-hand less than the reorder point into a temporary location in main memory.
2. Sort the data into order by inventory number.
3. Present the information on the video display screen (or printer).

The manager now has the information necessary to proceed with reordering certain low-stock items. The important thing to note is that the management did not have to specify how to get the job done, only what needed to be done. In other words, in our example, the user needed only to specify the questions, and the system automatically performed each of the three steps listed above.

Some query languages also allow the user to add data to and modify database files, which is identical to what database management systems software allows you to do. The difference between the definitions for query language and for database management systems software is so slight that most people consider the definitions to be the same.

Report generators are similar to query languages in that they allow users to ask questions of a database and retrieve information from it for a report (the output); however, in the case of a report generator, the user is unable to alter the contents of the database file. And with a report generator, the user has much greater control over what specify that the software automatically determine how the output should look or can create his or her own customized output reports using special report-generator command instructions. (Ordinary users may need the help of a computer specialist to use a report generator). In most reports, users require that a total or totals of one or more groups of numbers appear at the bottom. And, if more than one category of information is to be included in the report, the user usually wants subtotals to appear for each category. In the case of a third-generation language, the number of instructions necessary to create totals is about 10 times the number needed in a fourth-generation language because the programmer needs to specify not only what to total but how to total and where to place the total. Report generators have many built-in assumptions that relieve the user from having to make such tedious decisions.



Applications generators, as opposed to query languages and report generators which allow the user to specify only output-related processing tasks (and some input-related tasks, in the case of query languages), allow the user to reduce the time it takes to design an entire software application that accepts input, ensures data has been input accurately, performs complex calculations and processing logic, and output information in the form of reports. The user key info computer usable from the specifications for what the program is supposed to do. The resulting specification file is input to the applications generator, which determines how to perform the tasks and which then reduces the necessary instructions for software program. For example, a user like yourself could use an applications generator to design payroll runs-to calculate each employee's pay for a certain period and to output printed cheques. Again, as with query languages and report generators, the user does not have to specify how to get the processing tasks how to get the processing tasks performed.

Decision support systems and financial planning languages combine special interactive computer programs and some special hardware to allow high level managers to bring data and information together from different sources and manipulate it in new ways-to make projections, do what if analyses, and make long-term planning decisions. We correct fourth-generation software tools in more details.

Some microcomputer applications software can also be used to create specialized application-in other words, to create new software. Microcomputer software packages that fall into this category include many spreadsheet programs (such as Lotus 1-2-3), database managers (such as dBASE IV), and integrated packages (such as Symphony). For example, in a business without computers, to age accounts receivable (to penalize people with overdue account balances), some one has to manually calculate how many days have passed between invoice data and the current date and then calculate the appropriate penalty based on the balance due. This can take hours of work. However, with an electronic spreadsheet, in less than half an hour the user can create an application that will calculate accounts receivable automatically. And the application can be used over and over.

Another example of microcomputer software that is used to create new programs is HyperCard for the Macintosh-created by Bill Atkinson of Apple. IN general, this packages is a database management program that allows users to store, organize, and manipulate text and graphics, but it is also a programmable program that uses a new programming language called Hyper Talk to allow ordinary user to create customized software by following the authoring instruction that come with the package.

## 11.2 Fifth Generation

Natural languages represent the next step in the development of programming languages- fifth-generation languages. Natural language is similar to query languages, with one difference : it eliminates the need for the user or programmer to learn a specific vocabulary, grammar or syntax. The text of a natural-language statement very closely resemble human speech. In fact, one could word a statement in several ways- perhaps even misspelling some words or changing the order of the words- and get the same result. Natural language takes the user one step further away from having to deal directly and in detail with computer hardware and software. These languages are also designed to make to computer smarter- that is, to simulate the human learning process. Natural languages already available for microcomputer include Clout, Q & A and Savy retriever (for use with databases) and HAL (Human Access Language) for use with Lotus 1-2-3.

The use of natural language touches on expert systems, computerized collections of knowledge of many human experts in a given field, and artificial intelligence, independently smart computer system- two topics that are receiving much attention and development and will continue to do so in the future.

---

## 12 FOURTH GENERATION LANGUAGES

---

Along the road of computer history, one sees the evolution of computer technology in terms of both hardware and software. It can be traced back to the time of first generation computers which used vacuum tubes as basic components of internal

circuits. Then came the era of second generation computers the era of transistors. Since then a subsequent improvement in the basic design using integrated circuits and then using very large scale integrated circuits led to what was termed as the third and the fourth generations of computers. Computers of the fifth generation have also emerged from behind the academic curtains. The fifth generation computers are those which emulate artificial intelligence resembling human intelligence. This generation of computers represents a leap into knowledge processing compared to data and numerical processing carried out in the computers of all the previous generations.

Scanning the development phase, in the field of computers software, we see that the first generation software was very near to machine language coding. Since then, the following generation of languages have attempted to ease the effort which goes into programming. Second generation software was using a command language, e.g., the job control language (JCL) used in IBM 360 computers.

Third generation languages which are very commonly used include C, COBOL, Pascal and PL/I. We have entered into the era of fourth generation languages with languages like Focus, Ramis and Linc.

### 3.12.1 What is a 4GL?

Computer hardware has evolved through various generations as the vacuum tube gave way to transistors, then integrated circuits and subsequently to very large scale integrated circuits. Computer languages have also kept pace with this trend and have evolved over a period of time, from machine or first generation languages, featuring intricate combinations of 0s and 1s through assemble level languages (second generation) and the third generation COBOL, BASIC, etc., presently to the 4GLs.

Most application software that is available these days is the one written in third generation languages, like COBOL, BASIC and C. Till date, these languages, also called the higher level languages, have been used to solve any application demand whether suited for it or not. This results in a lot of unnecessary code which drains a great deal of time in programming activity and also increases the response time thereby causing a dip in the efficiency of the computer. So what was needed was a computer language which could do everything that a third generation language does but with much less effort. Hence what emerged was a fourth generation language.

A 4GL can be defined as a very high level computer language that enables rapid development of applications, sometimes without the help of information system (IS) professional, aiming to improve productivity in computer systems development and use.

A survey conducted to assess the gains of a 4GL showed that it required one-tenth of the time and effort to develop software using a 4GL. A 4GL is more of an application building language and has all encompassing syntax for every aspect of application building.

Trained manpower for software development is a scarce resource all over the world. While the hardware costs are dropping, the professional staff is becoming more expensive and harder to recruit and retain. Economic law dictates that there should be an effort towards replacing the more expensive resources by the less expensive ones. 4GLs provide a means to do so. 4GL significantly affects two major factors, i.e., effort and time.

Comparatively less effort is needed in designing applications using 4GL as these are generally very user-programmer-friendly. The programs written in 4GL have to be specified with what is required of the task and in what particular sequence it needs to be done. It requires much less expertise to write down the code compared to what is needed to program in a third generation language. Subsequently, much less effort is needed to debug and modify the programs.

Not only does 4GL let you build applications faster, it lets you run them faster too. These speeds have been achieved through the combination of automatic indexing, concise instruction set, abbreviated instruction, clear and well defined conditions- all together lead to improved programming productivity. Enhanced query optimizers and report generators go a long way in faster access information for the people who depend on it the most.

### 3.12.2 End-user Computing

Users themselves can get information out of computerized databases. This ensures that information systems can respond to end-users as per their needs. In that sense the system become demand driven and not supply driven. End users can successfully undertake complete development of single systems (less than 1 man year).

### 3.12.3 Prototyping

Usually a considerable amount of time is spent in arriving at the correct functional specifications. Over the years, a number of methodologies have been developed to help this process. 4GLs offer prototyping as a technique to reduce the time spent in the process. Application development in a prototyping environment proceeds as follow:

- Information system professionals quickly arrive at a prototype of the system.
- Users and information system professionals together review prototype and make changes till prototype is correct.
- Information system professional optimizes database design for programs.
- The critical program are rewritten for better throughout.

### 3.12.4 Non-procedural

A 4GL programmer writes a program specifying what needs to be done and the appropriate procedure to accomplish the task. This shift towards non-procedural programming de-skills the expertise required to write programs and also makes it simpler to modify the existing system.

---

## 3.13 CONSIDERATIONS IN APPLICATION DEVELOPMENT

---

Applications development is still a bottleneck in most organisations' effective use of, and satisfaction with, computers. For the purpose of discussion we are including all processes involved from conception of the application through its ongoing use as applications development. The considerations that apply in developing a suitable application and the problems associated with each consideration is given below.

1. The program must enable the user to do whatever he is doing currently as well as whatever may come up in the future which he has to do.
2. Most users expect the programs to be ready in a short time.
3. Programs must be close to error-free and errors, where found, must be solved in a short time.
4. Programs must be modified at short notice to take care of missed or new requirements.
5. Turnover in personnel or computer systems must not interrupt the running of programs.
6. Costs must be justifiable.
7. Some PC- based applications are meant for use by a senior manager or director of the company and are expected to be developed by a more or less trial-and-modification approach.

### 3.13.1 Problems in Application Development

1. This simple requirement is unfortunately not simple to accomplish as it necessitates the programmer to master the application before developing it. This requires time from the user and programmer, dialogue between the two, good communications skills on the part of both, and an effective recording of the understanding so all can later be up-to-date.

2. For the system to be better than the current one, innovation thinking is required and like most innovative ideas, they require considerable time for experimenting and check-out.
3. Trying to develop a system in a short time generally leads to tension and attempts at short cuts which affect the quality which, in turn, affects the schedule.
4. Programs typically deal with hundreds of abstract logical processes and to guarantee a 100 percent error-free performance is generally impossible. And, yet even a single error seems shocking. Probably, 60 per cent of the programming effort goes in trapping the 5 per cent errors and misunderstandings that enter the system. Testing programs is still more a skill than a science and accuracy is a function of the skill of the programmer, the complexity of the system, and the effort spent in testing.
5. Modifying programs is the most error-prone activity and gets tested the least because it is required soon, and testing the program fully takes a lot of time. And often, the part that is not modified malfunctions because of unanticipated interactions.
6. Programs being complex, to train someone else to take over a program requires substantial efforts, and with other schedules in the pipeline, as well as maintenance requirements, it is generally done only when a notice of termination of transfer is received. When someone leaves without adequate notice, it can be a disaster. Change of computer systems to incompatible languages is a huge project and must only be done with a significant budget, big expected gains, and a strong heart.
7. This, IS of course, the bottom-line and with the other problems involved, is a real challenge.
8. These MIS systems do not justify a rigorous Systems Analysis and Design approach and yet the quality is expected to be very high.

### 3.13.2 How 4 GLs Help to Solve Problems

For the purpose of comparison, LINC or MAPPER is treated as a representative 4GL and COBOL as a representative 3GL. The distinctive features of a 4 GL are:

1. They are much easier to learn and use.
2. They provide more powerful features, so fewer commands are required to accomplish the task at hand.
3. They provide convenient feedback on syntactic mistakes and enable the user to correct the same and continue the program.
4. They are being improved at a much faster rate than 3GLs.

The above features have, in a sense, introduced a revolutionary change in the programming scene. Some of the changes are explained below:

1. Many young professionals, who are not programmers, feel, and become competent enough to develop programs for their areas of interest. Due to their expertise in their applications, their programs meet, their requirements. As they represent both the user and the programmer, they are able to make better compromises between what must be handled manually and what must be handled programmatically. And, being primarily users, they are more user oriented and more cost-less and benefit-more oriented than a typical programmer. Thus, their programs are simpler to the necessary point, and easier to develop. This contributes to reducing many of the problems associated with applications.
2. Programmers spend less time in writing and testing programs due to the brevity of the commands. This has improved programmer productivity by about 2-3 times.
3. Ad-hoc information can be, comparatively, easily provided.

4. It is easier to develop models of the systems, i.e., prototyping developing models takes a fraction of the *effort* to develop the real thing and it helps to clarify the understanding of the application and thus serves a similar purpose as an architect's blueprint of a house that is to become a home.
5. The on-going improvement in 4GLs promise increasing productivity of programmers and computers.

### 3.13.3 Limitation of 4GLs

Is it necessary that the organisation using a 4GL for its database development will show a dramatic increase in the productivity? Will the use of some 4GLs bring about significant improvement in the organisation's performance? The answer might be a No. This is because our industry, overwhelmed by very conspicuous advantages, is offering 4GLs as a solution to all possible problems. Unfortunately, people are falling for this, overlooking that there is a substantial gap between where they are and where they want to be.

Let us view why 4GLs are not delivering the goods even though they have some very obvious benefits.

Probably the main reason for this is the not-so-good performance of many products. This might be because these products provide programmer productivity gains, but leave little scope for the designer or the analyst.

The approach of a 3GL in development was concentrated on the paper intensive technique which required a lot of time, effort, expertise and experience. 4GLs by contrast, support prototyping (which are working models of the system and can be changed quickly and easily) and methods of modifications which enable the user to be directly involved in the process of development - as the end user's requests can be accommodated and the change reflected in a very short span of time.

Secondly, the database management system supporting the 4GL might not be sound. For example, if a relational DBMS is being implemented, then it is required that the non-redundant tables should be used - which is not very often the case. Also the present 4GLs go in for no data dictionaries or passive data dictionaries which cannot maintain the data naming standards.

Another reason why the vendor claims of huge productivity gains from programming are not met is that not all the products claiming to be 4GL qualify to fall *under* this category. They can be carrying out one or more functions of a 4GL but lack in having all the characteristics of a 4GL. To list only a few, these products might be just query processors running against some file management system. They might be high level languages teamed with a DBMS, or they just might be COBOL code generators.

Hence to really get value for one's money spent, understanding of the differences and the gap between the old and new generation technologies is required.

Possibly, the biggest problem in using 4GLs is that experienced 3GL programmers have to invest considerable effort (about a few man-months) to master a new language. Also, since the installed programs in COBOL, BASIC, etc., are very many, and converting them to 4GLs is generally a prohibitively expensive effort, the new 4GL programs must also provide a mechanism for a full two-way transfer between 3GL and 4GL systems.

### 3.13.4 Impact of 4GLs

There are certain areas wherein the impact of 4th generation computer languages (4GLs) like Focus, Ramis, etc., deserve management attention.

1. Productivity and cost of software development.
2. Restructuring of the systems development process.
3. Increased emphasis on decision support systems (DSS) and end-user computing (EDU)

#### 4. Changes in the roles of users and systems professionals.

Let us study each in some detail.

##### Software Development

One major reason for the development of 4GLs is productivity. The 10-to-1 gain is now becoming visible, and though we see applications getting developed for implementation which a tenth of the man hours that it took with, for instance COBOL, it does not mean that cost has become one-tenth. The extra hardware resources required by 4GLs (more CPU cycles, additional memory, etc.) contribute towards some increase in cost, but in spite of this, 4GL solutions cost much less. And in the years to come, hardware cost will continue to plunge downwards while cost of each technical man-hour that is used for the development of software will more upwards rapidly.

Productivity increase takes place because of two characteristics of 4GLs. First, that every man hour of programming generates much more lines (COBOL equivalent) of program, and second, that the level of skills required to write programs in 4GLs is typically lower than that required to write programs in 3GLs like COBOL. The management implications are obvious. On the one hand, lower development cost means that more and more applications become cost effective for computerization and on the other there is a faster turnaround of applications because of which computerization can progress for more rapidly.

##### System Development Process

It begins with the generation of idea like why don't we evaluate the possibility of computerizing the maintenance planning activity? This type of thinking termed Conception, usually starts in the minds of senior managers, and leads to a process of evaluation in the form of a brief feasibility study. For this purpose, a task force comprising one or more people each from the computer section evaluate the idea from two viewpoints-the technical feasibility and economic viability. This stage which may be termed Initiation, is required because it is important to evaluate each attempt at developing systems. Cost can be prohibitive and application development skills are scarce. So rigid discipline (of ensuring that each idea is viable) needs to be introduced and no project should begin till this is done.

The analysis phase is carried out by the systems analyst once a clear go-ahead is obtained from the evaluation group. It ends with a very well defined set of deliverables, termed Functional Specification. The Design phase sees the system designer converting the functional specification into yet another set of highly defined parameters and guidelines that include file design, codification structure and program specification. Programming is the next phase and it all ends with a rigorous testing that concludes the development cycle.

It is a widely accepted fact (though not as widely practiced) that to keep things under control and in order, the sequence of phases in the Systems Development Cycle should be followed very rigidly. This means that the designer should not begin his work till the analyst has finished. And it should be accepted that Analysis is not over till such time as the user has signed and accepted the output formats to the very last detail. This also implies that once the user has committed himself to a set of output formats, it is wise not to seek changes till the system is settled and running smoothly since a good part of the subsequent activity (design, programming and testing) would have to be re-done.

One major reason attributed to the sizeable delays that usually take place in development is that the Systems Development Cycle has not been followed rigidly. The usual tendency is to take short cuts in analysis and then implement changes once the system is almost complete. This plays havoc with time-frames as well as the stability and reliability of the system, since changes introduced later can generate additional errors that may go undetected.

The impact of this phenomenon has been quoted often. If those changes that are sought after the system is operational had been identified earlier, through more intensive and thorough analysis in the Systems Analysis phase, then there would be an

almost hundred-fold saving of that time. In other words 100 hours spent in modifying a system that is operational could have been saved for every extra hour spent during analysis to ensure completeness and accuracy.

4GLs provide an environment where it is possible to iteratively and simultaneously carry out analysis, design and programming. This activity called prototyping needs skill as well as expertise and it is appropriate for systems of low to medium complexity. In prototyping an analyst identifies a part of the functional specifications and while he is carrying on with the remaining part of the analysis also carries out some part of the design (the in-built database packages in the 4GLs aid this process immensely). Programming in any case, is a simpler activity with 4GLs and is carried out concurrently for that part of the system which has gone through design. As a result of this a small, but perhaps the most important part, of the system is set up early and the end user can evaluate it for suitability before the remaining part is worked upon in greater detail. 4GLs provide inherent ease in polishing up the prototype using the embedded database coupled with powerful and easy-to-use interfaces.

Prototyping is no child's play, though today's software tools need major improvements to make this an easy process. Nonetheless the way in which 4GLs are evolving points to the distinct possibility of this being the dominant manner in which systems will be developed in the future.

#### Increased emphasis of DSS and EUC

While productivity increase is a major objective of 4GLs, an equally important expectation from 4GLs is their ease of use. On this count, all 4GLs are not equal but some even provide features that make it possible for end-users to create their own reports with just a few hours of training (not days and weeks).

This is accomplished through the provision of end-user interfaces and program generators. In Focus there is the revolutionary TALK technology that enables users to create complex reports as well as graphical outputs by interacting with the system through only the ARROW and ENTER keys. The process involves a series of interactions, where the computer screen shows a set of options and the user moves the cursor to the required option using the arrow key up, down or to the side - and then confirms the option by pressing the ENTER key. This throws up the next screen, and the process is repeated as the user moves the cursor and chooses an option.

This dialogue continues till the user has input the desired choices, at the end of which the report is generated.

This new found ability of the end-user to develop his own reports open up new vistas in computerization. Once the database is in place (Information System professionals could have played some role in the design of the database), the end-user is free to seek information in the manner and schedule that he wants it.

Ad hoc information retrieval is the first step in the process of development of more and more sophisticated Decision Support Systems. It is also the most commonly used application by top level managers. Gradually, we see the use of mathematical models and statistical techniques.

Leading 4GLs provide such tools in the form of building blocks and even these tools come with easy-to-use interfaces.

Relational database structure coupled with versatile modelling tools and front-ended powerful graphics as well as easy-to-use reporting functions - all of which are constituents of the more popular 4GLs - are making end-user computing and Decision Support Systems a reality that can be actualized by end-users themselves.

What it means from the management viewpoint is that now it is becoming possible to locate part of the software development responsibility to user departments directly.

#### Changing Roles

It becomes mandatory to train end-users adequately to cope with the new responsibilities.

Secondly, it is possible to draw up more ambitious computerization plans since a much larger number of people are directly working on the development of new systems.

Thirdly, top management needs to change the role of the IS professional from a developer of application to one of coach, guide and consultant to a large number of developers, i.e., end-users.

And finally, management needs to put in place a comprehensive set of policies, procedures, standards and guidelines (to be developed and monitored jointly with IS staff) that will enable end-users to develop profitable applications with standard software tools, using rigid consistent document standards across the organization.

### 3.13.5 What to Look for in a 4GL

The issues which go into making of a 4GL need to be carefully equated to have the greatest impact on the performance, resource usage and productivity. It is the 4GL design that extends or limits the ultimate degree of success in using it. Hence before entering into functional evaluation, one must understand how the 4GL is built and design Portability IS another aspect to ponder over. This is because it is quite likely that some time or the other a new version of the operating system might be installed or some change in the hardware configuration may be brought about. Hence questions about the present configuration and ease of migration to the new environment should be asked.

### LINC

Linc and information network computer is an Australian developed and supported example of a 4GL that is being used at more than 3000 sites around the world. Linc enables you to design, develop, modify and generate on-line applications systems. It consists of a definition language and an interactive computer which checks the validity coding as you input it and creates the program for the system. It allows the definition of the problem through a brief series of business-oriented, English-like statements which help minimise the errors and misinterpretations usually associated with complex programming languages. Errors in the code are highlighted and the corresponding explanation of the errors can be listed by giving an 'ERROR' command. This permits rapid development of the software code.

The Linc-interactive computer is a menu-assisted system which operates through an activities menu. The menu displays the activity modes which are used to create Linc systems, reports and networks. It is possible to access any one of them in any order. Each activity mode consists of one or more input screens through which one can define the appropriate part of the Linc system being developed, e.g., reports are defined through Report mode and networks through the Network mode.

Linc was developed for the business functions, hence its basic structure is business activity compatibility. A Linc system definition consists of three basic parts: components, events and profiles.

Events are business transactions that occur with this fixed data, e.g., receipt of goods sales, cash receipts and so on.

Profiles are used to specify the various ways in which the components and events in a system are to be combined, viewed and accessed. For example, profile on a component which deals in employee information could be a way to access or view the employee information in the ascending order of employee number. It might be in the order of employee name depending on the specific requirements.

Components and events were initially included in the Linc system by keeping the (master tile and transaction tile) idea of the business world in mind where components simulated a master file and events simulated the transaction file.

Creation: Using the Linc definition language and viewing the business activity in terms of components and events, the logic specification for these (components and events) can be declared.

Generation: Once the specifications are defined through the LDL (Linc Definition Language) the generation phase begins, in which all the applications system



specifications are established and maintained and the system made ready to use. This is done without any further programming activity. In this each functional aspect of programming code needed to create the system is generated automatically by Line. On this generated system it is possible to 'add', 'change', 'inquire', and 'delete' information that is present in the database.

Reports can be made to run and also a query session can be started using a DMS II inquiry

Reports: Reports are developed by the use of LIRC (Logic and Information Report Computer), which describes the function of each LIRC specification using LDL (Line Definition Language). LIRC is the reporting aspect of the Line system. Once a Line information system has been designed, LIRC may be used to design the reports required for accessing and presenting the information recorded in the system.

### Some Examples

An accountant who employs computers, developed the accounting system for his company alone in just two months (using the dBASE language in Clipper). He and his company are 'lay happy' with the system in terms of its productivity and reliability. Typically, an experienced non-accountant programmer would have taken over a man-year to develop the same application in COBOL, and possibly about 5 months in dBASE. Here, four productivity factors got compounded: dBASE is more productive than COBOL; an accountant programmer is more productive than a non-accountant programmer; user programmer is more productive than a non-user programmer; and dBASE enabled the other 3 productivity factors to be more practical. A programmer chartered accountant firm routinely turns out customized applications in almost the cost and time frame it takes to implement a packaged software. Probably, the reasons are: they are very competent, they quickly grasp user requirements, they program in the dBASE language, and they religiously try to keep their systems as simple as they can.

### Prospects for the future

Probably, for more programs are currently being developed in 4GLs than in 3GLs. This trend is expected to grow. As more and more managers are expected to use computers as they use their calculators and diaries, they will turn to 4GLs for meeting their information requirements. Also, improvements in 4GLs promise more power and ease to the users

There are several hundred products on the market that can claim to be 4GLs. The product can cover program generators, expert systems virtually anything that encourages an end-user to produce applications without resorting to conventional programming languages,

---

## 3.14 SUMMARY

---

4GLs promise easier and faster development of applications. Broadly, they cut down the number of lines of program code required and they provide a simplified approach to the design of programs, such that (in theory) the end-user is able to do the programming work for a particular task she requires.

4GLs have not, however, had the acceptance that the promotional hype would seem to warrant.

- They cater for the production of new applications; they do nothing for 'maintenance' programming - amending existing software to remove bugs or add function, - which is the bulk of programming work.
- Most DP departments and information centres spend a good deal of their resources on running systems for users, but 4GLs do nothing for the operations area,
- 4GLs exemplify the software equation that more capability demands more of a computer, 4GLs need a fast computer and a lot of memory.

- Programs written in 4GLs cannot mate well with existing software.
- There are no universally agreed upon standards and little real-world experience.
- 4GL may be relatively easy to learn, but the basic principles of good programming still apply. The end-user may not have the training or the professional background to follow them - error testing, maintainability so that someone else can take on the task of future amendment documentation so that others can use the program. That all takes skill; it can also take a good deal of time.
- The availability of 4GLs has encouraged end-users to utilize them in developing relatively trivial applications rather than the more sophisticated tasks for which they were intended. 4GLs can produce large clumsy program for such small jobs.

---

### 3.15 MODEL ANSWERS

---

#### Check Your Progress 1

1. Some advantages of CASE tools are: Integration of the activities of the SDLC, automatic standardization, guaranteed correctness and level of quality, removal of monotony, self-documentation, reduced cost and time of development etc.
2. Some disadvantages of CASE Tools could be : Reinforcement of the tendency of systems personnel to work only with the machine (and not with human beings, e.g., users in Analysis phase, or colleagues in Design phase); loss of data due to improper security or corruption, high cost for reasonable level of sophistication etc. There may be belief that the CASE tool is a panacea for all this.

#### Check Your Progress 2

1. Both Oracle and Ingres are Relational Database management systems. Their important features are:
  - Open architecture and distributed data management capability.
  - Support of standard query, SQL and 4GL tools like Reportwriter.
  - Support of screen-oriented development tools for painting entry screens.
  - Support of queer optimizer.
2. The future DBMS are designed to widen the applicability of database technology to new kind of applications. These applications include computer aided software engineering (CASE), mechanical and electrical computer aided design (CAD), computer aided manufacturing (CAM), scientific and medical applications, graphics representation office automation, knowledge representation for artificial intelligence and business applications where traditional DBMS have proven inadequate.
3. It is a programming paradigm which supports:
  - (i) Design of reusable software components using inheritance mechanism.
  - (ii) Design of code which can be easily maintained
  - (iii) Better conceptualization and modelling of real world phenomena.

---

# UNIT 4 CASE TOOLS

---

## Structure

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Software Crisis
- 4.3 What is Wrong with Current Development Methods?
  - 4.3.1 Software and its Increasing Cost
  - 4.3.2 Software Errors and their Impact
- 4.4 An Engineering Approach to Software
- 4.5 Why Case Fails?
- 4.6 Case Tools
  - 4.6.1 Generation of CASE Tools
  - 4.6.2 Categories of CASE Tools
  - 4.6.3 Selecting CASE Tools
  - 4.6.4 Deft CASE Tools
- 4.7 Factors Affecting Software Development
- 4.8 The Benefits of Using CASE
- 4.9 Summary
- 4.10 Model Answers

---

## 4.0 INTRODUCTION

---

Computer Aided Software Engineering (CASE) is the application of computers to assist in developing and maintaining software. CASE has been one of the most common uses of engineering workstations in the past decade.

CASE represents a comprehensive philosophy for modelling business, their activities and information system development. The CASE philosophy involves using the computer as a development tool to build models that describe the business, the business environment, and corporate planning, and to document computer system development from planning through implementation.

The complete picture of the CASE philosophy presents that specification for corporate plans, system design and system development become fully integrated. This occurs by sharing for the three functions of corporate planning, system analysis and design and system development across CASE component.

---

## 4.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss what is software crisis development
- List various stages of software development life cycle
- Discuss the development tools associated with different phases of software development life cycle.
- Classify categories of CASE tools.

---

## 4.2 SOFTWARE CRISIS

---

The software industry is facing a crisis today because of —

- increase in the size of the software packages
- increase in complexity of the problem areas
- project management/coordination problem due to increase in personnel requirements per project
- duplication of effort because most software packages are built manually, i.e., with no automation, no methodology for the most part.
- The increase in cost of software compared to hardware.

Large software projects (100 K lines of code) have become feasible for medium as well as large software organizations due to the falling cost of hardware systems. One of the major areas where the software industry has expanded is the area of specialised applications. The area is usually characterised by a high level of complexity. This has given rise to an increase in the use of formal methodologies for software development. However, these methodologies, for the most part, do not completely solve the problem of complexity.

Medium/large software projects require considerable manpower. This leads to management and coordination problems which in turn give rise to time and cost overruns. Typically in large software development efforts, the engineers working on different parts of the software package do not 'interact' very closely. This sometimes results in duplication of effort especially in the design and coding of small commonly used routines. It could also lead to problems in integration. As software development groups are becoming larger, it has been observed (1) that productivity per person reduces. On the other hand the cost of technically skilled personnel is rising. Most software packages built today have almost no flexibility or scope for extendibility. This contributes to higher maintenance costs. These are three of the factors which have contributed to rising software costs as opposed to falling hardware costs.

Advances in hardware technology and the lack of matching pace in software technology is a well known fact. Software continues to increase its share of the EDP budgets. Studies, surveys and symposia are being conducted the world over to pinpoint the concerns and issues so that they may be addressed.

1. Requirements analysis has been voted as the most troublesome phase of the life cycle in a recent survey. The basic problem is one of communication.
2. Maintenance is costly, entailing an expenditure amounting to at least 50 per cent of budgets.
3. There is a movement toward decentralization of control. End-user computing is being encouraged and being made effective and practical. There is still a long way to go before end users can really make a dent on the software applications backlog.
4. The applications backlog in the USA has been quoted to be between a staggering three to seven years.
5. The concept of **building software systems** as characterized by the waterfall life cycle is changing. Today, people are talking of **growing software systems** through **rapid prototyping** and manufacturing software systems through '**software factories**'. The basic problem is one of communication.

In another survey, most of the respondents recognized information technology as a real or potential source of competitive advantage. In India, with an increase in competition in the marketplace in general, information is soon going to a vital resource. The new system development technologies must address this issue. The top four concerns described by MIS executives were: Facilitating/managing end user computing, Translating information technology to a competitive advantage, having the top management understand needs and perspective of MIS, and measuring and improving MIS/DP effectiveness/productivity.

A third survey shows a 15 to 20 per cent average personnel turnover rate in the

country. Undocumented and unstructured systems become the nightmare of successors.

Another survey conducted by AT & T many years ago projected that, if the demand for telephones continued to grow, every soon every man, woman and child in USA would have to be a telephone operator. Of course, it did not happen. From surveys, the numbers have been extrapolated and by 2000 AD it is said that every man, woman and child in USA will have to be a programmer.

---

### 4.3 WHAT IS WRONG WITH CURRENT DEVELOPMENT METHODS?

---

'We have a wide variety of methods, and the life cycle curve is well-known, so what is going wrong? The computer press is full of articles on 'The Applications Backlog, 'The Maintenance Crisis' Skills Shortages and other related topics.

The analysis and programming tools were developed in an environment that was highly centralised. The machine and the DP (Data Processing) professionals were at a central site, often remote from the users. Also, there is a large commitment in terms of costs for the organisation which has such a site. (Salaries of the development and support staff, as well as machine costs). Therefore, it should be no surprise that this has led to development methods which do not offer sufficient flexibility, since one of the requirements for management was the increase in control over the projects offered by the tools and methodologies. Hence the adoption of the production environment which is different from a more flexible decentralized development environment.

The application backlog appears as a result of development teams spending the time allocated to development on maintenance and upgrading of existing systems. This in turn leads to a maintenance crisis, whereby systems that are coming live were designed in the past and hence were designed for past requirements. Modifications, sometimes called enhancements, therefore need to be added at a late stage in the development process. Systems were probably designed in such a way that end users were not considered at all. In fact they may have been the last people to be consulted.

The modern user has probably have brought up to use sophisticated systems, or have seen television programs that show them what is possible. Therefore, the expectations from any systems that is delivered have increased.

To achieve what the user wants is a skilled job; but there is still a shortage of skilled computer staff.

Another factor which has added to this crisis is the falling cost of hardware. This fall has been dramatic over the years, and it is now feasible for an organisation to have a personal computer in every office, if not on every desk, and to have many machines scattered about a site or many sites, rather than a central mainframe. The result is more problems, including lack of standardization of hardware and software, and the decentralisation of control as well as development. This leads to a conflict between existing methods for the centralized production of software to a fixed standard on a fixed machine, and the user who wishes to have more control and a quicker response on a decentralised set of machines. If a typical commercial project is 18 to 24 months in gestation, and due to the applications backlog this slips to around 48 months, it is not surprising that the requirements will alter during this time.

Often the delivered system is unreliable, its accuracy questionable and the functions offered are variants of the ones planned in the stages. The end result is that the user is frustrated or loses interest in the project. The cost of not getting it right is quite high, and the place where the errors or omissions occurred is significant.

Despite the advent of structured methods, (which not all development teams use) the major resource cost is the result of errors or omissions in the analysis phase of the project. Hence the three problems exacerbate each other, compounding the difficulties

of an already complex task. The solution lies in the use of automated analysis environments, structured programming methods and closer consultation with the end user.

Let us examine the cause of the software crisis more closely.

#### 4.3.1 Software and its Increasing Cost

Software costs increase because of the following reasons:

- Each application will need the generation of new programs;
- Purchasing of new equipment means that existing software will either have to be modified or rewritten;
- Programming is labour intensive and therefore strongly affected by inflation despite new techniques.

#### 4.3.2 Software Errors and their Impact

In the majority of current systems, the cost of testing and maintenance is around 40 per cent of the total spent. Some would go so far as to say it is 60 per cent. In any case it is a significant portion of the software budget. Indeed in certain case DP departments are so tied up with maintenance of current systems that no development can take place. The increasing costs of error correction can be identified with the following components:

- The increase in software complexity increase the testing complexity and the amount of testing that has to be done;
- Notification and communication of errors become widespread and more costly and there are frequent changes in documentation;
- Repeating tests that have been done before is costly. It should be possible to test the portions of code that are changed but often this is not the case;
- The project team will have been disbanded due to leavers and over commitments
- Inadequate or inaccurate requirements and needs used throughout the project due to lack of user involvement and consultation in the early stages of a development;
- The cost of maintaining a system is often underestimated at the outset of a project.

---

### 4.4 AN ENGINEERING APPROACH TO SOFTWARE

---

Computer Science is one of the most evolved technologies of the twentieth century. And, like any other rapidly evolving technology a necessary by-product of this one has been new techniques and terms. CASE or Computer Aided Software Engineering is among the most talked about of these techniques and terms.

We are aware that all forms of engineering originate from a science. Science provides the fundamental laws, technology uses these fundamental laws, and, engineering uses technology. When a technology is used for an engineering product the prime additional considerations become cost, ease of manufacture mass production, standardization and marketability.

It is interesting to note that Computer Software is also evolving from a science to an engineering discipline. The science of accessing information, computing, organizing information and applying logic to it developed into an implementable technology with the advent of programming languages. In the course of time, programming languages graduated from low-level Assembly languages to high-level languages like

FORTRAN, PASCAL, COBOL and C. This helped Software technology to gather momentum and proliferate. With the proliferation of this technology and the increase in its complexity, the need for mass production, cost reduction, and standardization was felt-which is why, today, we are talking about Software Engineering.

Software Engineering, or affordable production of complex software is becoming possible today, because we have the technology to build tools for Software Development. These CASE tools are based on techniques and methods that span the entire Software Development Life Cycle (SDLC), and are implementable on a computer. To get a good understanding of CASE methodology or CASE tools, it is important to appreciate the techniques and methods which have made CASE possible.

The important techniques and tools available at each stage of the SDLC are discussed and shown in figure 1.

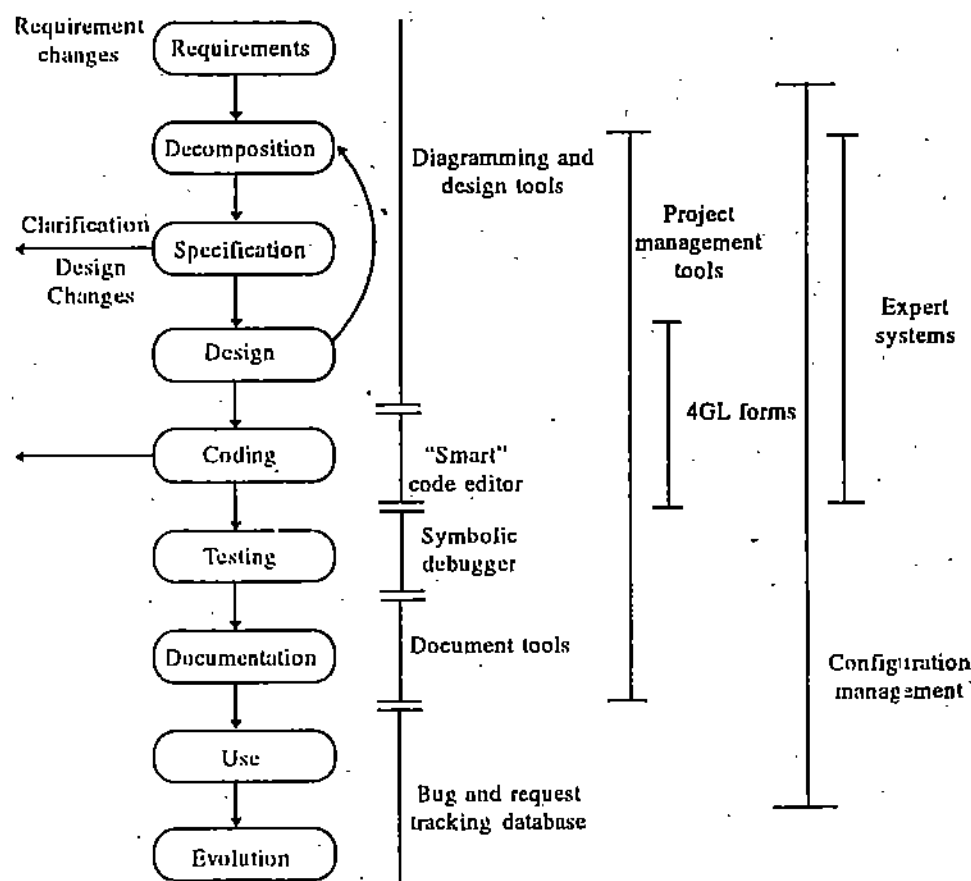


Figure 1: Typical Phases of the Software Life Cycle and Associated Development Tools

- a. **Feasibility Study:** This is the stage where the need for automation is felt, but its cost-benefit is not clear. A simple tools like spreadsheets can be used for carrying out a cost-benefit analysis. This tools may or may not be included in a CASE tools kit.
- b. **Requirements Study:** Once a green signal is obtained for automation, the first exercise to be undertaken is a requirements study. This study decides the automation boundary and specifies the requirements of the user in detail. The techniques used for requirements study are varied, but process modelling techniques like structured systems analysis and design (SSAD) and data modelling techniques like I-R modelling are popular. In both these techniques, a detailed data dictionary is prepared. CASE tools provide a means to document these techniques, and facilitate checking completeness and correctness.
- c. **Hardware Sizing and Capacity Planning:** The information collected during requirements study can be processed by a tools to arrive at hardware sizing. Since the tool needs to operate on the requirements study information, it is necessary for this tool to be integrated with the CASE tools for requirements study. Another

advantage of integrating sizing with study is prediction of capacity changes resulting from changes or enhancements to requirements.

- d. **Software Development Estimates:** The effort and duration for software design and development can be estimated using the information gathered for requirements study. Formal techniques like function point analysis (FPA) and construction cost model (COCOMO) are available for estimation. These techniques should ideally be integrated in a CASE tool.
- e. **Design:** The requirements study of a system identifies its inputs (screens), output (reports), inquiries, data dictionary and processing logic. During the design phase, these get translated to programs, libraries and files or databases. This translation can easily be aided by a CASE tool.
- f. **Software Development:** Once the design is complete, the code needs to be produced. Several tools are available for this. Fourth generation languages and program generators can be used to produce code. These often interface, or form an integral part of a CASE tool.
- g. **Testing and Quality Assurance:** Testing development software systematically, with every release, is essential. Tools for testing must allow easy preparation of test data, and automatic testing of the total system (on-line and batch) with error reporting and automatic recovery from error. Other tools like the Test Coverage Analyser (TCA) are also useful.
- h. **Implementation:** Implementation of developed software involves loading or distributing developed software to user sites and, training user personnel. Tools are available for loading and distributing software. Tools are also available for building tutorials and other forms of training material.
- i. **Maintenance:** Released and operational software requires maintenance, for error correction and enhancements. Traditionally it has been found that the highest costs are associated with this phase, and fortunately for us, it is this stage that the CASE tool addresses best. The documentation facilitated by proper usage of CASE tools from analysis through implementation becomes vital for maintenance. Besides, the information repository makes tools for Impact Analysis, Version Control and Amendment log possible.
- j. **Project Management:** Project Management spans the entire life cycle of a project. A tool for effective Project Management is essential for the success of a large project. The Project Management tool could very well be integrated in a CASE tool kit, because much of the information required for Project Management is available at the various stages of SDLC.

Some tools that can be used for Project Management are:

1. **Network and Bar chart Drawing:** A tool for drawing PDM (or ADM) network and bar charts can be effectively used for planning time, cost and resources. Such a tool facilitates updating changes to plans, and maintaining records of variance between plan and actuals.
2. **Skills Inventory:** A skills inventory system can be very useful for selection of suitable manpower for a project.
3. **Project Costing:** We have seen earlier how estimation can be integrated into CASE. With these estimates available, and with manpower allocation using a skills inventory system, manpower costing can be easily automated. Costing of other resources can be obtained from the network drawing tool. Any records maintained here can be very useful for sizing and estimating future projects.
4. **Software Metrics:** A tool for maintenance of software metrics can record errors detected at each stage of testing for future risk prediction.
5. **Quality Assurance Calendar:** A tool could help set up a quality assurance calendar (this could interface with the network drawing tool). This tool could also maintain QA review suggestions and recommendations.



CASE tools that address even a subset of the requirements motioned here are very useful. However, the ultimate CASE tools for quickly developed, easily maintainable, low-cost, standardized, quality software should address all aspects of SDLC.

---

## 4.5 WHY CASE FAILS?

---

Being aware of the most cited caused of CASE failure will, obviously, increase the chances of success. Among the most often cited reasons are:

- low management involvement
- unrealistic expectations
- no standards of methodologies already in place
- lack of integration with current practices.

Some other causes that are cited are:

- Misuse of CASE tools
- Too much emphasis on tools as total solution
- Ignoring the importance of management support and interaction
- Poor documentation (of tools)
- Not enough functionality
- Looking at CASE as a risk element.

There is a small and simple set of do's that are recommended. These are:

- Start slowly and do not invest very heavily on CASE tools.
- Begin with a few relatively small pilot projects.
- Get senior level management's blessings.
- Give the movement credibility by 'selling' the ideas within the company.
- Be sure to spend on training.
- Be patient.
- Make an analysis of needs and priorities and identify the tools for them.
- Start a metrics program. This will help quantify the benefits of CASE which would be critical when asking for more 'blessings' from the management.

---

## 4.6 CASE TOOLS

---

### 4.6.1 Generation of CASE Tools

There are two generation of CASE tools. The first generation CASE tools can be broadly classified into three groups; information generators or 4GL, front-end design/analysis tools applications tools and applications generators.

The variety of 4GL products includes the following; report generators, query languages, DBMS front ends and modelling languages. Most suffer from shortcomings: they are tied too closely to a proprietary database system thereby offering a very restricted solution; they are functionally too weak to be more than a building block in a larger application solution or they are not easily integrated with existing production system and data.

Design tools help a user to draw blue prints or design diagrams based on some pre-selected methodology. Typically high-level design documentation is provided automatically. The obvious flow in these tools is that they are standalone and their results are not easily integrated into the subsequent phases of the life cycle.

A major shortcoming of these first generation CASE products is their inability to bridge the gap between design and application generation. The second generation CASE tools evolved into two major categories: life cycle automation and solution software. The first category of tools is aimed at data processing profession to provide general solutions to their problems. The second category of tools is aimed at analyst or application specialist, in a restricted domain of application, to provide fast solution to the end user. Electronic spread sheets represent such a tools in the very restricted domain of financial analysis.

#### 4.6.2 Categories of CASE Tools

CASE tools might be classified into four broad categories according to the CASE problems on which they focus.

1. Front-end CASE tools, or Upper CASE tools:

These deal with the high-level design, specification and analysis of software and requirements. These tools include computer-aided diagramming tools oriented toward a particular programming design methodology, more recently including object-oriented design.

2. Back-end CASE tools, or Lower CASE tools:

These deal with the detailed design, coding, assembly, and testing of software. These tools may aid the programmer directly; for example, they include graphical debugging, aids and query and browsing facilities to find quickie a particular procedure or uses of a variable.

3. Maintenance Tools:

These deals with software after initial release. These tools may assist in tracking bug fixes and enhancement requests, porting to new platforms or performing new releases.

4. Support Software and Frameworks:

These provide basic functionality required in tools of type 1, 2 and 3. Support software includes basic operating-system functionality as well as higher-level support such as project management and scheduling software, and database support to track different version and configurations of software releases. Various projects have been directed toward standardizing frameworks to support and integrate CASE applications.

In addition to CASE methodology based on traditional programming languages and tools, there are two quite different approaches to CASE, particularly for back-end tools:

1. Higher-level languages and packages:

Some commercial products have focused on specific applications. For example, there are dozens of fourth-generation languages (4GLs), forms packages, and database design tools oriented toward the large market for business database applications on character terminals. Some more recent products are targeted at simplifying development of user interfaces in window systems. It is possible to make large gains in application programming productivity by focusing on a single application area.

2. Expert Systems

The application of artificial intelligence to programming, to select designs and produce code automatically in limited domains in another approach. So far, this

approach has seen limited application-sufficiently general automatic programming is very difficult to do.

### 4.6.3 Selecting Case Tools

This brings us to the onerous task of identifying, evaluating and selecting CASE tools. There are currently more than 200 CASE tools available in USA alone. The prices range from \$ 100 to ones which sell for upwards of \$ 300,000. In efficient markets, the price is a good proxy for functionality. The incredible variance in the type and functionality of tools which fall under the banner of CASE can thus be gauged. Selecting one from among them is not a trivial risk. If maximizing performance per unit is the objective then the obvious choice is Turbo Analyst. Telco's CASE offering Turbo Analyst priced at only Rs 20,000 is a low risk acquisition for organisations not yet convinced of the efficacy of the technology.

As already mentioned there are many other tools to choose from. Any comprehensive evaluation and selection methodology must pay due attention to two things.

Developing a menu of features and facilities that tools offer—a checklist of features/facilities/functionality that one desires for one's environment.

Importance must be given to each feature so as to facilitate a computation of scores for each short listed tool. The best is selected subject to the budget constraints. One such methodology has been developed by P-CUBE Corp., USA. We should adapt this to our context and include certain factors like after-sales services, support and training, foreign exchanges restrictions, duties, interfaces to packages/languages common in India, etc.

After all this evaluation, selection and implementation, one notices a drop in productivity. Have patience. Because CASE manages to improve long term productivity and quality significantly, once the learning curve has been taken care of.

### 4.6.4 Deft Case Tools

DEFT supplies Computer Aided System Engineering (CASE) products to engineers who work with Relational Data Base Management Systems. RDBMS engineers use the Deft CASE systems to:

- Assist them in gathering the initial requirements from end-user.
- Analyze these requirements and determine their feasibility
- Design the system's general algorithms
- Design an actual detailed implementation in terms of the target environment (hardware and operating system, specific RDBMS, etc.)
- Check their designs for completeness and consistency, and for contravention of specific RDBMS naming conventions.
- Automatically generate the RDBMS (tables, indices and forms) from the design
- Maintain their existing systems by reverse engineering the original databases from their host machines to Deft
- Control their development efforts through the medium of our configuration management tools.

#### The Deft CASE System

Deft CASE system consists of both tools and methods. Deft is flexible—it allows you the luxury of using either your own methods or The Deft Way is easy to use and simple methodology. In either case, your projects will run on-time and within budget, predictably and consistently.

#### The Deft Way

CASE methods employ the structured approach to software engineering and comprise various methods in which one draws diagrams or models of the computer system to be

built. The models each portray a certain aspect of the system, with four views required to adequately model a system that uses an RDBMS as the data repository. These four views are:

- **Data Flow Diagrams (DFDs):** These show the path of the data from one process to another and from and to the users of the system. The diagram represents the processes to be performed and identifies the data itself.
- **Entity Relationship Diagrams (ERDs):** These show the relationships of the various data entities to each other. With Deft's approach you can model not only your logical analysis, but also the physical database design itself, since Deft allows you to define key or index structures right in the model.
- **Program Structure Diagrams (PSDs):** These describe the logic or business rules involved in the processes. PSDs can be used to depict pseudo code for either 3GL or 4GL programs, and to break a module into subroutines or functions in addition to graphically showing the main procedures.
- **Form/Report templates and prototypes** complete the views required to model the target system. Deft provides a tool that allows you to rapidly create these forms or report templates. Using Deft's Gateway products, you can actually create these templates on your host machine within your RDBMS environment.

---

## 4.7 FACTORS AFFECTING SOFTWARE DEVELOPMENT

---

### Main factors

- The people that are to develop the product
- The work environment in which they develop the software
- The methodologies and tools that they use
- The need to produce quality software

### Subsidiary factors

- The politics of the organisation
- The need for experimentation and error
- The appointment of the correct person as development controller
- The psychology of the team members
- The need for standardisation.

---

## 4.8 THE BENEFITS OF USING CASE

---

The benefits of upper CASE are more direct if you usually perform corporate planning. By using an upper CASE system to build an enterprise model, you gain greater insight into the importance of certain functions and how the activities they control affect the entire organisation. You can better understand—

1. corporate and departmental mechanisms and responsibilities;
2. the goals of the company and its departments;
3. the influence of operations on achieving these goals;
4. their place within corporate and departmental administration and operations;
5. the timeliness and sequence of operations;

- factors influencing operations and goal achievement;
  - allocation of resources in support of operations;
  - the effect of external influences of the organisation;
  - problems facing the organisation; and
0. the importance of information relative to the success of the organisation.

Check your Progress

What is a SDLC ? List different phases of SDLC.

.....  
.....  
.....  
.....  
.....  
.....

What is a major shortcoming of the first generation CASE tools?

.....  
.....  
.....  
.....  
.....  
.....

What is the difference between upper CASE tools and lower CASE tools?

.....  
.....  
.....  
.....  
.....  
.....

---

## 9 SUMMARY

---

The purpose of this unit was to provide a broad perspective of the emerging CASE world. With software expenditure skyrocketing CASE has become a competitive edge for both major corporations and nations. Considering the fact that some countries spend around 30 to 40 billion on software, a 50% reduction in cost means billions of savings each year.

Currently CASE products are classified into upper CASE and lower CASE. Upper CASE tools support the front end of development life cycle; lower CASE tools support the back end of development life cycle. Many of the tools are designed to support a particular methodology. One approach is to integrate these tools to cover more of the development life cycle. Another is the development of a CASE shell, which is an environment that provides advanced facilities for the user to build his/her own tools.

---

## 4.10 MODEL ANSWERS

---

1. A SDLC is a full life cycle of a software project which goes through different stages: Requirements, Decomposition, Specification, Design, Coding, Testing, Documentation etc.
2. A major shortcoming of the first generation CASE products is their inability to bridge the gap between design and application programs.
3. The difference between the two is that the first deals with high level Design, Specification and analysis of S/W and requirement whereas the second deals with the detailed design, Coding, Assembly and Testing of Software.

