MCA - III SEM

SOFTWARE ENGINEERING

मध्यप्रदेश भोज (मुक्त) विश्वविद्यालय – भोपाल

MADHYA PRADESH BHOJ (OPEN) UNIVERSITY - BHOPAL

COURSE WRITERS

Rohit Khurana, Faculty and Head, I.T.L. Education Solutions Ltd., New Delhi

Units (1, 2, 3, 4.0-4.2, 4.3, 4.4-4.8, 4.10-4.14, 5)

Ravinder Kumar Arora, Professor (Finance and Accounting), International Management Institute, New Delhi

Unit (4.9)

Published by Registrar, MP Bhoj (Open) University, Bhopal in 2020

VIKAS®

Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

SYLLABI-BOOK MAPPING TABLE

Software Engineering

Syllabi	Mapping in Book
UNIT: 1 **Introduction to Software and Software Engineering:** The Origin of Software Engineering, characteristics of Software Engineering, Software Crisis, Software Engineering: Models Life Cycle Model, Spiral Model, Models of the Software Process.	**Unit-1:** Introduction to Software and Software Engineering **(Pages 3-29)**
UNIT: 2 **Software Engineering Methodologies:** Software Process, Software Metrics, Configuration Management Issues: Organizing the Process.	**Unit-2:** Software Engineering Methodologies and Configuration Management Issues **(Pages 31-42)**
UNIT: 3 **Software Requirement Analysis and Specification:** Requirements Definition Nonfunctional Requirements Definitions Formal Specification, Algebraic Specification, Model-based Specification, Z Schemas, Specification using Functions, specification using Sequences, Validation, The Prototyping Process, Prototyping Techniques.	**Unit-3:** Software Requirement Analysis and Specification **(Pages 43-71)**
UNIT: 4 **Principles of Software Project Management:** Principles of Software Project Management, Principles or Laws of Project Management, Software Project and Personnel Planning, Cost Estimation of Building a System, Software Metrics, The Project Plan, Resource Tracking and Stimulation Example, Quality Assurance Planning, Risk Analysis.	**Unit-4:** Principles of Software Project Management **(Pages 73-139)**
UNIT: 5 **Software Design:** Top-Down Design, Systems Design, Design Decomposition, Software Design, Quality, Design Description Languages, User Interface Design, User Interface Design Objectives, Function-Oriented Design, Data-Flow Diagrams, Structure Charts, Data Dictionaries, Deriving Structure Charts, Design Example, Concurrent Systems Design, Object-Oriented Design Inappropriate object-oriented Design, Design Quality Assurance, Design Reviews, Design Quality Metrics User Interface Evaluation, Verification and Validation, The Output Interface.	**Unit-5:** Software Design **(Pages 141-199)**

CONTENTS

INTRODUCTION

NOTES

The Institute of Electrical and Electronic Engineers (IEEE) defines software as 'a collection of computer programs, procedures, rules, and associated documentation and data'. Software is responsible for managing, controlling and integrating the hardware components of a computer system in order to accomplish a specific task. It tells the computer what to do and how to do it. For example, software instructs the hardware on how to print a document, take input from the user and display the output.

In the early days of computers, when computer memory was small, the language consisted of binary and machine codes and programmers would develop codes that could be used in developing more than one software system. Thus, software was simple in nature and did not demand much creativity. However, as technology improved, the need to build bigger and more complex software systems that could meet the users' changing and growing requirements arose. Software development necessitated a team that could prepare detailed plans and designs, carry out testing, develop intuitive user interfaces and integrate all these activities into a larger system of computers, machines and people. This led to the emergence of the discipline known as software engineering.

The notion of software engineering was first proposed in 1968. Since then, software engineering has evolved as a full-fledged engineering discipline that is accepted as a field involving in-depth study and research. Software engineering methods and tools have been successfully implemented in various applications spread across different walks of life. Software engineering has been defined as a systematic approach to develop software within a specified time and budget. It provides methods to handle complexities in a software system and enables the development of reliable software systems that maximize productivity.

This book, *Software Engineering*, follows the SIM format wherein each Unit begins with an Introduction to the topic followed by an outline of the 'Objectives'. The detailed content is then presented in a simple and an organized manner, interspersed with 'Check Your Progress' questions to test the understanding of the students. A 'Summary' along with a list of 'Key Terms' and a set of 'Self-Assessment Questions and Exercises' is also provided at the end of each unit for effective recapitulation.

UNIT 1 INTRODUCTION TO SOFTWARE AND SOFTWARE ENGINEERING

Structure

1.0 INTRODUCTION

Earlier, the software was simple in nature and did not demand much creativity from the developer. However, as technology improved, a need arose to develop larger and complex software systems that could meet the users' changing and growing requirements. Software development necessitated the presence of a team, which could prepare detailed plans and designs, carry out testing, develop intuitive user interfaces and integrate all these activities into a system. This team led to the emergence of a discipline known as software engineering.

Software engineering provides methods to handle complexities in a software system and enables the development of reliable software systems, which maximize productivity. In addition to the technical aspects of software development, it also covers management activities, which include guiding the team, budgeting, preparing schedules, etc. The notion of software engineering was first proposed in 1968. Since then, software engineering has evolved as a full-fledged engineering discipline, which is accepted as a field involving in-depth study and research.

In this unit, you will study about the introduction of software and software engineering, origin of software engineering and software engineering models.

1.1 OBJECTIVES

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

- Understand the significance of software and software engineering
- Know about the origin of software engineering
- Explain the various software engineering models

1.2 INTRODUCTION TO SOFTWARE

Software can be defined as a collection of programs, documentation and operating procedures. Institute of Electrical and Electronic Engineers (IEEE) defines software as a 'collection of computer programs, procedures, rules and associated documentation and data.' Software possesses no mass, no volume and no colour, which makes it a non-degradable entity over a long period. Software does not wear out or get tired. According to the definition by IEEE, software is not just about programs but also includes all the associated documentation and data.

Software is responsible for managing, controlling and integrating the hardware components of a computer system and to accomplish any given specific task. Software instructs the computer about what is to be done in a specific task and how it is to be done. For example, software instructs the hardware how to print a document, take input from the user and display the output.

Computers need instructions to carry out the intended task. These instructions are given in the form of computer programs. Computer programs are written in computer programming languages, like C, C++, etc. A set of programs, which is specifically written to provide users a precise functionality like solving specific problem, is termed as software package. For example, an accounting software package helps users in performing accounting-related activities.

History or Emergence of Software Development

Only in the later half of the 1940s did software development come into being with the creation of the first stored-program computer at Cambridge ESAC. Programs were earlier created as binary machine instructions. However, this approach was considered slow. Not only was it cumbersome but it was extremely difficult for people to memorize complicated and long binary strings. For this, the notion of human-readable shorthand for designing programs was formed. Some of the important datelines in the history of software development are listed in Table 1.1.

Table 1.1 *History of Software Development*

Period	Description
1950s	Majority of the programmer's time was spent in correcting errors in the software. By the late 1950s, managing programs even with the aid of reusable subroutines was becoming uneconomical. Hence, research in the area of automatic programming began. Automatic programming allowed programmers to write programs in high-level language code, which was easy to read. This programming improved the productivity of the programmers and made program portable across hardware platforms.
1960s	Software was developed and marketed separately. Software was created for specific areas. This broke the trend of providing software free along with a hardware platform. Additionally, the productivity of the programmer was improved by hiding the internal details of operating systems with the use of abstract programming interfaces.

1970s	When structured design was developed, software development models, based on a more organic and evolutionary approach, were also introduced. These deviated from the waterfall-based methodologies for hardware engineering. Research was done on quantitative techniques for designing software. During this time, researchers started focussing on software design aimed at addressing the problems of developing complex software systems.
1980s	The foscus of research in software engineering shifted towards integrating designs and design processes into the larger context of software development process and management. In the late 1980s, a new design paradigm known as object-oriented modelling was introduced. It was possible for software engineers to model the problem domain as well as the solution domain within the programming languages using the OOPs technique.
1990s	Class/responsibilities/collaborators (CRC) cards, use case analysis and other such design techniques were used to augment object orientation. Modelling notations and methods from structured design were used in the object-oriented modelling methods. This included diagramming techniques, such as state transition diagrams and processing models.
Presently	The complexity of designing and developing large-scale software systems was managed using a multi-viewed design approach. This approach resulted in the development of UML or unified modelling language. This integrates modelling concepts and notations from various methodologies.

Software Characteristics

Different individuals judge software on different basis. This is because they are involved with the software in different ways and users want the software to perform according to their requirements. Similarly, developers involved in designing, coding and maintenance of the software evaluate the software by looking at the internal characteristics of the products, before delivering it to the user. Software characteristics are classified into six major components which are shown in Figure 1.1.

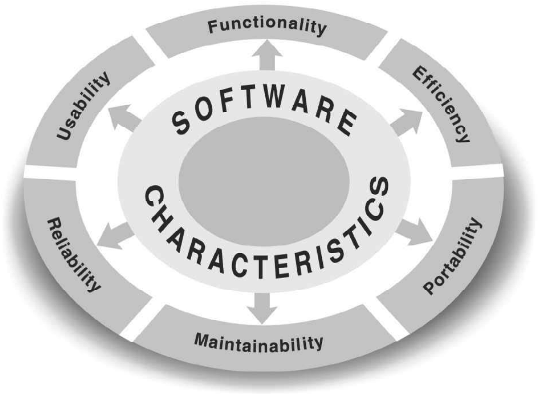


Fig. 1.1 *Characteristics of Software*

(i) Functionality: It refers to the degree of performance of the software against its intended purpose.

(ii) Reliability: It refers to the ability of software to execute the desired functions under the given conditions.

(iii) Usability: It means the degree to which the software is easy to use.

(iv) Efficiency: It portrays the ability of the software to use system resources in the most effective and efficient manner.

(v) Maintainability: It refers to the effort with which modifications can be made in a software system to extend functionalities, improve system performance or correct errors.

(vi) Portability: It deputes the ease with which the software can be transferred from one platform to another, without (or with minimum) changes. In simple terms, it is the ability of a software to function properly on different hardware and software platforms without making any changes in it.

In addition to the above-mentioned characteristics, robustness and integrity are also important. Robustness refers to the degree to which a software can keep on functioning in spite of providing invalid data, while integrity refers to the degree to which unauthorized access to the data can be prevented.

Classification of Software

Software can be applied in countless situations, such as in business, education, social sector and in other fields. The only thing that is required is a defined set of procedural steps. That is, software can be engaged in any field, which can be described in logical and related steps. Every software is designed to suit some specific goals. These goals are data processing, information sharing, communication and so on. Software is classified according to the range of potential applications. These classifications are listed as follows:

- **System software:** This class of software is responsible for managing and controlling operations of a computer system. System software is a group of programs rather than one program and is responsible for using computer resources efficiently and effectively. For example, operating system is system software, which controls the hardware, manages memory and multi-tasking functions and acts as an interface between applications programs and the computer.
- **Real-time software:** This class of software observes, analyses and controls real world events as they occur. Generally, a real-time system guarantees a response to an external event within a specified period of time. For example, real-time software is used for navigation in which the computer must react to a steady flow of new information without interruption. Most of the defence organizations all over the world use real time software to control their military hardware.
- **Business software:** This class of software is widely used in areas where the management and control of financial activities is of utmost importance. The fundamental component of a business system comprises of payroll, inventory, accounting and software that permits users to access relevant data from the database. These activities are usually performed with the help of specialized business software that facilitates efficient framework in the business operation and in management decisions.

- **Engineering and scientific software:** This class of software has emerged as a powerful tool to provide help in the research and development of next generation technology. Applications, such as study of celestial bodies, study of under-surface activities and programming of orbital path for space shuttle are heavily dependent on engineering and scientific software. This software is designed to perform precise calculations on complex numerical data that are obtained during real-time environment.
- **Artificial intelligence (AI) software:** This class of software is used where the problem solving technique is non-algorithmic in nature. The solutions of such problems are generally non-agreeable to computation or straightforward analysis. Instead, these problems require specific problem solving strategies that include expert system, pattern recognition and game playing techniques. In addition, it involves different kinds of searching techniques including the use of heuristics. The role of artificial intelligence software is to add certain degree of intelligence into the mechanical hardware to have the desired work done in an agile manner.
- **Web-based software:** This class of software acts as an interface between the user and the Internet. Data on the Internet can be in the form of text, audio or video format, linked with hyperlinks. Web browser is Web-based software that retrieves web pages from the Internet. The software incorporates executable instructions written in special scripting languages, such as CGI or ASP. Apart from providing navigation on the web, this software also supports additional features that are useful while surfing the Internet.
- **Personal computer (PC) software:** This class of software is used for official and personal use on daily basis. The personal computer software market has grown over the last two decades from normal text editor to word processor and from simple paintbrush to advance image-editing software. This software is used predominantly in almost every field, whether it is database management system, financial accounting package or a multimedia based software. It has emerged as a versatile tool for daily life applications.

Software can be also classified in terms of how closely software users or software purchasers are associated with the software development.

- **Commercial off the shelf (COTS):** In this category comes the software for which there is no committed user before it is put up for sale. The software users have less or no contact with the vendor during development. It is sold through retail stores or distributed electronically. This software includes commonly used programs, such as word processors, spreadsheets, games, income tax programs, as well as software development tools like software testing tools and object modelling tools.
- **Customized or bespoke:** In this classification, software is developed for a specific user, who is bound by some kind of formal contract. For example, software developed for an aircraft is usually done for a particular aircraft making company. They are not purchased 'off-the-shelf' like any word processing software.

NOTES

- **Customized COTS:** In this classification, user can enter into a contract with the software vendor to develop COTS product for a special purpose, that is, software can be customized according to the needs of the user. Another growing trend is the development of COTS software components which are purchased and used to develop new applications. The COTS software component vendors are essentially parts stores, which are classified according to their application types (see Figure 1.2). These types are listed as follows:
  - **(a) Stand-alone software:** It resides on a single computer and does not interact with any other software installed in a different computer.
  - **(b) Embedded software:** It is a part of unique application involving hardware like automobile controller.
  - **(c) Real-time software:** Operations execute within very short time limits, often microseconds. For example, radar software in air traffic control system.
  - **(d) Network software:** Software and its components interact across a network.

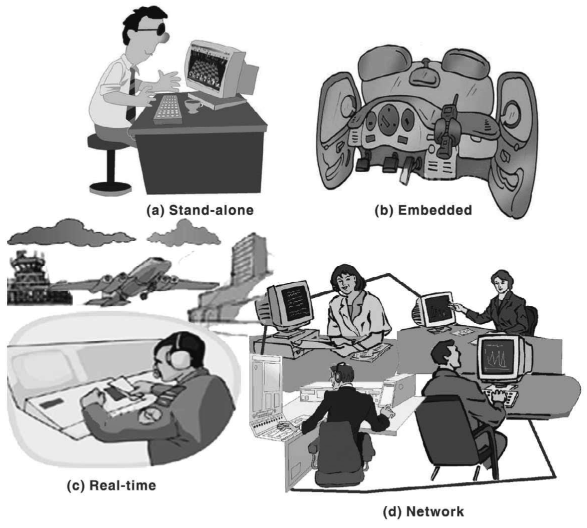


Fig. 1.2 *Types of Customized COTS*

Software Myths

The development of software requires dedication and understanding on the developers' part. Many software problems arise due to myths that are formed during the initial stages of software development. Unlike ancient folklore that often provides valuable lessons, software myths propagate false beliefs and confusion in the minds of management, users and developers.

Management Myths

Managers, who own software development responsibility, are often under strain and pressure to maintain a software budget, time slippage, improved quality and many other considerations. Common management myths are listed in Table 1.2.

Table 1.2 *Management Myths*

Myths	Realities
The members of an organization can acquire all the information they require from a manual, which contains standards, procedures and principles.	• Standards are rarely used. • Developers rarely know about them. • Standards are often out-of-date and incomplete.
State-of-the-art hardware is the essential ingredient for successful software production.	• Software tools are usually more important than hardware tools for achieving quality and productivity.
If the project is behind schedule, increasing the number of programmers can reduce the time gap.	• Adding more manpower to the project, which is already behind schedule, further delays the project. • New workers take longer to learn about project time as compared to those already working on the project.
If the project is outsourced to a third party, the management can relax and let the other firm develop software for them.	• Outsourcing software to a third party does not help the organization, which is incompetent in managing and controlling the software project internally. The organization invariably suffers when it outsources the software project.

In most cases, users tend to believe myths about the software because software managers and developers do not try to correct the false beliefs. These myths lead to false expectations and ultimately develop dissatisfaction among the users. Common user myths are listed in Table 1.3.

Table 1.3 *User Myths*

Myths	Realities
Brief requirement stated in the initial process is enough to start development, detailed requirement can be added at the later stages.	• Insufficient knowledge about requirements is the major cause of software failure. • Formal and detailed descriptions of data, functionality design constraints and validation criteria are essential. • Communication between user and developer is vital.
Software is flexible, hence, software requirement changes can be added during any phases of the development process.	• The impact of change varies according to the time when it is introduced. • Early requests for change can be accommodated easily and can be much cheaper. • Changes made during design, implementation and installation have a severe impact on cost.

Developer Myths

In the early days of software development, programming was viewed as an art but now software development has gradually become an engineering discipline. However, developers still have some myths (listed in Table 1.4).

Table 1.4 *Developer Myths*

Myths	Realities
Software development is considered complete when the code is delivered.	50 per cent to 70 per cent of all the efforts are expended after the software is delivered to the user.
The success of a software project depends on the quality of the product produced.	The success of a project does not depend only on the quality of programs. Documentation and software configuration are also essential.
Software engineering makes unnecessary documentation, which slows down the project.	Software engineering is about creating quality at every level of the software project. Enhanced quality results in reducing the amount of rework.
The only deliverable is the working program(s).	A working program is only one part of a software configuration, which includes requirements and specification documents, testing information and other developmental and maintenance information.
Software quality can be assessed only after the program is executed.	One of the most effective software quality assurance mechanisms is the formal technical review, which can be applied from the inception of the project.

1.2.1 Software Crisis

In the late 1960s, it was quite evident that software development was not like the construction of physical structures. This was because in software development, it was not possible to add more programmers merely to increase the speed of a development project which was lagging behind. Software had become an essential element of many systems. Yet, it was not possible to develop it with a certain schedule or quantity owing to its complex nature. As a result, the problem affected the safety, both financial and public.

Software errors have caused large-scale financial losses as well as inconvenience to many. Disasters, such as the Y2K problem, have affected economic, political and administrative systems of various countries around the world. This situation, where catastrophic failures have occurred, is known as software crisis.

Software crisis is a term that has been used since the early days of software engineering to describe the impact of the rapid increases in computer power and its complexity. Software crisis occurs due to problems associated with poor quality software. This includes problems arising from malfunctioning of software systems, inefficient development of software and most important, dissatisfaction among users of the software. Other problems associated with software are as follows:

- Software complexity can be managed by dividing the system into subsystems, but as systems grow, the interaction between subsystems increases non-linearly. This leads to a situation where the problem domain cannot be understood properly.
- It is difficult to establish an adequate and stable set of requirements for a software system. This is because hidden assumptions exist. In addition, there is no analytic procedure available for determining whether the

developers are aware of the user's requirements or not, thus creating an environment where both users and developers are unaware of the requirements.

The software market today has a turnover of millions of rupees. Out of this, approximately 30 per cent of software is used for personal computers and the remaining software is developed for specific users or organizations. Application areas, such as the banking sector are completely dependent on software application for their working. Software failures in these technology-oriented areas have led to considerable loss in terms of time, money and even human lives. History has been witness to many such failures, some of which are:

- During the Gulf War in 1991, United States of America used Patriot missiles as a defence against Iraqi Scud missiles. However, the Patriot failed to hit the Scud missile many times. As a result 28 US soldiers were killed in Dhahran, Saudi Arabia. An inquiry into the incident concluded that a small bug had resulted in the miscalculation of the missile path.
- The Arian-5 space rocket, developed at the cost of $7000 million over a period of 10 years was destroyed in 39 seconds, after its launch. The crash occurred because there was a software bug in the rocket guidance system.
- In June 1980, the North American Aerospace Defence Command (NORAD) reported that the US was under missile attack. The report was traced to a faulty computer circuit that generated incorrect signals. If the developers of the software responsible for processing these signals had taken into account the possibility that the circuit could fail, the false alert might not have occurred.
- The year 2000 (Y2K) problem refers to the widespread snags in processing dates after the year 2000. Seeds of the Y2K trouble were planted during 1960-80, when commercial computing was new and storing memory was relatively limited. The developers at that time shortened the 4-digit date format like 1994 to a 2-digit format, like 94. In the 1990s, experts began to realize this major shortcoming in the computer application and millions were spent to handle this problem.

1.3 SOFTWARE ENGINEERING

As discussed earlier, over the last fifty years there has been a dramatic advancement in the field of technology, leading to improvements in hardware performance and profound changes in computing architectures. This advancement has led to the production of complex computer-based systems that are capable of providing information in a wide variety of formats. The increase in computer power has made unrealistic computer applications a feasible proposition, marking the genesis of an era where software products are far more complex as compared to their predecessors. Using software engineering practices, these complex systems can be developed in a systematic and an efficient manner.

According to IEEE, software engineering is defined as 'the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.' In a nutshell, software engineering can be defined as a systematic approach to develop software within specified time and budget.

Software Engineering Layers

Software engineering can be viewed as a layered technology (see Figure 1.3). The various layers are listed as follows:

- **The process layer** allows the development of software on time. It defines an outline for a set of key process areas that must be acclaimed for effective delivery of software engineering technology.
- **The method layer** provides technical knowledge for developing software. This layer covers various activities which comprise requirements, analysis, design, program construction, testing and support phases of the software development.
- **The tools layer** provides computerised or semi-computerized support for the process and method layer. Sometimes tools are integrated in such a way that other tools can use information created by one tool. This multi-usage is commonly referred to as Computer-Aided Software Engineering (CASE). CASE combines software, hardware, and software engineering database to create software engineering analogous to computer-aided design (CAD) for hardware. CASE helps in application development including analysis, design, code generation, and debugging and testing. This is possible by using CASE tools, which provide automated methods for designing and documenting traditional-structure programming techniques.

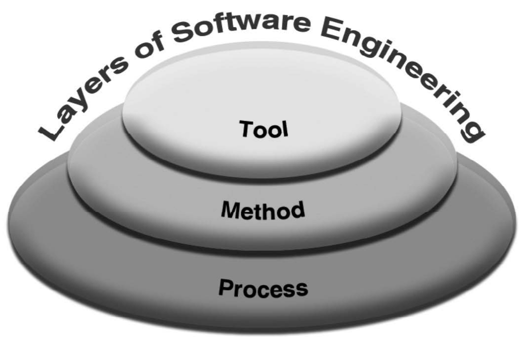


Fig. 1.3 *Layers of Software Engineering*

The main goal of software engineering is to help developers to obtain software of high-quality in minimum time at low cost. Therefore, like all other engineering disciplines, software engineering is also driven by three main parameters, namely, cost, schedule and quality. Among these, the cost and quality are the considered as the primary driving factors in software engineering because the schedule value is more or less fixed for a given cost.

A Generic View of Software Engineering

In general, software engineering means analysis, design, construction, verification and validation and the management of computer software. To engineer software adequately, three distinct phases, namely definition, development and support are performed.

The Definition Phase

This phase concentrates on 'what to do'. During this phase, the software engineer tries to determine answers to questions like what information is to be processed, what are the key requirements of the system, what functions the system is supposed to perform, what are the design constraints and validation criteria, etc. The major tasks that take place during this phase include system engineering, requirement analysis and software project planning and scheduling.

The Development Phase

This phase concentrates on 'how to do'. During this phase, the software engineers attempts to identify how to structure the data, how to implement the functions, how to establish the interfaces, how to translate the software requirements into a programming language, how to perform testing, etc. The major tasks that take place during this phase include software design, coding and testing.

The Support Phase

This phase concentrates on the changes related to enhancements made because of changes in users' requirements, fixing bugs and adaptations required due to change in environment. Generally, the following types of activities take place during this phase.

- **Corrective maintenance:** It is concerned with fixing reported errors in the software.
- **Adaptive maintenance:** It means changing the software to some new environment, such as adapting a new version of an operating system.
- **Perfective maintenance:** It involves implementing new functional or non-functional requirements.
- **Preventive maintenance:** It involves implementing changes to prevent occurrence of errors.

Principles of Software Engineering

Various principles of software engineering that must be considered while developing software are as follows:

- **Simplicity**: All systems should be designed as simple as possible. It facilitates easy understanding of the system, which helps in making changes simple and easy.
- **Practicality:** Software design should minimize the Intellectual distance between the software and problem existing in the real world. The system should be designed such that it always relates with the real-world problem.
- **Modularity:** A fundamental principle of software engineering for managing complexity is modularity which can be achieved by decomposing a large system into smaller and more manageable subunits. These subunits are called modules. The basic idea underlying modular design is to organize a complex system (large program) into a set of distinct components, which are developed independently and then are connected together. This may appear

as a simple idea; however, the effectiveness of the technique depends critically on the manner in which the systems are divided into components and the mechanisms used to connect components together.

- **Abstraction:** Abstraction is a mechanism to hide irrelevant details and represent only the essential features so that one can focus on important things at a time. IEEE defines abstraction as 'a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information'. For example, while driving a car, a driver only knows the essential features to drive a car, such as how to use clutch, brake, accelerator, gears, steering, etc., and least bothers about the internal details of the car, such as motor, engine, wiring, etc. Abstraction allows managing complex systems by concentrating on the essential features only.
- **Reusability**: The system should be designed in such a way that the existing design can be effectively reused. Reuse saves both time and efforts, thus increasesing productivity.

Check Your Progress

1. Give the definition of software according to IEEE.
2. What is software crisis?
3. Define the term software engineering.

1.4 THE ORIGIN OF SOFTWARE ENGINEERING

As discussed earlier, the discipline of software engineering is the result of advancement in the field of technology. In this section, we will discuss the various innovations and technologies that led to the emergence of software engineering discipline.

Early Computer Programming

As we know that in early 1950s, computers were slow and expensive. Though the programs at that time were very small in size, these computers took considerable time to process them. They relied on assembly language which was specific to computer architecture. Thus, developing a program required much effort. Every programmer used his own style to develop the programs.

High-Level Language Programming

With the introduction of semiconductor technology, the computers became smaller, faster, cheaper and reliable than the predecessors. One of the major developments includes the progress from assembly language to high-level languages. Early high-level programming languages such as COBOL and FORTRAN came into existence. As a result, the programming became easier and thus, increased the productivity of the programmers. However, still the programs were limited in size and the programmers developed programs using their own style and experience.

Control Flow-Based Design

With the advent of powerful machines and high-level languages, the usage of computers grew rapidly. In addition, the nature of programs also changed from simple to complex. The increased size and the complexity could not be managed by individual style. It was analysed that clarity of control flow (the sequence in which the program's instructions are executed) is of great importance. To help the programmer to design programs having good control flow structure, flow charting technique was developed. In flowcharting technique, the algorithm is represented using flow charts. A flowchart is a graphical representation of an algorithm where sequence of steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows. The boxes represent operations and the arrows represent the sequence in which the operations are implemented. A flowchart helps to clarify how things are currently working and how they could be improved. It also assists in finding the key elements of a process by drawing clear lines between where one process ends and the next one begins. A sample flowchart is shown in Figure 1.4. This flowchart calculates and displays the tax applicable on the items purchased.

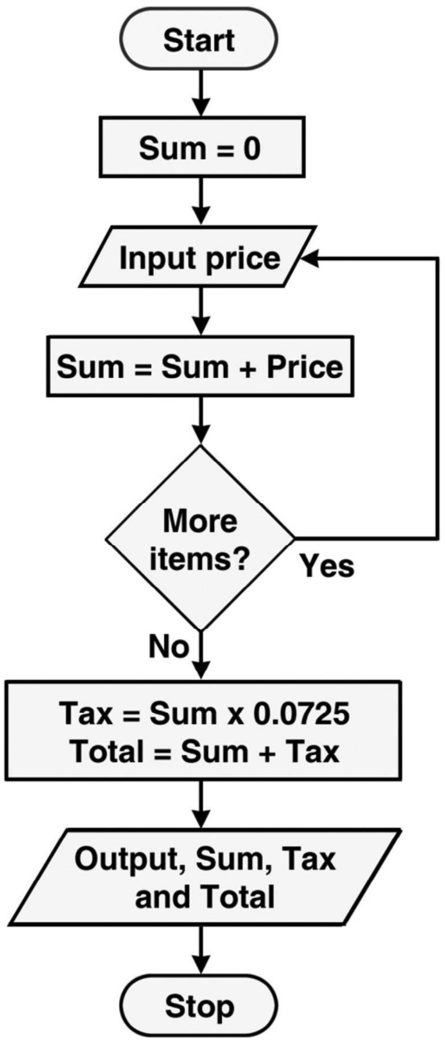


Fig. 1.4 *A Flowchart to Calculate Tax*

It was analysed that a program having more GOTO constructs has messy control flow structure, which makes it difficult to understand and debug. An efficient program can be developed using the constructs-decisions, sequences and loops. The decision structures are used for conditional execution of statements (for example, if statement). The sequence structures are used for the sequentially executed statements. The loop structures are used for performing some repetitive tasks in the program. Thus, the use of GOTO statements was restricted. This formed the basis of the structured programming methodology.

Structured programming became a powerful tool that allowed programmers to write moderately complex programs easily. Structured programming forces a logical structure in the program to be written in an efficient and understandable manner. The purpose of structured programming is to make the software code easy to modify when required. Some languages, such as Ada, Pascal and dBase are designed with features that implement the logical program structure in the software code. Primarily, the structured programming focuses on reducing and removing the following statements from the program:

- 'GO TO' statements
- 'Break' or 'Continue' outside the loops
- Multiple exit points to a function, procedure, or subroutine. For example, multiple 'Return' statements should not be used
- Multiple entry points to a function, procedure or a subroutine

Data-Flow-Oriented Design

With the introduction of Very Large Scale Integrated circuits (VLSI), the computers became more powerful and faster. As a result, various significant developments like networking, GUIs came into being. Clearly, the complexity of software could not be dealt using control flow based design. Thus, a new technique, namely, data flow-oriented technique came into existence. In this technique, the flow of data through business functions or processes is represented using data-flow diagram (DFD). IEEE defines a data-flow diagram (also known as bubble chart and work flow diagram) as 'a diagram that depicts data sources, data sinks, data storage and processes performed on data as nodes, and logical flow of data as links between the nodes'. Generally, DFDs are used as a design notation to represent architectural design (external design) and top-level internal design specifications. DFD allows the software development team to depict flow of data from one or more processes to another. In addition, DFD accomplishes the following objectives:

- It represents system data in hierarchical manner and with required level of detail.
- It depicts processes according to defined user requirements and software scope.

Note: Different processes in a DFD can be performed in parallel. Thus, a number of processes can be executed concurrently.

A DFD depicts the flow of data within a system and considers a system that transforms inputs into the required outputs. When there is complexity in a system, data needs to be transformed using various steps to produce an output. These steps are required to refine the information. The objective of DFD is to provide an overview of the transformations that occur in the input data within the system in order to produce an output.

DFD consists of four basic notations (symbols), which help to depict information in a system. These notations are rectangle, circle, open-ended rectangle and arrows. Rectangles represent external entities, that is, the source or destination of data within the system. Each external entity is identified with a meaningful and unique name. Circles represent processes that show transformation or manipulation of data within the system and open-ended rectangles represent data stores that indicate the place for storing information within the system. Arrows are used to represent data-flows that show the movement of data from its source to destination within the system (see Figure 1.5).

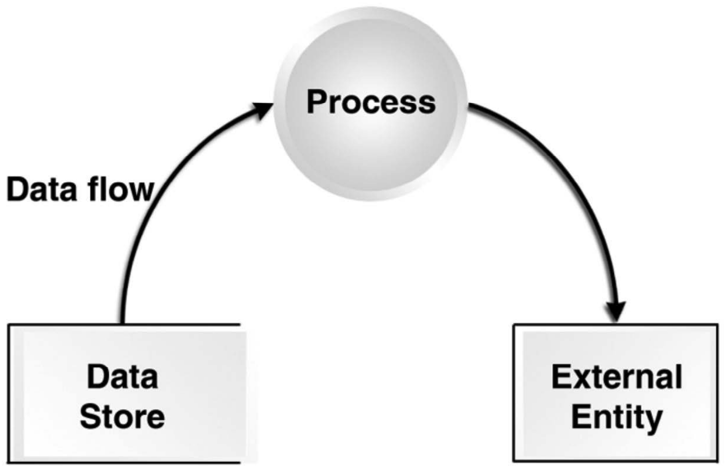


Fig. 1.5 *Data-Flow Diagram*

Object-Oriented Design

Object-Oriented design technique has revolutionized the process of software development. It includes the best features of structured programming but also some new and powerful features such as encapsulation, abstraction, inheritance, and polymorphism. These new features have tremendously helped in the development of well-designed high-quality software. Object-oriented techniques are widely used these days as they allow re-usability of code. They lead to faster software development and high-quality programs. Moreover, they are easier to adapt and scale, that is, large system can be created by assembling re-usable subsystems

1.5 SOFTWARE PROCESS MODELS

A process model can be defined as a strategy (also known as software engineering paradigm), comprising processes, methods, tools or steps for developing a software. These models provide a basis for controlling various activities required to develop and maintain the software. In addition, they help the software development team in facilitating and understanding the activities involved in the project.

A process model for software engineering depends on the nature and application of a software project. Thus, it is essential to define process models for each software project. IEEE defines a process model as 'a framework containing the processes, activities and tasks involved in the development, operation and maintenance of a software product, spanning the life of the system from the definition of its requirements to the termination of its use.' A process model reflects the goals of software development, such as developing a high-quality product and meeting the schedule on time. In addition, it provides a flexible framework for enhancing the processes. Other advantages of the software process model are:

- **It enables effective communication:** It enhances understanding and provides a specific basis for process execution.
- **It facilitates process reuse:** Since process development is a time-consuming and expensive activity, the software development team utilizes the existing processes for different projects.
- **It is effective:** Since process models can be used again and again, reusable processes provide an effective means for implementing processes for software development.
- **It facilitates process management:** Process models provide a framework for defining process status criteria and measures for software development. Thus, effective management is essential to provide a clear description of the plans for the software project.

Every software development process model takes requirements as input and delivers products as output. However, a process should detect defects in the phases in which they occur. This requires verification and validation (V&V) of the products after each and every phase of the software development life cycle (see Figure 1.6).

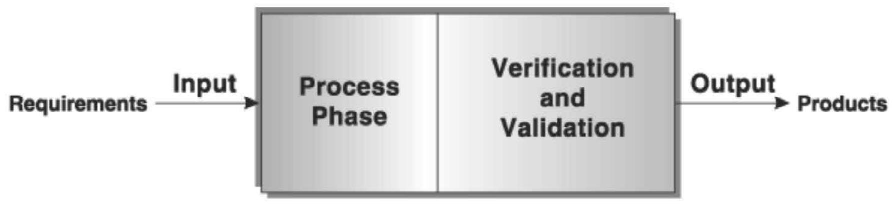


Fig. 1.6 *Phases in Development Process*

Verification is the process of evaluating a system or its components for determining the product developed at each phase of software development. IEEE defines verification as 'a process for determining whether the software products of an activity fulfil the requirements or conditions imposed on them in the previous activities.' Thus, it confirms that the product is transformed from one form to another as intended and with sufficient accuracy.

Validation is the process of evaluating the product at the end of each phase to check whether the requirements are fulfilled or not. In addition, it is the process of establishing a procedure and method, which performs according to the intended outputs. IEEE defines validation as 'a process for determining whether the requirements and the final, as-built system or software product fulfils its specific intended use.' Thus, validation substantiates the software functions with sufficient accuracy with respect to its requirement specifications.

The various kinds of process models used are waterfall model, prototyping model, spiral model and incremental model.

1.5.1 Waterfall Model

In the waterfall model (also known as the classical life cycle model or linear sequential model) the development of software proceeds linearly and sequentially from requirement analysis to design, coding, testing, integration, implementation and maintenance.

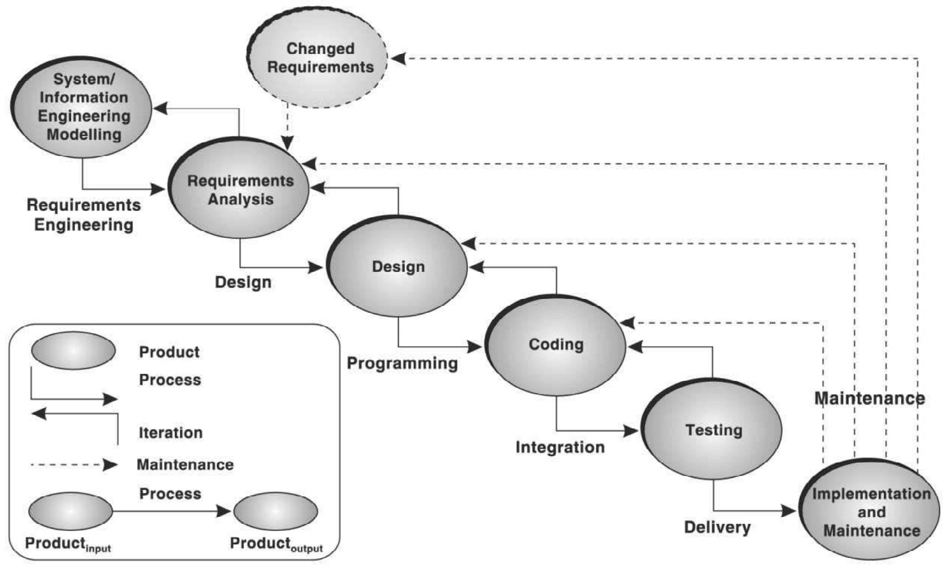


***Fig. 1.7** Waterfall Model*

This model is simple to understand and represents processes which are easy to manage and measure. The waterfall model comprises different phases and each phase has its distinct goal. After the completion of one phase , the development of software moves to the next phase (Figure 1.7). Each phase modifies the intermediate product to develop a new product as an output. The new product becomes the input of the next process as listed in Table 1.5.

***Table 1.5** Processes and Products of Waterfall Model*

Input to Phase	Process/Phase	Output of the Phase
Requirements defined through communication	Requirements analysis	Software requirements specification document
Software requirements specification document	Design	Design specification document
Design specification document	Coding	Executable software modules
Executable software modules	Testing	Integrated product
Integrated product	Implementation	Delivered software
Delivered software	Maintenance	Changed requirements

As stated earlier, the waterfall model comprises several phases. These phases are as follows:

- **System/information engineering modelling:** This phase establishes the requirements for the system known as computer-based system. Hence, it is essential to establish the requirement of that system. A subset of requirements is allocated to the software. The system view is essential when the software interacts with the hardware. System engineering includes collecting requirements at the system level. Information gathering is necessary when the requirements are collected at a level where all decisions regarding business strategies are taken.
- **Requirement analysis:** This focuses on the requirements of the software which is to be developed. It determines the processes that are incorporated during the development of a software. To specify the requirements, users' specification should be clearly understood and their requirements should be analyzed. This phase involves interaction between the user and the software engineer and produces a document known as software requirement specification (SRS).
- **Design:** This determines the detailed process of developing a software after the requirements have been analysed. It utilizes software requirements defined by the user and translates them into a software representation. In this phase, the emphasis is on finding a solution to the problems defined in the requirement analysis phase. The software engineer is mainly concerned with the data structure, algorithmic detail and interface representations.
- **Coding:** This emphasizes translation of design into a programming language using the coding style and guidelines. The programs created should be easy to read and understand. All the programs written are documented according to the specification.
- **Testing:** This ensures that the software is developed as per user requirements. Testing is performed to check whether the software is running efficiently, and with minimum errors. It focuses on the internal logic and external functions of the software and ensures that all the statements have been exercised (tested). However, one must note that testing is a multi-stage activity, which emphasises verification and validation of the software.
- **Implementation and maintenance:** This delivers fully functioning operational software to the user. Once the software is accepted and deployed at the user's end, various changes occur due to changes in the external environment (these include upgrading a new operating system or addition of a new peripheral device). The changes also occur due to changing requirements of the user and changes occurring in the field of technology. This phase focuses on modifying software, correcting errors and improving the performance of the software.

Various advantages associated with the waterfall model are as follows:

1. It is relatively simple to understand.
2. Each phase of development in the waterfall model proceeds in sequential order.

3. It allows managerial control where a schedule with deadlines is set for each stage of development.
4. It helps in controlling schedules, budgets, and documentation.

Various disadvantages associated with the waterfall model are as follows:

1. In the waterfall model, you need to specify the requirements before the development proceeds.
2. The changes of requirements in the later phases of the waterfall model cannot be done. This implies that once a software enters the testing phase, it becomes difficult to incorporate changes at such a late stage.
3. No user involvement and working version of the software is available when the software is developed.
4. It does not involve risk management.
5. It assumes that requirements are stable and are frozen across the project span.

1.5.2 Prototyping Model

The prototyping model is applied when detailed information related to input and output requirements of the system is not available. In this model, it is assumed that all the requirements may not be known at the starting of the development of a system. It is usually used when a system does not exist or in the case of a large and complex system where there is no manual process to determine the requirements. This model allows the user to interact and try out with a working model of the system known as prototype. A prototype gives the user an actual feel of the system.

At any stage, if the user is not satisfied with the prototype, it can be discarded and an entirely new system is developed. Generally, prototyping can be prepared by the following approaches:

- By creating main user interfaces without any substantial coding, so that users can get a feel of how the actual system will appear

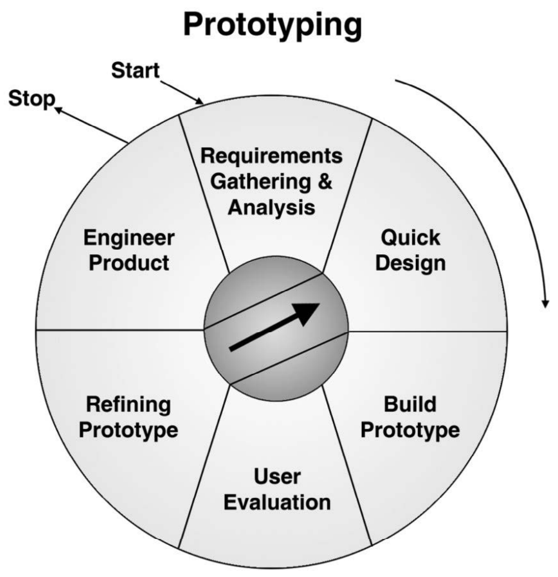


Fig. 1.8 *Prototyping Model*

- By abbreviating a version of the system that will perform limited subsets of functions
- By using system components to illustrate the functions that will be included in the system to be developed

Figure 1.8 illustrates the steps carried out in the prototyping model. These steps are explained as follows:

1. **Requirements gathering and analysis:** A prototyping model begins with requirements analysis and the requirements of the system are defined in detail. The user is interviewed in order to know the requirements of the system.
2. **Quick design:** When requirements are known, a preliminary design or quick design for the system is created. It is not a detailed design, however and includes the important aspects of the system, which gives an idea of the system to the user. A quick design helps in developing the prototype.
3. **Build prototype:** Information gathered from quick design is modified to form a first prototype, which represents a 'rough' design of the required system.
4. **Assessment or user evaluation:** Next, the proposed system is presented to the user for thorough evaluation of the prototype to recognize its strengths and weaknesses, such as what is to be added or removed. Comments and suggestions are collected from the users and are provided to the developer.
5. **Prototype refinement:** Once the user evaluates the prototype and is not satisfied, the current prototype is refined according to the requirements. That is, a new prototype is developed with the additional information provided by the user. The new prototype is evaluated just like the former prototype. This process continues until all the requirements specified by the user are met. Once the user is satisfied with the developed prototype, a final system is developed on the basis of final prototype.
6. **Engineer product:** Once the requirements are completely known, the user accepts the final prototype. The final system is evaluated thoroughly and followed up by routine maintenance on a regular basis for preventing failures and minimizing downtime.

Various advantages associated with prototyping model are as follows:

1. It provides a working model to the user early in the process, enabling early assessment and increasing user confidence.
2. The software developer gains experience and insight by developing a prototype, thereby resulting in better implementation of requirements.
3. It serves to clarify requirements, which are not clear, hence reducing ambiguity and improving communication between developer and user.
4. There is a great involvement of users in software development. Hence, the requirements of the users are met to the greatest extent.
5. It helps in reducing risks associated with the software.

Various disadvantages associated with prototyping model are as follows:

1. If the user is not satisfied by the developed prototype, then a new prototype is developed. This process goes on until a perfect prototype is developed. Thus, this model is time consuming and expensive.
2. The developer loses focus of the main goal of prototype, hence may compromise with the quality of the software. For example, developer may use some inefficient algorithms while developing the prototype.
3. Prototyping can lead to false expectations. For example, a situation may be created where the user believes that the development of the system is complete when it is not.
4. The main goal of prototyping is speedy development, thus, the system design can suffer as it is developed in series without considering integration of all the other components.

1.5.3 Spiral Model

In the 1980's Boehm introduced a process model known as the spiral model. The spiral model comprises activities organized in a spiral and has many cycles. This model combines the features of the prototyping model and waterfall model, and is advantageous for large, complex and expensive projects. The spiral model determines requirement problems in developing the prototypes. In addition, it guides and measures the need of risk management in each cycle of the spiral model. IEEE defines spiral model as 'a model of the software development process in which the constituent activities, typically requirements analysis, preliminary and detailed design, coding, integration, and testing, are performed iteratively until the software is complete.'

The objective of the spiral model is to emphasize management to evaluate and resolve risks in the software project. Different areas of risks in the software project are project overruns, changed requirements, loss of key project personnel, delay of necessary hardware, competition from other software developers, and technological breakthroughs, which make the project obsolete. Figure 1.9 shows the spiral model. The steps involved in the model are listed as follows:

Step I: Each cycle of the first quadrant commences with identifying the goals for that cycle. In addition, it determines other alternatives, which are possible in accomplishing those goals.

Step II: The next step in the cycle evaluates alternatives based on objectives and constraints. This process identifies the areas of uncertainty and focuses on significant sources of the project risks. Risk signifies that there is a possibility that the objectives of the project cannot be accomplished. If so, the formulation of a cost-effective strategy for resolving risks is followed. Figure 1.9 shows the strategy, which includes prototyping, simulation, benchmarking administrating user questionnaires or risk resolution technique.

Step III: The development of the software depends on remaining risks. The third quadrant develops the final software while considering the risks that can occur. Risk management considers the time and effort to be devoted to each project activity, such as planning, configuration management, quality assurance, verification and testing.

NOTES

Step IV: The last quadrant plans the next step and includes planning for the next prototype and thus, comprises of requirements plan, development plan, integration plan and test plan.

One of the key features of the spiral model is that each cycle is completed by a review conducted by the individuals or users. This includes the review of all the intermediate products, which are developed during the cycles. In addition, it includes the plan for next cycle and the resources required for the cycle.

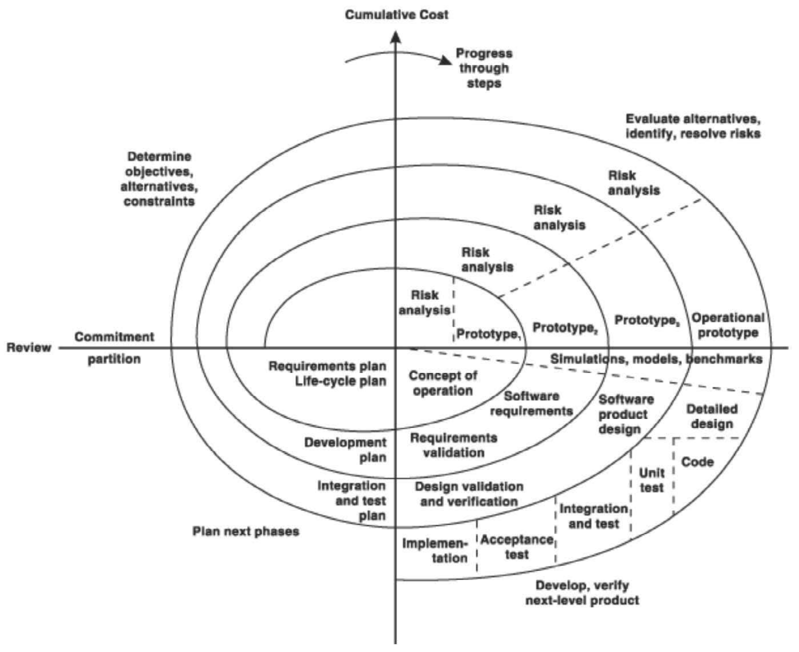


Fig. 1.9 Spiral Model

The spiral model is similar to the waterfall model, as software requirements are understood at the early stages in both the models. However, the major risks involved with developing the final software are resolved in the spiral model. When these issues are resolved, a detailed design of the software is developed. It is to be noted that processes in the waterfall model are followed by different cycles in the spiral model as shown in Figure 1.10.

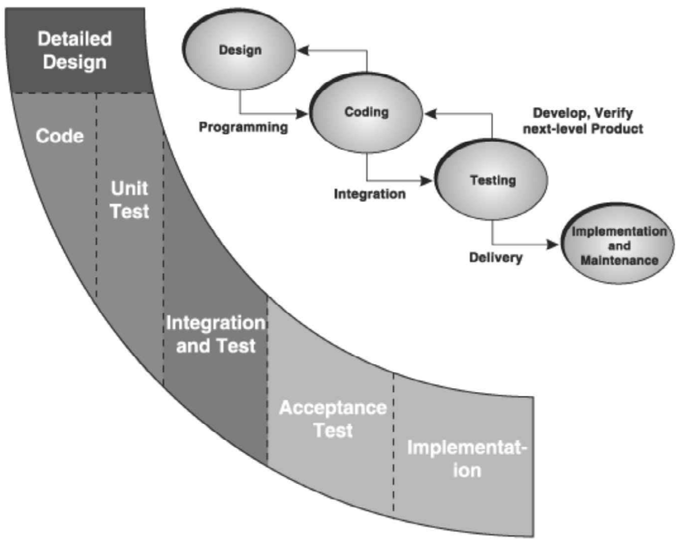


Fig. 1.10 *Spiral and Waterfall Model*

The spiral model is also similar to the prototyping process model as one of the key features of prototyping is to develop a prototype until the user requirements are accomplished. The second step of the spiral model functions similarly. The prototype is developed to clearly understand and achieve user requirements. If the user is not satisfied with the prototype, a new prototype known as operational prototype is developed.

Various advantages associated with the spiral model are:

1. It avoids the problems resulting in a risk-driven approach in the software.
2. It specifies a mechanism for software quality assurance activities.
3. It is utilized by complex and dynamic projects.
4. Re-evaluation after each step allows changes in user perspectives, technology advances or financial perspectives.
5. The estimation of budget and schedule gets realistic as the work progresses.

Various disadvantages associated with the spiral model are:

1. In the spiral model, assessment of project risks and its resolution is not an easy task.
2. It is difficult to estimate budget and schedule in the beginning, as some of the analysis is not done until the design of the software is developed.

1.5.4 Incremental Model

This process model comprises the features of waterfall model in an iterative manner. The waterfall model performs each phase for developing complete software whereas the incremental model has phases similar to the linear sequential model and has an iterative nature of prototyping. As shown in Figure 1.11, during the implementation phase, the project is divided into small sub-sets known as increments that are implemented individually. This model comprises several phases where each phase produces an increment. These increments are identified in the beginning of the development process and the entire process from requirements gathering to delivery of the product is carried out for each increment.

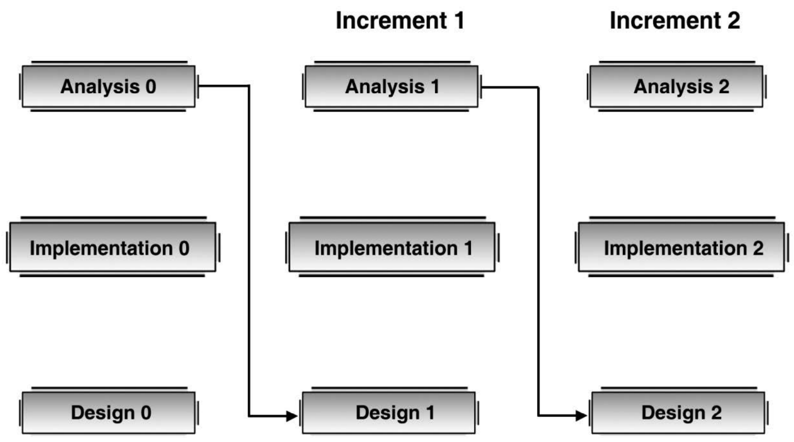


Fig. 1.11 *Incremental Model*

The basic idea of this model is to start the process with requirements and iteratively enhance the requirements until the final software is implemented. In addition, as in prototyping, the increment provides feedback from the user, specifying the requirements of the software. This approach is useful as it simplifies the software development process as implementation of smaller increments is easier than implementing the entire system.

These stages add additional functionality to the product and pass it on to further stages. The first increment is generally known as a core product and is used by the user for a detailed evaluation. This process results in creation of a plan for the next increment. This plan determines the modifications (features or functions) of the product in order to accomplish user requirements. The iteration process, which includes the delivery of the increments to the user, continues until the software is completely developed. The increments result in implementations, which are assessed in order to measure the progress of the product.

Various advantages associated with incremental model are:

1. The problem understanding increases through successive refinements in the incremental method.
2. It performs cost benefit analysis before enhancing software with capabilities.
3. It incrementally grows in effective solution after each multiple iterations.
4. It does not involve high complexity rate.
5. Early feedback is generated in this model, because implementation occurs rapidly for a small sub-set of the software.

Various disadvantages associated with the incremental model are:

- It requires planning at the management and technical level.
- It becomes invalid when there is time constraint in the project schedule or when the users cannot accept the phased deliverables.

Check Your Progress

4. Elaborate on the term flowchart.
5. What is DFD?
6. Define the term software process model.
7. What is waterfall model?

1.6 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Institute of Electrical and Electronic Engineers (IEEE) defines software as a 'collection of computer programs, procedures, rules and associated documentation and data.'
2. Software crisis is a term that has been used since the early days of software engineering to describe the impact of the rapid increases in computer power and its complexity. Software crisis occurs due to problems associated with poor quality software. This includes problems arising from malfunctioning of software systems, inefficient development of software and most important, dissatisfaction among users of the software.

3. Software engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
4. A flowchart is a graphical representation of an algorithm where a sequence of steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows.
5. Data-flow diagram (DFD) is defined as a diagram that depicts data sources, data sinks, data storage and processes performed on data as nodes and logical flow of data as links between the nodes.
6. IEEE defines a process model as 'a framework containing the processes, activities and tasks involved in the development, operation and maintenance of a software product, spanning the life of the system from the definition of its requirements to the termination of its use.'
7. In the waterfall model (also known as the classical life cycle model or linear sequential model) the development of software proceeds linearly and sequentially from requirement analysis to design, coding, testing, integration, implementation and maintenance.

1.7 SUMMARY

- Software is a collection of programs, procedures, rules, data and associated documentation. Software is responsible for managing, controlling and integrating the hardware components of a computer system and to accomplish any given specific task.
- Software characteristics are classified into six major components, namely functionality, reliability, usability, efficiency, maintainability and portability.
- Software can be applied in countless situations such as business, education, social sector and other fields. The only thing that is required is a defined set of procedural steps. On the basis of its applications, software is classified as system software, real-time software, business software, engineering and scientific software, artificial intelligence software, Web-based software and personal computer software.
- Depending on how closely software users or purchasers are associated with software development, software is classified as commercial off-the-shelf software, customized or bespoke software and customized COTS software.
- The major causes of a software crisis are the problems associated with poor quality of software, such as malfunctioning of software systems, inefficient development of software and dissatisfaction among people who are using the software.
- Software engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

- There are three layers of software engineering, namely the process layer method layer and tools layer.
- To engineer software adequately, three distinct phases, namely definition, development and support phase are applied.
- To help the programmer design programs having good control flow structure, a flowcharting technique was developed. In the flowcharting technique, the algorithm is represented using flowcharts.
- Structured programming became a powerful tool that allowed programmers to write moderately complex programs easily. Structured programming forces a logical structure in the program to be written in an efficient and understandable manner.
- IEEE defines a data-flow diagram as a diagram that depicts data sources, data sinks, data storage, processes performed on data as nodes and logical flow of data as links between the nodes.
- Some of the process models are the waterfall model, prototyping, spiral, incremental model, time boxing model, RAD model, V model, build and fix model and the formal methods model.
- In the prototyping model, a working representation, known as prototype, of the software is developed. It gives the actual feel of the software to the user. Thus, it gives an indication of how complex the system will be after it is developed.
- The spiral model was developed to incorporate the project as the major aspect of software development. This model provides a disciplined framework for software development that overcomes the deficiencies of other process models.
- The incremental model combines the characteristics of the waterfall model with the prototyping model. This model produces a functional product known as increment at the end of each linear sequence.

1.8 KEY TERMS

- **Software:** A collection of programs, documentation and operating procedures.
- **Usability:** The degree to which the software is user friendly.
- **Robustness:** The extent to which software can continue to operate correctly despite the introduction of valid input.
- **Integrity:** The extent to which unauthorized access or modification of the software or data can be controlled in the computer system.
- **Real-time software:** The type of software which observes, analyses and controls real-world events as they occur.
- **AI software:** The type of software which is used where the problem-solving technique is non-algorithmic in nature.

- **Process models:** They are models that comprise processes, methods, tools or steps for developing software.
- **Increment:** It is a functional product produced at the end of each linear sequence, by the incremental model. It has the combined characteristics of the waterfall and prototyping process models.

1.9 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Define software along with its characteristics.
2. State about the software myths.
3. What are the problems associated with software?
4. What are the principles of software engineering that are considered while developing a software?
5. What is spiral model?
6. Write the various advantages and disadvantages of the incremental model.
7. What are the advantages of software process model?

Long-Answer Questions

1. Explain how software can be classified on the basis of its applications and the way in which software users are associated with software development.
2. Explain the spiral model of software engineering.
3. Discuss about the various layers of software engineering.
4. What is a process model? Outline the major steps involved in a spiral model.
5. Discuss how the waterfall model is different from the prototyping model.
6. Explain why the waterfall model is known as the linear sequential model.
7. How is the spiral model similar to the waterfall and prototyping model?

1.10 FURTHER READING

Mali, Rajib. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hall of India.

Pressman, Roger S. *Software Engineering: A Practitioner’s Approach*. New York: McGraw-Hill Education.

Jalote, Pankaj. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Sommerville, Ian. *Software Engineering*. New Delhi: Addison Wesley.

Fenton, N.E. and F.L. Pfleeger. *Software Metrics*. Mumbai: Thomson Learning.

UNIT 2 SOFTWARE ENGINEERING METHODOLOGIES AND CONFIGURATION MANAGEMENT ISSUES

Structure

2.0 INTRODUCTION

In order to accomplish a set of tasks, it is important to go through a sequence of steps. This sequence of steps refers to a road map, which in software engineering helps in developing a timely, high-quality and highly efficient product or system. A road map, commonly referred to as a software process, comprises activities, constraints and resources that are used to produce an intended system. Software process helps to maintain a level of consistency and quality in products or services that are produced by different people. The software development process, also known as the software life cycle, describes the evolution of a software product from its conception through its design, implementation, delivery and finally to its maintenance.

Software configuration management is used to determine those products which are supposed to be changed and establish a relationship among them. On the other hand, the quality assurance plan is a structured process for defining the procedures and methods that are used to develop software.

In this unit you will study about the software engineering process, software metrics and software configuration management issues.

2.1 OBJECTIVES

After going through this unit you will be able to:

- Understand the software engineering process
- Define the software metrics
- Know about the software configuration management issues

2.2 SOFTWARE ENGINEERING PROCESS

Software engineering comprises of interrelated and recurring entities, which are essential for software development. A software is developed efficiently and effectively with the help of well-defined activities or processes. A process is defined as a series of steps involving activities and resources, which produce the desired output. A process can also be defined as a collection of procedures to develop a software product according to certain goals or standards. Generally, the following points are noted about software processes:

- Processes use resources subject to given constraints and produce intermediate and final products.
- Processes are composed of sub-processes that are organized in such a manner that each sub-process has its own process model.
- Each process is carried out with an entry and exit criteria that help in monitoring the beginning and completion of the activity.
- Every process includes guidelines, which explain the objectives of each activity.
- Processes are vital because they impose uniformity on the set of activities.
- A process is regarded as more than just procedures, tools and techniques, which are collectively used in a structured manner to produce a product.
- Software processes include various technical and management issues, which are required to develop a software.

The characteristics of software processes are listed in Table 2.1.

Table 2.1 *Characteristics of a Software Process*

Characteristic	**Description**
Comprehensibility	The extent to which the process is explicitly defined and the ease with which the process definition is understood.
Visibility	Whether the process activities culminate in clear results or not, so that the progress of the process is visible externally.
Supportability	The extent to which CASE tools can support the process activities.
Acceptability	The extent to which the defined process is acceptable and usable by the engineers responsible for creating the software product.
Reliability	The manner in which the process is designed so that errors in the process are avoided or trapped before they result in errors in the product.
Robustness	Whether or not the process can continue in spite of unexpected problems.
Maintainability	Whether the process can evolve to reflect the changing organizational requirements or identify process improvements.
Rapidity	The speed with which the complete software can be delivered with given specifications.

A project is defined as a specification essential for developing or maintaining a specific product. A software process or a software project can be easily developed. The activities in a software project comprise various tasks for managing resources and developing product. Figure 2.1 shows that a software project involves people (developers, project manager end users, and so on) also referred to as participants who use software processes to produce a product according to user requirements. The participants play a major role in the development of the project and select the appropriate process for the project. In addition, a project is efficient if it is developed within the time constraint. The outcome or the result of a software project is known as product. Thus, a software project uses software processes to produce a product.

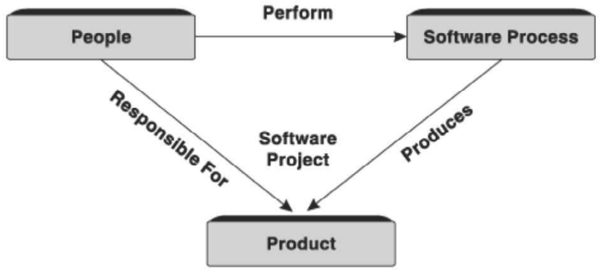


Fig. 2.1 *Software Project*

A software process can consist of many software projects, each of which can produce one or more software products. The interrelationship among these three entities (process, project and product) is shown in Figure 2.2. A software project begins with requirements and ends with the accomplishment of the requirements. Thus, a software process should be performed to develop the final software by accomplishing user requirements. It is to be noted that software processes are not specific to the software project.

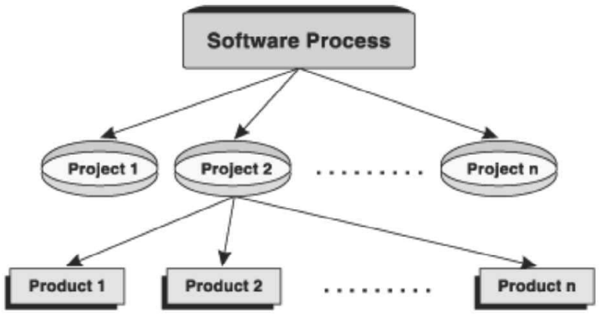


Fig. 2.2 *Processes, Projects and Products*

Processes and Their Allocation to Physical Elements

The objective of a software process is to develop a product, which accomplishes user requirements. For this, software processes requires components, which are shown in Figure 2.3. The major components of software process includes process management process and product engineering process. The process management processes (PMP) aim to improve software processes so that a cost-effective and high-quality product is developed. To achieve a high-quality product, the existing processes of the completed project are examined. The process of comprehending the existing process, analysing its properties, determining how to improve it and then effecting the improvement is carried out by PMP. A group known as Software Engineering Process Group (SEPG) performs the activities of process management.

Based on the afore-mentioned analysis, the product engineering processes are improved, thereby improving the software process. The aim of the product engineering processes is to develop the product according to user requirements. The product engineering process comprises three major components, namely:

- **Development process:** It implies the process, which is used during the development of a software. It specifies the development and quality assurance activities that are to be performed. Programmers, designers, testing personnel and others perform these processes.
- **Project management process:** It is concerned with the set of activities or tasks, which are used to successfully accomplish a set of goals. It provides the means to plan, organize and control the allocated resources to accomplish project costs, time and performance objectives. For this, various processes, techniques and tools are used to achieve the objectives of the project. Project management performs the activities of this process.
- **Configuration control process:** It manages the changes that occur as a result of modifying the requirements. In addition, it maintains integrity of the products with the changed requirements. The activities in configuration control processes are performed by a group called the Configuration Control Board (CCB).

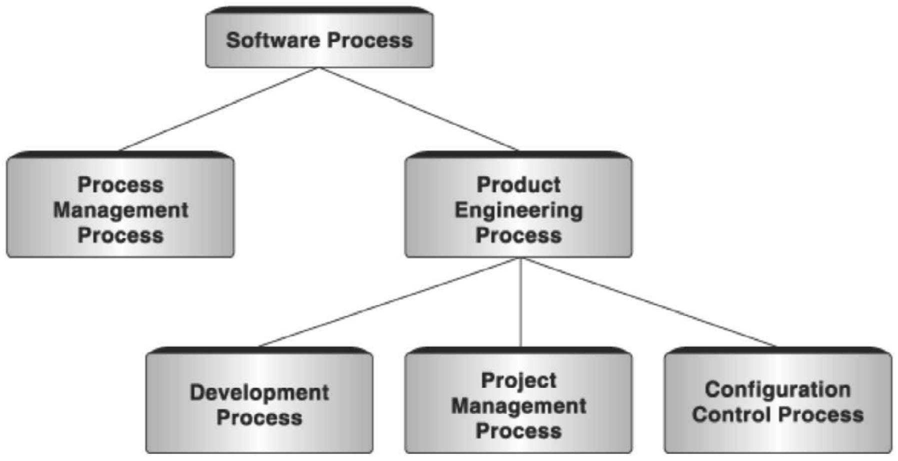

Software Process
Process Management Process
Product Engineering Process
Development Process
Project Management Process
Configuration Control Process

Fig. 2.3 *Components of Software Processes*

It must be noted that the project management process and configuration control process depend on the development process. The management process aims to control the development process, depending on the activities in the development process.

Process Framework

A process framework determines the processes that are essential for completing a complex software project. This framework identifies certain activities which are applicable to all software projects regardless of their type and complexity. The activities used for these purposes are commonly referred to as framework activities, as shown in Figure 2.4. Some of the framework activities are:

- **Communication:** It involves communication with the user so that the requirements are easily understood.
- **Planning:** It establishes a project plan for the project. In addition, it describes the schedule for the project, the technical tasks involved, expected risks and the required resources.
- **Modelling:** It encompasses creation of models, which allows the developer and the user to understand software requirements. In addition, it determines the designs to achieve those requirements.
- **Construction:** It combines generation of code with testing to uncover errors in the code.
- **Deployment:** It implies that the final product (that is, the software) is delivered to the user. The user evaluates the delivered product and provides a feedback based on the evaluation.

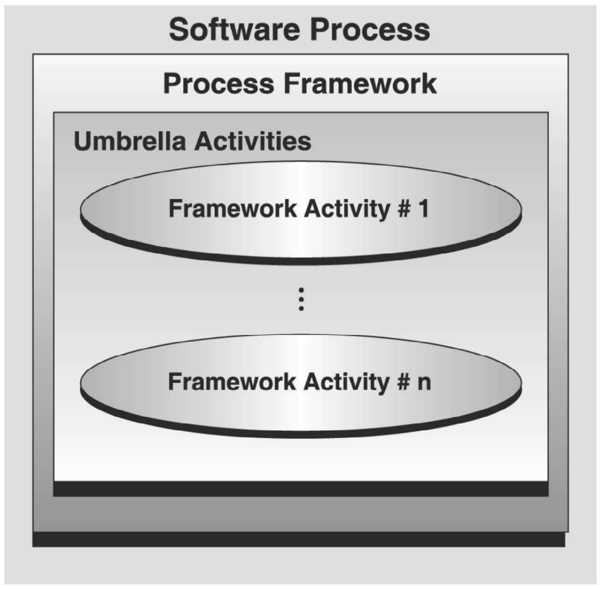


Fig. 2.4 *Process Framework*

In addition to these activities, process framework also comprises a set of activities known as umbrella activities which are used throughout the software process. These are:

- **Software project tracking and control:** It monitors the actual process so that the management can take necessary steps if the software project deviates from the plans laid down. It involves tracking procedures and reviews to check whether the software project is according to user requirements or not. A documented plan is used as a basis for tracking the software activities and revising the plans. The management monitors these activities.
- **Formal technical reviews:** They assess the code, products and documents of software engineering practices to detect errors.

- **Software quality assurance:** It assures that the software is according to the requirements. In addition, it is designed to evaluate the processes of developing and maintaining quality of the software.
- **Reusability management:** It determines the criteria for products' reuse and establishes mechanisms to achieve reusable components.
- **Software configuration management:** It manages the changes made in the software processes of the products throughout the life cycle of the software project. It controls changes made to the configuration and maintains the integrity in the software development process.
- **Risk management:** It identifies, analyses, evaluates and eliminates the possibility of unfavourable deviations from expected results, by a systematic activity and then develops strategies to manage them.

2.2.1 Software Metrics

Once measures are collected they are converted into metrics for use. IEEE defines metric as a quantitative measure of the degree to which a system, component, or process possesses a given attribute. The goal of software metrics is to identify and control essential parameters that affect software development. The other objectives of using software metrics are:

- Measuring the size of the software quantitatively
- Assessing the level of the complexity involved
- Assessing the strength of the module by measuring coupling
- Assessing the testing techniques
- Specifying when to stop testing
- Determining the date of release of the software
- Estimating the cost of resources and project schedule

Note that to achieve these objectives, software metrics are applied to different projects for a long period of time to obtain indicators. Software metrics help project managers to gain an insight into the efficacy of the software process, project and product. This is possible by collecting quality and productivity data and then analysing and comparing these data with past averages in order to know whether quality improvements have occurred or not. Also, when metrics are applied in a consistent manner, it helps in project planning and project management activity. For example, schedule-based resource allocation can be effectively enhanced with the help of metrics.

2.3 CONFIGURATION MANAGEMENT ISSUES

Changes occur during all the phases of software development. They arise when user requirements are not understood properly or when new business or market conditions cause changes in product requirement (see Figure 2.5). Various other reasons for changes are:

- New user requirements may demand modification of data or functions produced by the software.
- The growth in business may change the project priorities.
- Cost and time constraints may require the system or product definition to be redefined.

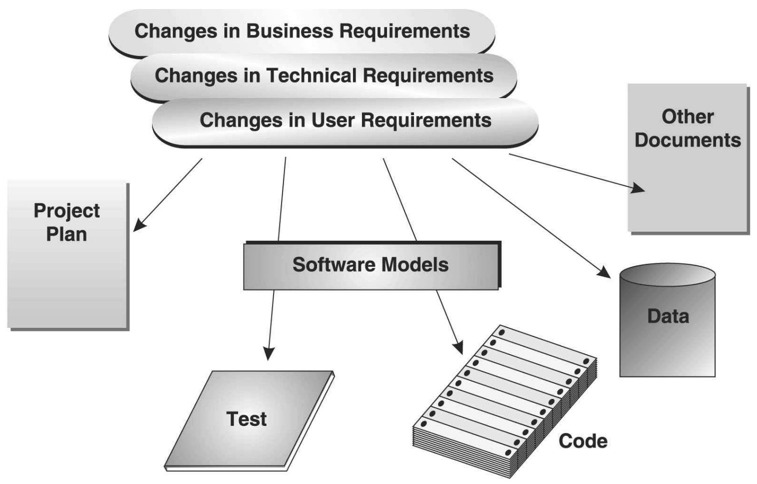

Changes in Business Requirements
Changes in Technical Requirements
Changes in User Requirements
Other Documents
Project Plan
Software Models
Data
Test
Code

Fig. 2.5 *Changes in Software Development*

To control these changes, SCM is used that determines those products that are supposed to be changed and establishes relationships among them. Note that in software configuration management, changes to work products should not affect their integrity and traceability. SCM is also responsible for auditing and reporting these changes. Other objectives of software configuration management include planning the configuration management activities, controlling changes identified in work products and identifying the selected work products.

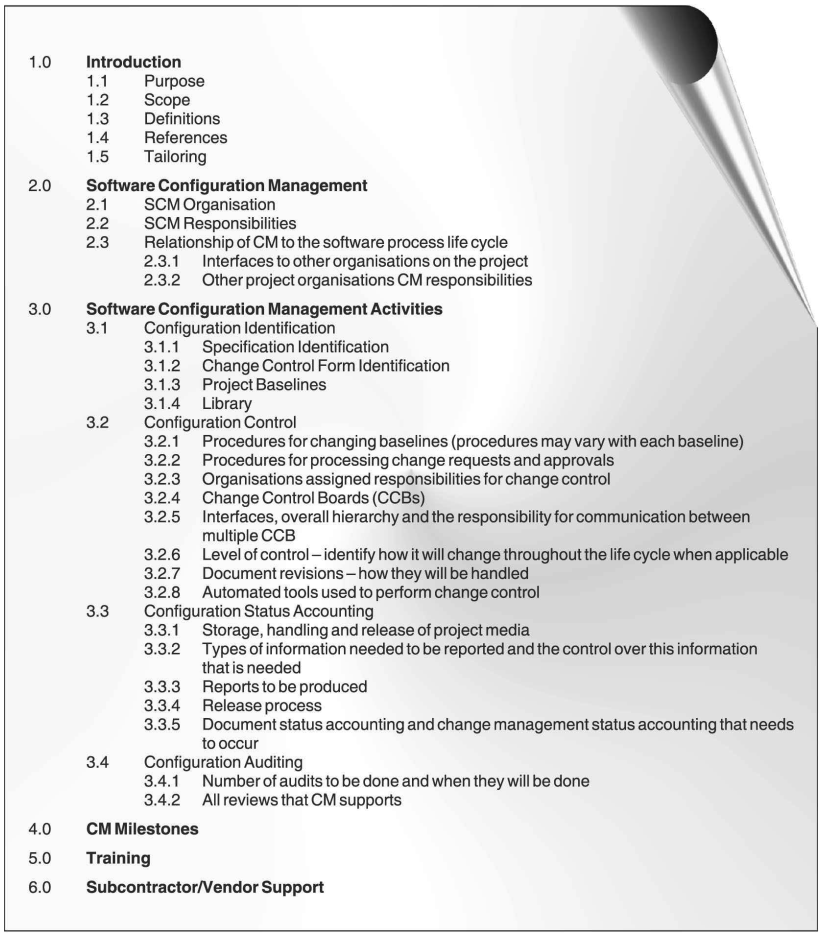
IEEE defines software configuration management as, 'The process of identifying and defining components in a system, controlling the release and change throughout the life cycle, recording and reporting the status of components and change requests and verifying the completeness and correctness of system components.'

The software configuration management provides significant benefits to all software projects regardless of their size, scope and complexity. These benefits are:

- It ensures that only approved changes are implemented in both new and existing products.
- It verifies that the changes are implemented according to the desired specifications.
- It evaluates the impact of changes and communicates it to the project manager.
- It allows overall system progress to be faster and efficient.
- It ensures that all updates are accurately shown in the documentation.

Software Configuration Management Plan

The activities to be performed in software configuration management should be planned in a proper manner. For this, a plan known as *software configuration management plan* (SCMP) is developed in the early stages of software development. This plan describes the standards and procedures used in SCM. It also determines SCM activities to be performed, schedule for those activities, assign responsibilities and resources required that include staff and the various tools. A software configuration plan should include a configuration database or repository to keep a record of configuration information. Figure 2.6 shows a software configuration management plan. This plan comprises various sections, as follows:



- 1.0 **Introduction**
  - 1.1 Purpose
  - 1.2 Scope
  - 1.3 Definitions
  - 1.4 References
  - 1.5 Tailoring
- 2.0 **Software Configuration Management**
  - 2.1 SCM Organisation
  - 2.2 SCM Responsibilities
  - 2.3 Relationship of CM to the software process life cycle
    - 2.3.1 Interfaces to other organisations on the project
    - 2.3.2 Other project organisations CM responsibilities
- 3.0 **Software Configuration Management Activities**
  - 3.1 Configuration Identification
    - 3.1.1 Specification Identification
    - 3.1.2 Change Control Form Identification
    - 3.1.3 Project Baselines
    - 3.1.4 Library
  - 3.2 Configuration Control
    - 3.2.1 Procedures for changing baselines (procedures may vary with each baseline)
    - 3.2.2 Procedures for processing change requests and approvals
    - 3.2.3 Organisations assigned responsibilities for change control
    - 3.2.4 Change Control Boards (CCBs)
    - 3.2.5 Interfaces, overall hierarchy and the responsibility for communication between multiple CCB
    - 3.2.6 Level of control – identify how it will change throughout the life cycle when applicable
    - 3.2.7 Document revisions – how they will be handled
    - 3.2.8 Automated tools used to perform change control
  - 3.3 Configuration Status Accounting
    - 3.3.1 Storage, handling and release of project media
    - 3.3.2 Types of information needed to be reported and the control over this information that is needed
    - 3.3.3 Reports to be produced
    - 3.3.4 Release process
    - 3.3.5 Document status accounting and change management status accounting that needs to occur
  - 3.4 Configuration Auditing
    - 3.4.1 Number of audits to be done and when they will be done
    - 3.4.2 All reviews that CM supports
- 4.0 **CM Milestones**
- 5.0 **Training**
- 6.0 **Subcontractor/Vendor Support**

Fig. 2.6 *Software Configuration Management Plan*

- **Introduction**: It specifies information regarding the purpose, scope, definitions and the references of SCM activities. In addition, it includes the steps and activities that describe how configuration management is to be performed for the work product. The definitions and abbreviations used for software configuration are also described in this section.
- **Software configuration management**: It specifies the responsibilities of the organization in configuration activities. It determines the relationship between configuration management and SDLC.

- **Software configuration management activities**: It specifies the configuration activities to be conducted. As shown in Figure 2.6, it comprises subsections, which are:
  - *Configuration identification* identifies the configuration items or the documents requiring changes. The specification identification describes the labelling and numbering scheme for the documents. In addition, it describes how the identification scheme addresses version and releases. Project baselines identify different baselines used in the project and keep track of whether the baseline requires the changes or not. The library section specifies the plans and procedures for recovery when loss occurs due to the changes made. For this, it determines the number of libraries and their types utilized for SCM activities.
  - *Configuration control* describes the procedures for changing baselines (as the procedures vary for each baseline) and for processing of a change request. This can be implemented with change report documentation. In addition, this section keeps a record of the organizational responsibilities for change control as the changes occur throughout the life cycle of the software project. The change control board or configuration control board (CCB) manage the decisions regarding changes. Besides this, the section also describes automated tools that are required to perform the change control.
  - *Configuration status accounting* stores all the information related to software configuration activities. In this section, status of the configuration items and handling and release process is also described. For this, information needs to be stored in the report. Various reports produced can be management reports, quality assurance reports, configuration control board reports and so on.
  - *Configuration auditing* specifies the number of audits and the time when they are to be performed. The audit provides information regarding the role of configuration management in the audit. In addition, this section determines the documents to be audited.
- **CM milestones**: It specifies all configuration management milestones (such as baselines, reviews, audits, and so on) and determines how these milestones are according to software development process. In addition, it identifies the criteria to be utilized for achieving a milestone.
- **Training**: It specifies the kind and the amount of training (such as orientation, tools) provided to the individuals according to software configuration management activities.
- **Subcontractor/Vendor support**: It specifies vendor support and interfacing, if applicable.

The responsibilities of various individuals associated with configuration management are as follows:

- **Configuration manager**: The latter checks whether policies and procedures for creating and modifying the product are carried out successfully or not. For this, he/she uses a mechanism for making a request for the changes. These *change requests* are examined and approved by the configuration manager. After the approval for change request, information is collected for determining the problematic components in the system.
- **Software engineer**: The latter uses tools to develop a consistent product. This is possible only when engineers do not interfere unnecessarily with the work of others while the code is being created and tested. However, some interaction is important when the work of engineers is dependent on each other. This interaction involves discussing and consulting one another about the tasks that are required and the tasks that are completed.
- **Project manager**: The latter ensures that the project is developed within the allotted time. For this, the project manager keeps track of the development process and identifies as well as reacts to the problems. This can be achieved by analysing reports and status and by having regular reviews of the system.
- **Users**: They play a major role in software configuration management as the product is developed for them. Users follow a specified procedure for requesting changes and indicate the problems associated with the product.

Check Your Progress

1. Define the term software engineering process.
2. What is process framework?
3. Write the goal of software metrics.
4. Give the definition of software configuration management according to IEEE.

2.4 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A software engineering process is defined as a series of steps involving activities and resources, which produce the desired output. A process is a collection of procedures to develop a software product according to certain goals or standards.
2. A process framework determines the processes that are essential for completing a complex software project. This framework identifies certain activities which are applicable to all software projects regardless of their type and complexity.
3. The goal of software metrics is to identify and control essential parameters that affect software development.

4. IEEE defines software configuration management as, 'The process of identifying and defining components in a system, controlling the release and change throughout the life cycle, recording and reporting the status of components and change requests and verifying the completeness and correctness of system components.'

2.5 SUMMARY

- A software engineering process is defined as a series of steps involving activities and resources, which produce the desired output. A process is a collection of procedures to develop a software product according to certain goals or standards.
- A project is defined as a specification essential for developing or maintaining a specific product. A software process or a software project can be easily developed. The activities in a software project comprise various tasks for managing resources and developing product.
- A software process can consist of many software projects, each of which can produce one or more software products. The interrelationship among these three entities (process, project and product).
- The process management processes (PMP) aim to improve software processes so that a cost-effective and high-quality product is developed. To achieve a high-quality product, the existing processes of the completed project are examined.
- IEEE defines metric as a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- The goal of software metrics is to identify and control essential parameters that affect software development.
- IEEE defines software configuration management as, 'The process of identifying and defining components in a system, controlling the release and change throughout the life cycle, recording and reporting the status of components and change requests and verifying the completeness and correctness of system components.'
- The activities to be performed in software configuration management should be planned in a proper manner. For this, a plan known as software configuration management plan (SCMP) is developed in the early stages of software development.
- A software configuration plan should include a configuration database or repository to keep a record of configuration information.
- Configuration control describes the procedures for changing baselines (as the procedures vary for each baseline) and for processing of a change request.
- Configuration status accounting stores all the information related to software configuration activities.
- Configuration auditing specifies the number of audits and the time when they are to be performed. The audit provides information regarding the role of configuration management in the audit.

2.6 KEY TERMS

- **Project management process:** It is concerned with the set of activities tasks, which are used to successfully accomplish a set of goals.
- **Configuration control process:** It manages the changes that occur as a result of modifying the requirements.
- **Process framework:** It determines the processes that are essential for completing a complex software project.
- S**oftware quality assurance**: It assures that the software is according to the requirements.
- **Software metric:** It defines metric as a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- **Project manager:** The latter ensures that the project is developed within the allotted time.

2.7 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write the characteristics of software process.
2. State about the activities of process framework.
3. Write the objectives of software metrics.

Long-Answer Questions

1. Briefly explain the software engineering process giving appropriate examples.
2. Discuss various components of software processes with the help of diagram.
3. Describe briefly software configuration management issues with the help of diagram.
4. Explain the software configuration management plan with the help of appropriate examples.

2.8 FURTHER READING

Mali, Rajib. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hall of India.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Education.

Jalote, Pankaj. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Sommerville, Ian. *Software Engineering*. New Delhi: Addison Wesley.

Fenton, N.E. and F.L. Pfleeger. *Software Metrics*. Mumbai: Thomson Learning.

UNIT 3 SOFTWARE REQUIREMENT ANALYSIS AND SPECIFICATION

NOTES

Structure

3.0 INTRODUCTION

In the software development process, the requirements phase is the first software engineering activity. It translates the ideas or views into a requirements document. This phase is a user-dominated phase. Defining and documenting the user’s requirements in a concise and unambiguous manner is the first major step to achieve a high-quality product. The requirement phase encompasses a set of tasks that help to specify the impact of the software on the organization, customer’s needs, and how users will interact with the developed software. The requirements are the basis of system design. If the requirements are not correct, the end product will also contain errors. Note that requirement activity like all other software engineering activities should be adapted to the needs of the process, the project, the product and the people involved in the activity. Also, the requirements should be specified at different levels of detail.

Requirement analysis (also known as problem analysis) helps to understand, interpret, classify and organize the software requirements in order to assess the feasibility, completeness and consistency of the requirements. The requirement specification can be in the form of a written document, a mathematical model, a prototype, and so on. In the validation phase, the work products created as a consequence of requirements engineering are examined for consistency, omissions and ambiguity. The basic objective is to ensure that the software requirements specification reflects the actual requirements accurately and clearly.

In this unit you will study about software requirement, requirement analysis and specification, model-based specification and validation.

3.1 OBJECTIVES

After going through this unit you will be able to:

- Understand the software requirements
- Define the requirement analysis and specification
- Discuss the different types of requirements
- Understand the algebraic specification
- Know about model-based specification
- Define the Z schemas
- Understand the requirement of validation

3.2 SOFTWARE REQUIREMENTS

Software requirement is a condition or a capability possessed by software or system component in order to solve a real world problem. The problems can be to automate a part of a system, to correct shortcomings of an existing system, to control a device, and so on. IEEE defines requirement as, '(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation of a condition or capability as in (1) or (2).'

Software requirements document (SRD) describe how a system should act, appear or perform. For this, when users request for a software, they possess an approximation of what the new system should be capable of doing. Software requirements differ from one user to another and from one business process to another.

***Note**: Information about software requirements is stored in a database which helps the software development team to understand user requirements and develop software according to those requirements.*

Guidelines for Expressing Requirements

The purpose of SRD is to provide a basis for the mutual understanding between users and designers of the initial definition of the SDLC, including the requirements, operating environment and development plan.

The SRD should include in the overview, the proposed methods and procedures, a summary of improvements, a summary of impacts, security, privacy and internal control considerations, cost considerations and alternatives. The requirements section should state the functions required of the software in quantitative and qualitative terms, and how these functions will satisfy the performance objectives. The SRD should also specify the performance requirements, such as accuracy, validation, timing and flexibility. Inputs and outputs need to be explained, as well as data characteristics. Finally, the SRD needs to describe the operating environment and provide or make reference to a development plan.

There is no standard method to express and document software requirements. Requirements can be stated efficiently by the experience of knowledgeable individuals, observing past requirements and by following guidelines. Guidelines act as an efficient method of expressing requirements, which also provide a basis for software development, system testing and user satisfaction. The guidelines that are commonly followed to document requirements are:

- Sentences and paragraphs should be short and written in active voice. Also, proper grammar, spelling and punctuation should be used.
- Conjunctions such as 'and' and 'or' should be avoided as they indicate the combination of several requirements in one requirement.
- Each requirement should be stated only once so that it does not create redundancy in the requirements specification document.

Types of Requirements

Software requirements help to understand the behaviour of a system, which is described by various tasks of the system. For example, some of the tasks of a system are to provide a response to input values, determine the state of data objects, and so on. Note that requirements are considered prior to the development of the software. The requirements, which are commonly considered, are classified into three categories (see Figure 3.1), namely *functional requirements*, *non-functional requirements* and *domain requirements*.

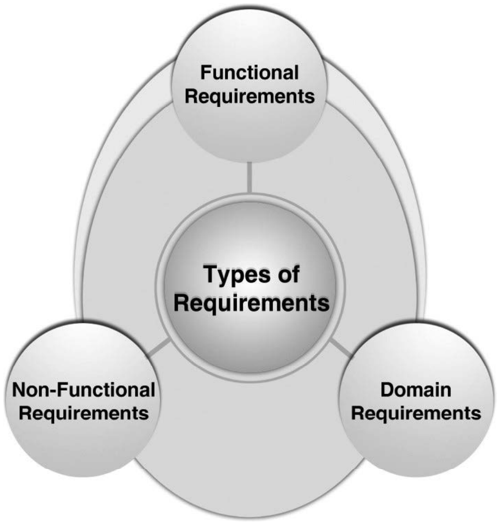


Fig. 3.1 *Types of Requirements*

3.2.1 Functional Requirements

The functional requirements (also known as *behavioural requirements*) describe the functionality or services that software should provide. For this, functional requirements describe the interaction of software with its environment and specify the inputs, outputs, external interfaces and the functions that should not be included in the software. Also, the services provided by functional requirements specify the procedure by which the software should react to particular inputs or behave in particular situations. IEEE defines functional requirements as, 'A function that a system or component must be able to perform.'

To understand functional requirements properly, let us consider the following example of an online banking system.

- The user of the bank should be able to search the desired services from the available ones.
- There should be appropriate documents for users to read. This implies that when a user wants to open an account in the bank, the forms must be available so that the user can open an account.
- After registration, the user should be provided with a unique acknowledgement number so that he can later be given an account number.

The aforementioned functional requirements describe the specific services provided by the online banking system. These requirements indicate user requirements and specify that functional requirements may be described at different levels of detail in online banking system. With the help of these functional requirements, users can easily view, search and download registration forms and other information about the bank. On the other hand, if requirements are not stated properly, software engineers may misinterpret them and user requirements would not be met.

The functional requirements should be complete and consistent. *Completeness* implies that all the user requirements are defined. *Consistency* implies that all requirements are specified clearly without any contradictory definition. Generally, it is observed that completeness and consistency cannot be achieved in large software or in a complex system due to the errors that arise while defining the functional requirements of these systems. The different needs of stakeholders also prevent the achievement of completeness and consistency. Due to these reasons, requirements may not be obvious when they are first specified and may further lead to inconsistencies in the requirements specification.

3.2.2 Non-Functional Requirements

The non-functional requirements (also known as *quality requirements*) relate to system attributes, such as reliability and response time. Non-functional requirements arise due to user requirements, budget constraints, organizational policies, and so on. These requirements are not directly related to any particular function provided by the system.

Non-functional requirements should be accomplished in software to make it perform efficiently. For example, if an aeroplane is unable to fulfil reliability requirements, it is not approved for safe operation. Similarly, if a real-time control system is ineffective in accomplishing non-functional requirements, the control functions cannot operate correctly.

NOTES

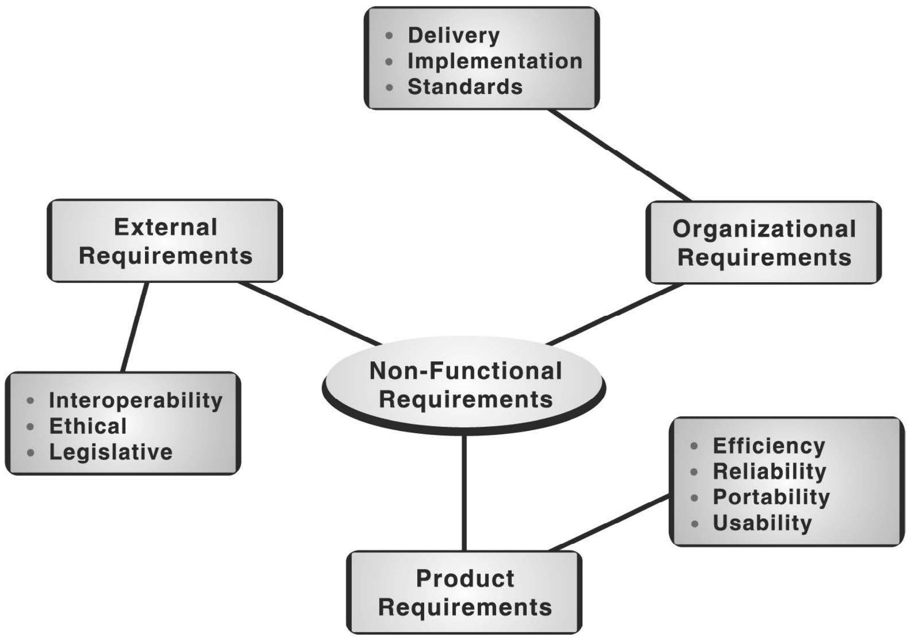


Fig. 3.2 *Types of Non-Functional Requirements*

Different types of non-functional requirements are shown in Figure 3.2. The description of these requirements is listed here.

- **Product requirements**: These requirements specify how a software product performs. Product requirements comprise the following:
  - *Efficiency requirements*: These describe the extent to which the software makes optimal use of resources, the speed with which the system executes, and the memory it consumes for its operation. For example, the system should be able to operate at least three times faster than the existing system.
  - *Reliability requirements*: These describe the acceptable failure rate of the software. For example, the software should be able to operate even if a hazard occurs.
  - *Portability requirements*: These describe the ease with which the software can be transferred from one platform to another. For example, it should be easy to port the software to a different operating system without the need to redesign the entire software.
  - *Usability requirements*: These describe the ease with which users are able to operate the software. For example, the software should be able to provide access to functionality with fewer keystrokes and mouse clicks.
- **Organizational requirements**: These requirements are derived from the policies and procedures of an organization. Organizational requirements comprise the following:
  - *Delivery requirements*: These specify when the software and its documentation are to be delivered to the user.

NOTES

  - *Implementation requirements*: These describe requirements, such as programming language and design method.
  - *Standards requirements*: These describe the process standards to be used during the software development. For example, the software should be developed using standards specified by the International Organization for Standardization (ISO) and IEEE standards.
- **External requirements**: These requirements include all the requirements that affect the software or its development process externally. External requirements comprise the following:
  - *Interoperability requirements*: These define the way in which different computer-based systems interact with each other in one or more organizations.
  - *Ethical requirements*: These specify the rules and regulations of the software so that they are acceptable to users.
  - *Legislative requirements*: These ensure that the software operates within the legal jurisdiction. For example, pirated software should not be sold.

Non-functional requirements are difficult to verify. Hence, it is essential to write non-functional requirements quantitatively so that they can be tested. For this, non-functional requirements metrics are used. These metrics are listed in Table 3.1.

Table 3.1 *Metrics for Non-Functional Requirements*

Features	Measures
Speed	▪ Processed transaction/second ▪ User/event response time ▪ Screen refresh rate
Size	▪ Amount of memory (KB) ▪ Number of RAM chips
Ease of use	▪ Training time ▪ Number of help windows
Reliability	▪ Mean time to failure (MTTF) ▪ Portability of unavailability ▪ Rate of failure occurrence
Robustness	▪ Time to restart after failure ▪ Percentage of events causing failure ▪ Probability of data corruption on failure
Portability	▪ Percentage of target-dependent statements ▪ Number of target systems

3.2.3 Domain requirements

Requirements, which are derived from the application domain of a system, instead from the needs of the users, are known as *domain requirements*. These requirements may be new functional requirements or specify a method to perform some particular computations. In addition, these requirements include any constraint that may be present in the existing functional requirements. As domain requirements reflect the fundamentals of the application domain, it is important to understand these requirements. Also, if these requirements are not fulfilled, it may be difficult to make the system work as desired.

A system can include a number of domain requirements. For example, it may comprise a design constraint that describes the user interface, which is capable of accessing all the databases used in a system. It is important for a development team to create databases and interface designs as per established standards. Similarly, the requirements of the user, such as copyright restrictions and security mechanism for the files and documents used in the system are also domain requirements.

When domain requirements are not expressed clearly, it can result in various difficulties such as:

- **Problem of understandability**: When domain requirements are specified in the language of application domain (such as mathematical expressions), it becomes difficult for software engineers to understand them.
- **Problem of implicitness**: When domain experts understand the domain requirements but do not express these requirements clearly, it may create a problem (due to incomplete information) for the development team to understand and implement the requirements in the system.

3.3 REQUIREMENTS ANALYSIS

IEEE defines requirements analysis (also called problem analysis) as, '(1) The process of studying user needs to arrive at a definition of a system, hardware or software requirements. (2) The process of studying and refining system, hardware or software requirements.' Requirements analysis helps to understand, interpret, classify and organize the software requirements in order to assess the feasibility, completeness and consistency of the requirements. The various other tasks performed using requirements analysis are as follows:

- To detect and resolve conflicts that arises due to unclear and unspecified requirements
- To determine operational characteristics of software and how they interact with the environment
- To understand the problem for which the software is to be developed
- To develop an analysis model to analyse the requirements in the software

Software engineers perform analysis modelling and create an *analysis model* to provide information of 'what' software should do instead of 'how' to fulfil the requirements in the software. This model emphasizes information, such as the functions that the software should perform, behaviour it should exhibit and constraints that are applied on the software. This model also determines the relationship of one component with another components. The clear and complete requirements specified in the analysis model help the software development team to develop the software according to those requirements. The analysis model is created to help the development team to assess the quality of the software when it is developed. The analysis model helps to define a set of requirements that can be validated when the software is developed.

Let us consider an example of constructing a study room, where a user knows the dimensions of the room, the location of doors and windows and the available wall space. Before constructing the study room, he provides information about flooring; wallpaper, and so on to a constructor. This information helps the constructor to analyse the requirements and prepare the analysis model that describes the requirements. This model also describes what needs to be done to accomplish those requirements. Similarly, the analysis model created for the software facilitates the software development team to understand what is required in the software and then develop it.

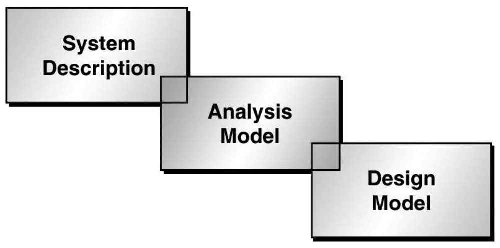


Fig. 3.3 *Analysis Model as Connector*

In Figure 3.3, the analysis model connects the system description and design model. System description provides information about the entire functionality of system, which is achieved by implementing the software, hardware and data. In addition, the analysis model specifies the software design in the form of the design model that provides information about the software's architecture, user interface and component level structure.

The guidelines followed while creating the analysis model are:

- The model should concentrate on requirements that are present within the problem with detailed information about the problem domain. However, the analysis model should not describe the procedure to accomplish requirements in the system.
- Every element of the analysis model should help in understanding the software requirements. This model should also describe the information domain, function and behaviour of the system.
- The analysis model should be useful to all stakeholders because every stakeholder uses this model in his own manner. For example, business stakeholders use this model to validate requirements whereas software designers view this model as a basis for design.
- The analysis model should be as simple as possible. For this, additional diagrams that depict no new or unnecessary information should be avoided. Also, abbreviations and acronyms should be used instead of complete notations.

The choice of representation is made according to requirements to avoid inconsistencies and ambiguities. Due to this, the analysis model comprises structured analysis, object-oriented modelling and other approaches (see Figure 3.4). Each of these describes a different manner to represent the functional and behavioural information. Structured analysis expresses this information through data-flow diagrams, whereas object-oriented modelling specifies the functional and behavioural information using objects. Other approaches include E–R modelling and several requirements specification languages and processors.

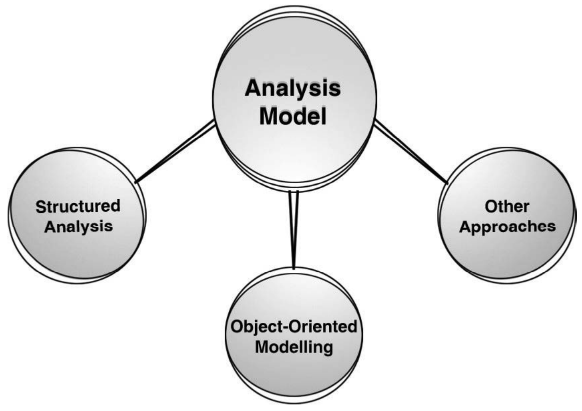


Fig. 3.4 *The Analysis Model*

3.4 REQUIREMENT SPECIFICATION

The output of the requirements phase of a software development process is the *software requirement specification* document (also known as requirements document). IEEE defines software requirement specification as, 'A document that clearly and precisely describes each of the essential requirements (functions, performance, design constraints and quality attributes) of the software and the external interfaces. Each requirement is defined in such a way that its achievement can be objectively verified by a prescribed method, for example, inspection, demonstration, analysis or test.' Note that requirement specification can be in the form of a written document, a mathematical model, a collection of graphical models, a prototype, and so on.

Role of Software Requirements Specification

This document lays a foundation for software engineering activities and is created when entire requirements are elicited and analysed. Software requirement specification (SRS) is a formal document, which acts as a representation of software that enables the users to review whether it (SRS) is according to their requirements or not. In addition, the requirements document includes user requirement for a system as well as detailed specification of the system requirement.

Essentially, what passes from requirements analysis activity to the specification activity is the knowledge acquired about the system. The need for maintaining a requirements document is that the modelling activity essentially focuses on the problem structure and not its structural behaviour. While in SRS, performance constraints, design constraints, standard compliance recoveries are clearly specified in the requirements document. This information helps in properly developed design of a system. Various other purposes served by SRS are:

- **Feedback**: It provides a feedback that ensures to the user that the organization (which develops the software) understands the issues or problems to be solved and the software behaviour necessary to address those problems.

- **Decompose problem into components**: It organizes the information and divides the problem into its component parts in an orderly manner.
- **Validation**: It uses validation strategies, applied to the requirements to acknowledge that requirements are stated properly.
- **Input to design**: It contains sufficient detail in the functional system requirements to devise a design solution.
- **Basis for agreement between the user and the organization**: It provides a complete description of the functions to be performed by the system. In addition, it helps the users to determine whether the specified requirements are accomplished or not.
- **Reduce the development effort**: It enables developers to consider user requirements before the designing of the system commences. As a result, 'rework' and inconsistencies in the later stages can be reduced.
- **Estimating costs and schedules**: It determines the requirements of the system and thus enables the developer to have a 'rough' estimate of the total cost and schedule of the project.

Various individuals in the organization use the requirement document. As shown in Figure 3.5, system customers need SRS to specify and verify whether the requirements meet the desired needs or not. In addition, SRS enables the managers to plan for the system development processes. Systems engineers need a software requirements document to understand what system is to be developed. These engineers also require this document to develop validation tests for the required system. This document is also needed by system maintenance engineers to use the requirement and the relationship between its parts.

The requirements document has diverse users. Therefore, along with communicating the requirements to the users; it also has to define the requirements in precise detail for developers and testers. In addition, it should also include information about possible changes in the system, which can help system designers to avoid restricted decisions on design. SRS also helps maintenance engineers to adapt the system to new requirements.

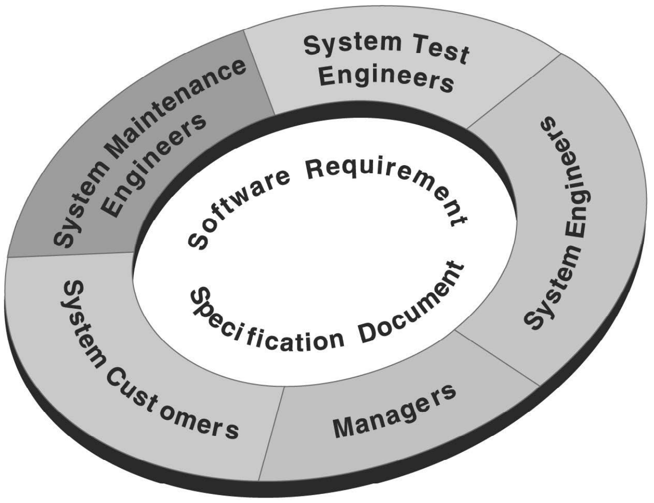

System Test Engineers
System Engineers
Managers
System Customers
System Maintenance Engineers
Software Requirement Specification Document

Fig. 3.5 *SRS Users*

Characteristics of Software Requirements Specification

Software requirement specification should be accurate, complete and efficient, and of high quality so that it does not affect the entire project plan. An SRS is said to be of high quality when the developer and user easily understand the prepared document. Other characteristics of SRS are discussed as follows:

- **Correct**: SRS is correct when all user requirements are stated in the requirements document. The stated requirements should be according to the desired system. This implies that each requirement is examined to ensure that it (SRS) represents user requirements. Note that there is no specified tool or procedure to assure the correctness of SRS. Correctness ensures that all specified requirements are performed correctly.
- **Unambiguous**: SRS is unambiguous when every stated requirement has only one interpretation. This implies that each requirement is uniquely interpreted. In case there is a term used with multiple meanings, the requirements document should specify the meanings in the SRS so that it is clear and easy to understand.
- **Complete**: SRS is complete when the requirements clearly define what the software is required to do. This includes all the requirements related to performance, design and functionality.
- **Ranked for importance/stability**: All requirements are not equally important, hence, each requirement is identified to make differences among other requirements. For this, it is essential to clearly identify each requirement. Stability implies the probability of changes in the requirement in future.
- **Modifiable**: The requirements of the user can change, hence, requirements document should be created in such a manner that those changes can be modified easily, consistently maintaining the structure and style of the SRS.
- **Traceable**: SRS is traceable when the source of each requirement is clear and facilitates the reference of each requirement in future. For this, forward tracing and backward tracing are used. Forward tracing implies that each requirement should be traceable to design and code elements. Backward tracing implies defining each requirement explicitly referencing its source.
- **Verifiable**: SRS is verifiable when the specified requirements can be verified with a cost-effective process to check whether the final software meets those requirements or not. The requirements are verified with the help of reviews. Note that unambiguity is essential for verifiability.
- **Consistent**: SRS is consistent when the subsets of individual requirements defined do not conflict with each other. For example, there can be a case when different requirements can use different terms to refer to the same object. There can be logical or temporal conflicts between the specified requirements and some requirements whose logical or temporal characteristics are not satisfied. For instance, a requirement states that an

event ‘a’ is to occur before another event ‘b’. But then another set of requirements states (directly or indirectly by transitivity) that event ‘b’ should occur before event ‘a’.

Structure of SRS

The requirements document is devised in a manner that is easier to write, review and maintain. It is organized into independent sections and each section is organized into modules or units. Note that the level of detail to be included in the SRS depends on the type of the system to be developed and the process model chosen for its development. For example, if a system is to be developed by an external contractor, then critical system specifications need to be precise and detailed. Similarly, when flexibility is required in the requirements and where an in-house development takes place, requirements documents can be less detailed.

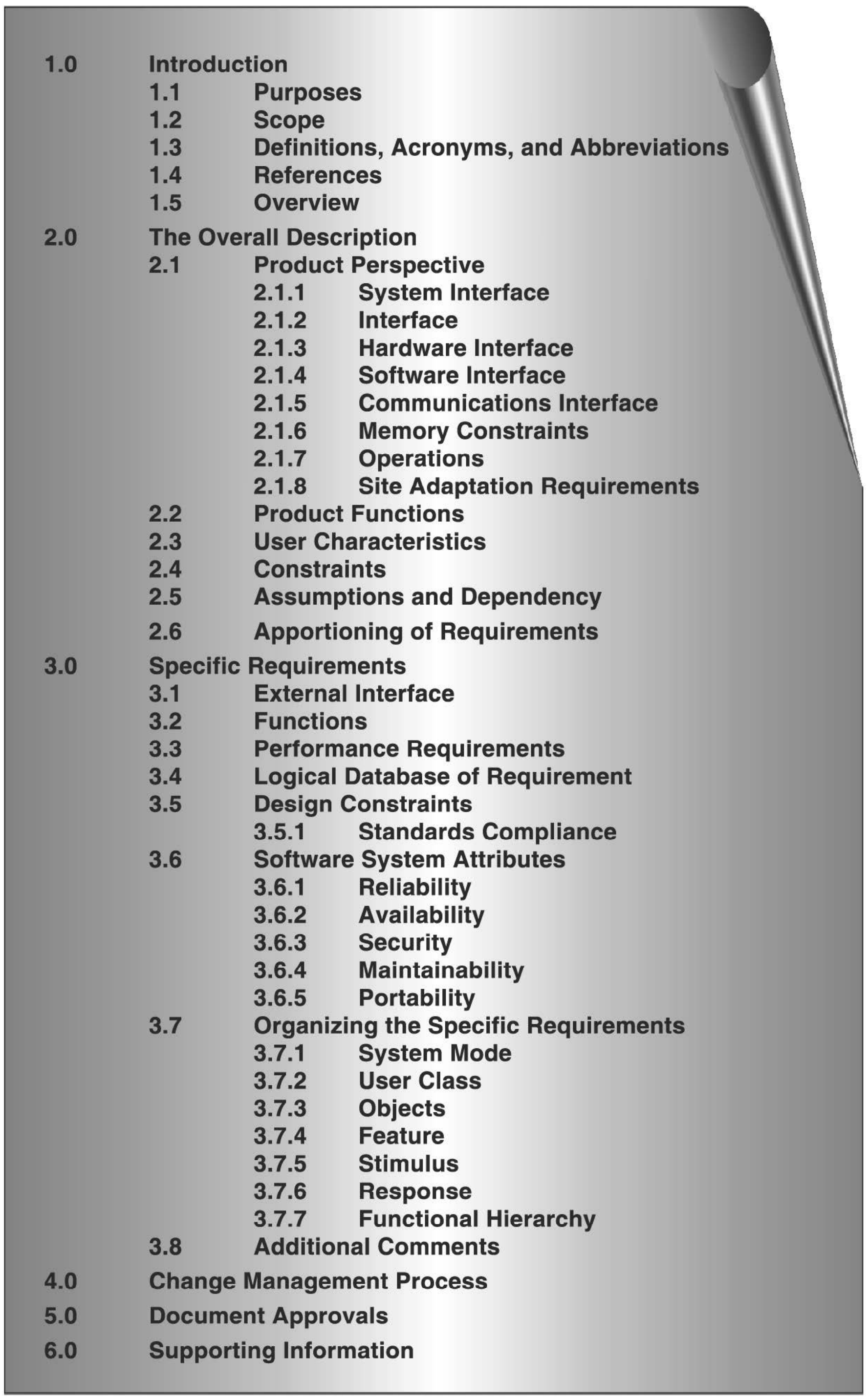
Since the requirements document serves as a foundation for subsequent software development phases, it is important to develop the document in the prescribed manner. For this, certain guidelines are followed while preparing the SRS. These guidelines are:

- **Functionality:** It should be separate from implementation.
- **Analysis model:** It should be developed according to the desired behaviour of a system. This should include data and functional response of a system to various inputs given to it.
- **A cognitive model** (express a system as perceived by the users)**:** It should be developed instead of developing a design or implementing model.
- **The content and structure of the specification:** It should be flexible enough to accommodate changes.
- **Specification:** It should be robust that is, it should be tolerant towards incompleteness and complexity.

The information to be included in SRS depends on a number of factors, for example, the type of software being developed and the approach used in its development. If software is developed using iterative development process, the requirements document will be less detailed as compared to the software being developed for critical systems. This is because specifications need to be very detailed and accurate in these systems. A number of standards have been suggested to develop a requirements document. However, the most widely used standard is by IEEE, which acts as a general framework. This general framework can be customized and adapted to meet the needs of a particular organization.

Each SRS fits into a certain pattern, thus it is essential to standardize the structure of the requirements document to make it easier to understand. For this IEEE standard is used for the SRS to organize requirements for different projects, which provides different ways of structuring SRS. Note that in all requirements documents, the first two sections are the same.

NOTES



1.0 Introduction
- 1.1 Purposes
- 1.2 Scope
- 1.3 Definitions, Acronyms, and Abbreviations
- 1.4 References
- 1.5 Overview

2.0 The Overall Description
- 2.1 Product Perspective
  - 2.1.1 System Interface
  - 2.1.2 Interface
  - 2.1.3 Hardware Interface
  - 2.1.4 Software Interface
  - 2.1.5 Communications Interface
  - 2.1.6 Memory Constraints
  - 2.1.7 Operations
  - 2.1.8 Site Adaptation Requirements
- 2.2 Product Functions
- 2.3 User Characteristics
- 2.4 Constraints
- 2.5 Assumptions and Dependency
- 2.6 Apportioning of Requirements

3.0 Specific Requirements
- 3.1 External Interface
- 3.2 Functions
- 3.3 Performance Requirements
- 3.4 Logical Database of Requirement
- 3.5 Design Constraints
  - 3.5.1 Standards Compliance
- 3.6 Software System Attributes
  - 3.6.1 Reliability
  - 3.6.2 Availability
  - 3.6.3 Security
  - 3.6.4 Maintainability
  - 3.6.5 Portability
- 3.7 Organizing the Specific Requirements
  - 3.7.1 System Mode
  - 3.7.2 User Class
  - 3.7.3 Objects
  - 3.7.4 Feature
  - 3.7.5 Stimulus
  - 3.7.6 Response
  - 3.7.7 Functional Hierarchy
- 3.8 Additional Comments

4.0 Change Management Process

5.0 Document Approvals

6.0 Supporting Information

Fig. 3.6 *Software Requirements Specification Document*

A software requirement specification document is shown in Figure 3.6. This document has many sections, namely:

- **Introduction**: This provides an overview of the entire information described in SRS. This involves purpose and the scope of SRS, which states the functions to be performed by the system. In addition, it describes definitions, abbreviations and the acronyms used. The references used in SRS provide a list of documents that is referenced in the document.

- **Overall description**: This determines factors that affect the requirements of the system. It provides a brief description of the requirements to be defined in the next section called 'specific requirement'. It comprises various subsections such as:
  - *Product perspective*: This determines whether the product is an independent product or an integral part of the larger product. It determines the interface with hardware, software, system and communication. It also defines memory constraints and operations utilized by the user.
  - *Product functions*: These provide a summary of the functions to be performed by the software. The functions are organized in a list so that the users easily understand them.
  - **User characteristics**: These determine general characteristics of the users.
  - **Constraints**: These provide the general description of the constraints, such as regulatory policies, audit functions, reliability requirements, and so on.
  - **Assumption and dependency**: This provides a list of assumptions and factors that affect the requirements as stated in this document.
  - **Apportioning of requirements**: This determines the requirements that can be delayed until release of future versions of the system.
- **Specific requirements**: These determine all requirements in detail so that the designers can design the system in accordance with them. The requirements include description of every input and output of the system and functions performed in response to the input provided. It comprises various subsections such as:
  - *External interface*: It determines the interface of software with other system that can include interface with operating system, and so on. External interface also specifies the interaction of the software with users, hardware or other software. The characteristics of each user interface of the software product are specified in SRS. For the hardware interface, SRS specifies the logical characteristics of each interface among the software and hardware components. If the software is to be executed on the existing hardware, then characteristics such as memory restrictions are also specified.
  - *Functions*: These determine the functional capabilities of the system. For each functional requirement, the accepting and processing of inputs in order to generate outputs are specified. This includes validity checks on inputs, exact sequence of operations, relationship of inputs to output, and so on.
  - *Performance requirements*: These determine the performance constraints of the software system. Performance requirement is of two types: static requirements and dynamic requirements. *Static requirements* (also known as capacity requirements) do not impose constraint on the

execution characteristics of the system. These include requirements like number of terminals and users to be supported. *Dynamic requirements* determine the constraints on the execution of the behaviour of the system, which includes response time (the time between the start and ending of an operation under specified conditions) and throughput (total amount of work done in a given time).

- *Logical database of requirement*: It determines logical requirements to be stored in the database. This includes type of information used, frequency of usage, data entities and relationship among them, and so on.
- *Design constraint*: It determines all design constraints that are imposed by standards, hardware limitations, and so on. *Standard compliance* determines requirements for the system, which are in compliance with the specified standards. These standards can include accounting procedures and report format. *Hardware limitations* implies when the software can operate on existing hardware or some pre-determined hardware. This can impose restrictions while developing a software design. Hardware limitations include hardware configuration of the machine and operating system to be used.
- *Software system attributes*: These provide attributes such as, reliability, availability, maintainability and portability. It is essential to describe all these attributes to verify that they are achieved in the final system.
- *Organizing specific requirements*: It determines the requirements so that they can be properly organized for optimal understanding. The requirements can be organized on the basis of mode of operation, user classes, objects, feature, response and functional hierarchy.

- **Change management process**: This determines the change management process in order to identify, evaluate and update SRS to reflect changes in the project scope and requirements.
- **Document approvals**: These provide information about the approvers of the SRS document with the details, such as approver's name, signature, date, and so on.
- **Supporting information**: This provides information, such as table of contents, index, and so on. This is necessary especially when SRS is prepared for large and complex projects.

3.4.1 Algebraic Specification

Algebraic specification was originally introduced by J.V. Guttag to define abstract data types (ADT). In an abstract data types, only the permissible operations on the data types are highlighted, whereas the representation details are hidden from the outside world. Specification of an ADT using this approach involves the following:

- Defining the syntax of the operations
- Specifying the interactions among operations in terms of axioms

An algebraic specification is generally presented in following sections:

- **Type section:** This section specifies the data types (known as sorts) being used. A sort is the name of a set of objects with common characteristics.
- **Exception section:** This section specifies the names of the exceptional conditions that might occur during the execution of the operation. These exception conditions are used in the later sections of the algebraic specification.
- **Signature (or syntax) section:** This section defines the syntax of the interface to ADT. The signature part consists of the names of the operations, the number and the data types (sorts) of the parameters and the return types of the operations.
- **Axioms (or equation) section:** This section defines the semantics of the operations by defining a set of axioms that characterizes the behavior of the ADT. These axioms relate the operations used to construct entities (constructor operations) of the defined part with the operations used to inspect its values (inspector operations).

The steps involved in the creation of algebraic specification are as follows:

1. The informal interface specification is first organized into a set of ADTs or object classes. The operations associated with each ADT or class should also be defined informally.
2. A name for each abstract type specification is established.
3. A set of required operations for each specification are identified, and they are classified into the following categories:
   - **Basic constructor operations** are those operations that create or modify the entities of the type (sort) defined in the specification, e.g., create, add, update, etc.
   - **Extra constructor operations** are those operations that are not basic construction operations. Remove, for example, is an extra constructor because all the items in the ADT can be retrieved, even without having the remove operator.
   - **Basic inspection operations** are those operations that evaluate the attributes of the sort defined in the specification without modifying them. For example, eval, get, etc.
   - **Extra inspection operations** are those operations that are not basic inspection operations.
4. An informal specification of each operation describing how the operations affect the defined sort is written.
5. The signature part of each operation and the parameters to each operation is defined.
6. Finally, the semantics of the operations is defined in the form of axioms.

An axiom is created for the composition of each basic constructor over each basic inspector and extra constructor. Also, an axiom is created for each of the extra inspectors in terms of any of the basic inspectors. Therefore, if we have c1 basic constructors, c2 extra constructors, i1 basic inspectors, i2 extra inspectors, then the total number of minimum axioms required will be

```
c1 – (c2 + i1) + i2
```

Note: If the data type (sort) being specified in the specification appears on the right-hand side of the expression, then the function is a constructor; otherwise, it is an inspection operation.

To understand the algebraic specification method, consider a stack of integers. The following operations can be performed on a stack:

- `create:` To create a new empty stack
- `push:` To push an element onto the stack
- `pop:` To pop the top element from the stack
- `top:` To return the top element of the stack without popping it
- `isempty:` To check whether the stack is empty

The algebraic specification for the ADT stack is given in Figure 3.7. In this specification, we have two basic constructors, one extra constructor, two basic inspectors and zero extra inspectors. Therefore, total number of axioms in this case will be

```
2*(1+2)+0 = 2*3 = 6
```

The axioms can be stated in English as follows:

1. A new created stack is empty.
2. Once an element is pushed onto the stack, the stack is not empty.
3. If an attempt is made to pop an element from a new stack, it results in an underflow condition.
4. If an attempt is made to get the top element from a new stack, it gives an undefined value.
5. Pushing an item onto a stack and immediately popping it off leaves the stack unchanged.
6. Pushing an item onto a stack and immediately retrieving the top element returns the item just pushed onto the stack.

Note: The algebraic specification should possess three important properties, namely completeness, finite termination and unique termination property.

```
    Sorts:
       defines stack[integer]
    uses Boolean, item
       (item is an integer value to be stored in the
stack)
    Exception:
       underflow, undef_value
    Syntax:
    1. create()® stack      (basic constructor)
    2. push(stack, item) ® stack  (basic constructor)
    3. pop(stack) ® stack + {underflow}
(extra constructor)
```

```
    4. top(stack) ® item + {undef_value}(basic inspector)
    5. isempty(stack) ® Boolean
(basic inspector)
Axioms(Equations):
    1. isempty(create()) = true
    2. isempty(push(stk,item)) = false
    3. pop(create()) = underflow
    4. top(create()) = undef_value
    5. pop(push(stk,item) = stk
    6. top(push(stk,item)) = item
```

Fig. 3.7 *Algebraic Specification for ADT Stack*

Consider the following instance of the type stack:

```
push(pop(push(push(create(), 35),76)),98)
```

After the execution of this sequence of operations, the stack will have two elements, 98 and 35. This state of stack can also be obtained by the following expression:

```
push(push(create(), 35),98)
```

3.5 MODEL-BASED SPECIFICATION

Requirement specification is one of the important tasks to be done in software development, since the outcome of the requirement specification phase, that is, SRS becomes the basis for further phases. It requires the use of some specification language, which should be easy to learn and use. In addition, it should also have many of the features required by the SRS, such as correctness, modifiability, stability, etc. The specification language may be informal or formal.

In an *informal specification*, a natural language, say English, is widely used for specifying the user requirements. The major advantage of using natural language is that it is understandable to both the client and supplier. However, using natural language for specifying requirements has some limitations. First, natural languages are by nature imprecise and ambiguous. Second, they are quite verbose; it means specifying requirement more precisely results in voluminous documents. To reduce some of these limitations, natural language is often used in structured fashion. One such example of informal specification is *structured English*.

On the other hand, the formal methods allow a software engineer to create more complete, consistent and unambiguous specification. Set theory and logic notation are the basis for creating a clear statement of requirements. *Formal specification* is expressed in a language whose syntax and semantics are formally defined. This language comprises a *syntax* that defines specific notation used for specification representation; *semantic,* which uses objects to describe the system; and a *set of relations,* which uses rules to indicate the objects to satisfy the specification. This mathematical specification can then be analysed to improve its consistency and completeness. Moreover, since the specification is based on mathematics, it is likely to be less ambiguous than those produced by informal methods. Some of the commonly used formal techniques are *finite state machines* and *decision tables*.

Structured English

In structured English, requirements are broken into sections, paragraphs and subparagraphs. Although there is no single standard, structured English makes use of action verbs and noun phrases and contains no adjectives or adverbs.

In the early days, software was developed in-house as the size of the software to be developed was very small. Thus, conveying the requirements verbally in natural language was enough for the development. Later, as the size of the software became very large; industries whose sole aim is to develop and provide software for other organizations come into the picture, verbal requirement was not enough. Instead, clearly stated requirements in written form were required.

Structured English is suitable to represent all the three processes typical to structured programming, i.e., sequence, conditional statements and repetition. *Sequence* requires no special structure but can be represented with one sequential statement after another. *Conditional statements* can be represented using a structure with sequence followed by conditional statements. For example,

```
If (marks of a student is less than 0 and greater than
MAX) then
{reject the form}
Else
{accept the form}
End if
```

Alternatively, it can also be represented as:

```
Read marks of student
Select Case
Case 1: Marks is less than 0 and greater than MAX
       {reject the form}
Case 2: Marks is between 0 and MAX
       {accept the form}
End Select
```

Repetition construct can be represented using a number of different structures. For example, we can represent the repetition construct using Do statement/loop as follows:

```
Do
Read marks of student
While (marks is less then 0 and greater than MAX)
```

Finite state machine

A *finite state machine* (or *finite state automata*) is a machine that has a finite set of states, out of which there is a start or initial state, and zero, one or more final states. The machine transits from one state to another state with the occurrence of an event or actions. Finite state automata (FSA) can be specified as a finite directed graph or as a transition table. When specified as a directed graph, each state of

FSA becomes a node in the graph and there exists some directed arcs between nodes of the graph. Each directed arc is labelled with the input that causes the machine to transit from one state to another state.

For example, consider the FSA specified as a graph shown in Figure 3.8. Here, FSA has five states namely, S1, S2, S3, and S4. Initially, the FSA is in the state S1, and two labelled arcs from S1 indicates that it either transits to state S2 or S3 depending on the input whether it is 0 or 1. Suppose, the input to state S1 is 1, and then it goes to state S2. Again, there is two arcs from state S2—one specifies that it remains in state S2 if input is 0, other specifies that it goes to state S4 (final state) if input is 1. Now, for all 1s it remains in the state S4, and transits to state S2 for input 0. Thus, it is clear that the FSA is in the final state S4 if the starting and ending input is 1. Similarly, FSA is in the final state S5 if the starting and ending input is 0.

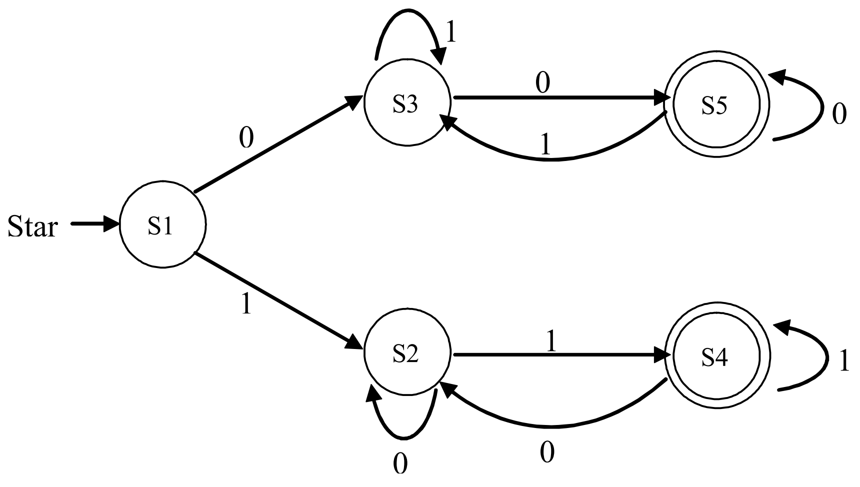


Fig. 3.8 *FSA Specified as a Graph*

***Note**: The node representing the start state by the word start and an arrow pointing to that node, and the final states by double circle have been indicated.*

When FSA is represented as a transition table, the table specifies the new state for each state for each input. For example, the FSA explained earlier can be represented as a transition table given in Table 3.2.

Table 3.2 *Transition Table*

States	Input 0	Input 1
Start S1	S3	S2
S2	S2	S4
S3	S5	S3
Final S4	S2	S4
Final S5	S5	S3

Decision table

Decision table may be defined as a table that contains all the possible conditions for a specific problem and the corresponding results using condition rules that connect conditions with results. These tables are used to specify the complicated decision logic. A decision table is composed of rows and columns, which contain four separate quadrants, as shown in Figure 3.9.

Conditions	Condition Alternatives
Actions	Action Entries

Fig. 3.9 *Four Quadrants of a Decision Table*

- **Conditions** (upper left quadrant): These contain a list of all the possible conditions pertaining to a problem.
- **Condition alternatives** (upper right quadrant): These contain the condition rules (set of possible values for each condition) of alternatives.
- **Actions** (lower left quadrant): These contain actions, which can be a procedure or operation to be performed.
- **Action entries** (lower right quadrant): These contains the action rules, which specify the actions to be performed on the basis of the set of condition alternatives corresponding to that action entry.

Note that each column in the quadrant can be interpreted as a *processing rule*. The basic four-quadrant structure is common for all the decision tables; however, they may differ in the way of representation of the condition alternatives and action entries. For example, in some decision tables, condition alternative are represented using true/false values while in other tables, Yes/No, numbers (0 or 1), fuzzy logic or probabilistic representations may be used. Similarly, in some decision tables, action entries are represented by placing check marks, cross marks or numbering the actions to be performed.

Developing a decision table

In order to build decision tables, first the maximum size of the table is determined. Any impossible situations, inconsistencies or redundancies are then eliminated from the table. To develop a decision table, follow these steps:

1. Determine all actions that can be associated with a specific module or procedure.
2. Determine the number of conditions that may affect the decision.
3. Associate the specified conditions with specific actions. In the simplest form of decision table, there are two alternatives (Y or N) for each condition.
4. Define rules by indicating actions that occur for the specified conditions.

Let us consider an example to understand the concept of decision table by designing a program for a phone card company that sends monthly bills to its customers. If the customers pay their bills within two weeks, they are offered discount as per the scheme given as follows:

- If the amount > 350, then the discount is 5%.
- If the amount >= 200 and amount <= 350, then the discount is 4%.
- If the amount < 200, there is no discount.

The decision table for this example is shown in Figure 3.10.

NOTES

Conditions	Rules			
	1	**2**	**3**	**4**
Paid within 2 weeks	Y	Y	Y	N
Order over Rs 350	Y	N	N	-
Rs 200 < = order < = Rs 350	N	Y	N	-
Order < 200	N	N	Y	-
Actions				
5% discount	X			
4% discount		X		
No discount			X	X

Fig. 3.10 *Decision Table*

3.5.1 Z Schemas

Z specifications are structured as a set of schemas a boxlike structure that introduces variables and specifies the relationship between these variables. A schema is essentially the formal specification Analog of the programming language subroutine or procedure. In the same way that procedures and subroutines are used to structure a system, schemas are used to structure a formal specification.

3.5.2 Specification using Sequences

A sequence diagram or System Sequence Diagram (SSD) shows object interactions arranged in time sequence in the field of software engineering. It depicts the objects involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of scenario. Sequence diagrams are typically associated with use case realizations in the logical view of the system under development. Sequence diagrams are sometimes called event diagrams or event scenarios. For a particular scenario of a use case, the diagrams show the events that external actors generate, their order, and possible inter-system events. All systems are treated as a black box, the diagram places emphasis on events that cross the system boundary from actors to systems. A system sequence diagram should be done for the main success scenario of the use case, and frequent or complex alternative scenarios.

Check Your Progress

1. How does IEEE define software requirement?
2. Define the term requirement analysis.
3. Give the definition of software requirement specification according to IEEE.

3.6 VALIDATION

The development of software begins, once the requirements document is 'ready'. One of the objectives of this document is to check whether the delivered software system is acceptable or not. For this, it is necessary to ensure that the requirements specification contains no errors and that it specifies the user's requirements correctly. Also, errors present in the SRS will adversely affect the cost if they are detected later in the development process or when the software is delivered to the user. Hence, it is desirable to detect errors in the requirements before the design and development of the software begins. To check all the issues related to requirements, requirements validation is performed.

In the validation phase, the work products created as a consequence of requirements engineering are examined for consistency, omissions and ambiguity. The basic objective is to ensure that the SRS reflects the actual requirements accurately and clearly. Other objectives of the requirements document are:

- To certify that the SRS contains an acceptable description of the system to be implemented
- To ensure that the actual requirements of the system are reflected in the SRS
- To check the requirements document for completeness, accuracy, consistency, requirement conflict, conformance to standards and technical errors

Requirements validation is similar to requirements analysis as both processes review the *gathered requirements*. Requirements validation studies the 'final draft' of the requirements document while requirements analysis studies the 'raw requirements' from the system stakeholders (users). Requirements validation and requirements analysis can be summarized as follows:

- *Requirements validation*: Have we got the requirements right?
- *Requirements analysis*: Have we got the right requirements?

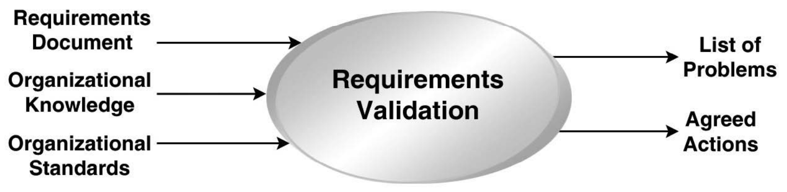


Fig. 3.11 *Requirements Validation*

In Figure 3.11, various inputs, such as requirements document, organizational knowledge and organizational standards are shown. The *requirements document* should be formulated and organized according to the standards of the organization. The ***organizational knowledge*** is used to estimate the realism of the requirements of the system. The *organizational standards* are the specified standards followed by the organization according to which the system is to be developed.

The output of requirement validation is a list of problems and their agreed actions. The *lists of problems* indicate the problems encountered in the requirements document of the requirement validation process. The *agreed action* is a list that displays the actions to be performed to resolve the problems depicted in the problem list.

Requirements Review

Requirements validation determines whether the requirements are substantial to design the system or not. Various problems are encountered during requirements validation, such as:

- Unclear stated requirements
- Conflicting requirements are not detected during requirements analysis
- Errors in the requirements elicitation and analysis
- Lack of conformance to quality standards

To avoid the problems stated here, a *requirement review* is conducted which consists of a review team that performs a systematic analysis of the requirements. The review team consists of software engineers, users and other stakeholders who examine the specification to ensure that the problems associated with consistency; omissions and errors are detected and corrected. In addition, the review team checks whether the work products produced during the requirements phase conform to standards specified for the process, project and the product or not.

At the review meeting, each participant goes over the requirements before the meeting starts and marks the items, which are dubious or need clarification. Checklists are often used for identifying such items. Checklists ensure that the reviewers do not overlook any source of errors, whether major or minor. A 'good' checklist consists of the following:

- Is the initial state of the system defined?
- Is there a conflict between one requirement and the other?
- Are all requirements specified at the appropriate level of abstraction?
- Is the requirement necessary or does it represent an add-on feature that may not be essentially implemented?
- Is the requirement bounded and has a clear defined meaning?
- Is each requirement feasible in the technical environment where the product or system is to be used?
- Is testing possible once requirement is implemented?
- Are requirements associated with performance, behaviour and operational characteristics clearly stated?
- Are requirement patterns used to simplify the requirements model?
- Are the requirements consistent with the overall objective specified for the system/product?
- Have all hardware resources been defined?
- Is the provision for possible future modifications specified?
- Are functions included as desired by the user (and stakeholder)?
- Can the requirements be implemented in the available budget and technology?
- Are the resources of requirements or any system model (created) stated clearly?

The checklists ensure that the requirements reflect users' needs and that requirements provide the groundwork for design. Using the checklist, the participants specify the list of potential errors they have uncovered. Last, the requirement analyst either agrees to the presence of errors or states that no errors exist.

Other Requirements Validation Techniques

A number of other requirement validation techniques are used either individually or in conjunction with other techniques to check the entire system or parts of the system. The selection of the validation technique depends on the appropriateness and the size of the system to be developed. Some of these techniques are:

- **Test case generation**: The requirements specified in the SRS document should be testable. The test in the validation process can reveal problems in the requirement. In some cases test becomes difficult to design which implies that the requirement is difficult to implement and requires improvement.
- **Automated consistency analysis**: If the requirements are expressed in the form of structured or formal notation, then CASE tools can be used to check the consistency of the system. A requirements database is created using a CASE tool that checks the entire requirements in the database using rules of method or notation. The report of all inconsistencies is identified and managed.
- **Prototyping**: Prototyping is normally used for validating and eliciting new requirements of the system. This helps to interpret assumptions and provide an appropriate feedback about the requirements to the user. For example, if users have approved a prototype, which consists of graphical user interface, then the user interface can be considered validated.

3.6.1 Prototyping Process

Prototyping is an approach used for problem analysis. It is different from other modelling approaches, such as structured analysis and object-oriented modelling. In this approach, partial system is designed and used to understand the problem and requirements. Note that the partial system is not delivered to the user but used only for understanding the problem.

Generally, it is observed that users are unable to understand the software requirements specification document and thus find it difficult to visualize how the software will work when it is developed. In such a case, prototyping of a system is required as it facilitates users to determine the requirements when they see the working of the system, regardless of the fact that it is only a partial system. This approach is suitable when the system is entirely new and users have no idea about their requirements as well as the working of the software. Prototyping considers practical experience to be the best help to understand and determine user requirements, which may not be clarified with the help of requirements written on paper. Once the users operate the partial system, they are able to express the features and functions they require in the system along with the ones they do not require.

NOTES

Check Your Progress

4. Give some commonly used formal specification techniques.
5. State about the Z specification.
6. List the various problems that are encountered during requirements validation.
7. What is prototyping?

3.7 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A requirement is defined as (i) a condition or capability needed by a user to solve a problem or achieve an objective. (ii) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents. (iii) A documented representation of a condition or capability as in (i) or (ii).
2. IEEE defines requirements analysis (also called problem analysis) as, the process of studying user needs to arrive at a definition of a system, hardware or software requirements the process of studying and refining system, hardware or software requirements.
3. IEEE defines software requirement specification as, 'A document that clearly and precisely describes each of the essential requirements (functions, performance, design constraints and quality attributes) of the software and the external interfaces. Each requirement is defined in such a way that its achievement can be objectively verified by a prescribed method, for example, inspection, demonstration, analysis or test.'
4. Finite state machines and decision table are commonly used formal specification techniques.
5. Z specifications are structured as a set of schemas a boxlike structure that introduces variables and specifies the relationship between these variables. A schema is essentially the formal specification Analog of the programming language subroutine or procedure.
6. The various problems that are encountered during requirements validation are listed as follows:
   - Unclear stated requirements
   - Conflicting requirements are not detected during requirements analysis
   - Errors in the requirements elicitation and analysis
   - Lack of conformance to quality standards
7. Prototyping is an approach used for problem analysis. It is different from other modelling approaches, such as structured analysis and object-oriented modelling. In this approach, partial system is designed and used to understand the problem and requirements. Note that the partial system is not delivered to the user but used only for understanding the problem.

3.8 SUMMARY

NOTES

- A requirement is defined as (i) a condition or capability needed by a user to solve a problem or achieve an objective. (ii) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents. (iii) A documented representation of a condition or capability as in (i) or (ii).
- The functional requirements, also known as behavioural requirements, describe the functionality or services that software should provide. For this, functional requirements describe the interaction of software with its environment and specify the inputs, outputs, external interfaces and sometimes the functions that should not be included in the software.
- The non-functional requirements, also known as quality requirements, relate to system attributes, such as reliability and response time. Different types of non-functional requirements include product requirements, organizational requirements and external requirements.
- Domain requirements are derived from the application domain of a system, instead from the needs of the users. These requirements may be new functional requirements or specify a method to perform some particular computations.
- IEEE defines requirements analysis (also called problem analysis) as, the process of studying user needs to arrive at a definition of a system, hardware or software requirements the process of studying and refining system, hardware or software requirements.
- Software requirement specification (SRS) is a formal document, which acts as a representation of software that enables the users to review whether it (SRS) is according to their requirements or not. In addition, the requirements document includes user requirement for a system as well as detailed specification of the system requirement. The structure of SRS differs according to the project and the requirements.
- Algebraic specification was originally introduced by Guttag to define abstract data types (ADT). Specification of an ADT using this approach involves defining the syntax of the operations and specifying the interactions among operations in terms of axioms.
- The algebraic specification should possess three important properties, namely completeness, finite termination and unique termination property.
- The specification language used for SRS may be informal or formal.
- Informal specification is a natural language, such as English, and it is widely used for specifying the user requirements.
- In structured English, requirements are broken into sections, paragraphs and subparagraphs. Although there is no single standard, structured English makes use of action verbs and noun phrases and contains no adjectives or adverbs.

NOTES

- The formal methods allow a software engineer to create more complete, consistent and unambiguous specification Formal methods make heavy use of natural language (like English) and a number of graphical notations.
- Z specifications are structured as a set of schemas a boxlike structure that introduces variables and specifies the relationship between these variables. A schema is essentially the formal specification Analog of the programming language subroutine or procedure.
- Prototyping is an approach used for problem analysis. It is different from other modelling approaches, such as structured analysis and object-oriented modelling. In this approach, partial system is designed and used to understand the problem and requirements. Note that the partial system is not delivered to the user but used only for understanding the problem.

3.9 KEY TERMS

- **Software requirement**: It is a condition or a Capability possessed by software or a system component in order to solve a real-world problem
- **Requirement or problem analysis**: It is the process of studying user needs to arrive at a definition of a system, hardware or software requirements.
- **Requirement specification**: It is a document that clearly and precisely describes each of the essential requirements of the software and the external interfaces. Each requirement is defined in such a way that its achievement can be objectively verified by a prescribed method, for example, inspection, demonstration, analysis or test.
- **Informal specification**: It is a natural language, such as English, and it is widely used for specifying the user requirements.
- **Validation**: It is similar to requirements analysis as both processes review the gathered requirements. It studies the ‘final draft’ of the requirements document while requirements analysis studies the ‘raw requirements’ from the system stakeholders (users).

3.10 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Define the term domain requirement.
2. What is the importance of problem analysis?
3. State about the role of software requirement specification.
4. What is the structure of SRS?
5. State about the specification using sequences.

NOTES

Long-Answer Questions

1. Explain non-functional requirements along with its types as applied on a system. Give examples.
2. Discuss about the analysis model with the help of diagram.
3. Describe various characteristics of software requirement specification with the help of appropriate examples.
4. What is algebraic specification? Discuss its four sections.
5. Explain the model-based specification with the help of appropriate example.
6. Briefly explain the validation and its techniques giving appropriate example.

3.11 FURTHER READING

Mali, Rajib. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hall of India.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Education.

Jalote, Pankaj. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Sommerville, Ian. *Software Engineering*. New Delhi: Addison Wesley.

Fenton, N.E. and F.L. Pfleeger. *Software Metrics*. Mumbai: Thomson Learning.

UNIT 4 PRINCIPLES OF SOFTWARE PROJECT MANAGEMENT

NOTES

Structure

4.0 INTRODUCTION

Software development is a complex activity involving people, processes and procedures. Therefore, effective management of a software project is essential for its success. Software project management (responsible for project planning) specifies the activities necessary to complete a project. The activities include determining project constraints, checking project feasibility, defining roles and responsibilities of the persons involved in the project, and much more. One of the crucial aspects of project planning is the estimation of costs, which includes the work to be done, the resources and the time required to develop the project. A careful and accurate estimation of cost is important, as cost overrun may agitate customers and lead to cancellation of the project, while a cost underestimate may force a software team to invest its time without much monetary consideration.

Risk analysis is a very important step in project management. Its significance is increasing with increasing competition, fast changing tastes, high rate of change in technology, product innovations and other current trends that we see in the sphere of business. A systematically carried out risk analysis of a proposed project helps in understanding and identifying the sources and degree of risk which, in turn, could be mitigated to the extent possible. The purpose of risk analysis is not to avoid any risky project, nor can the risk be entirely managed. Risk management aims at appropriate designing of the proposed project in a manner that the inherent risk is minimized without a corresponding reduction in returns.

In this unit you will study about the principles of software project management, principles laws of project management, software project, software metrics, resource tracking and simulation quality assurance planning and risk analysis.

4.1 OBJECTIVES

After going through this unit you will be able to:

- Understand the principles of software project management
- Discuss the need of project management
- Define the software project
- Know about the software metrics
- Understand the resource tracking and simulation
- Elaborate on the quality assurance planning
- Describe the risk analysis

4.2 ROLE OF SOFTWARE PROJECT MANAGER

The main responsibility of a Software Project Manager is to plan and schedule software project development tasks. A software project manager is concerned about whether the product meets the required standards and be finally available for use after meeting the time and financial constraints. (Brooks 1975)

The task of a software manager is the same as that of the project manager of other engineering disciplines. But to some extent, the job of a software engineer is considered to be different and difficult (Sommerville, 1998) from that of the other types of engineers. The following are some reasons why a software project manager differs from other engineers:

1. **The intangible nature of a software product**: As the software product does not have any physical appearance and can neither be seen nor be touched, so, for a project manager it is difficult to monitor the progress. The project manager has to rely on the documentation developed by others to review the progress.
2. **Lack of standard process**: There are many software processes that have been developed in the last few years and it is difficult to clearly outline the relationship between the software process and the type of product. It is difficult to clearly predict with certainty which process is likely to cause problems in development.
3. **Big difference in the new system**: It is common for a new system to be different from the previous projects in many ways. Because of this 'one-off' nature of projects, the extensive experience gained in the previous project cannot be used in the new project to reduce uncertainties.

Thus, the objective of project management is to ensure that a software product meets the quality specifications of the requirement definitions to be produced on schedule and at an economically justifiable cost within the planned resources (Royce 1998; Conte 1986).

Traditional project management has tasks grouped under three categories, which are as follows:

- **Planning—**Planning comprises tasks such as setting up overall objectives for the project; decomposing a task into smaller components; deciding and planning the process, methodology and life cycle phases for the purpose of construction of the software system; recruiting skilled people required to work on the system, both internally and externally; assigning tasks and responsibilities to the team members; developing quality project plans and identifying logistics such as, hardware, development environment and tools, long lead time items, and so on.
- **Doing—**It refers to enabling the project members to commence and perform the tasks assigned to them.
- **Monitoring and adjusting—**It refers to the frequent collection of progress reports, evaluation of the original as compared with the planned report against the metrics and milestones as set. Post collection and comparison, the required adjustments and corrections are made, if any.

Changes in the Management Framework Over the Years

Management framework has led to the evolution of many innovative and breakthrough ideas over the years. Perceived needs lead to different ideas being expressed at different times in accordance with the sentiments as prevalent in the society. The process of evolution is continuous. Ideas and concepts change frequently.

1. Management framework is like the philosophy of life. It helps an organization to build a road map, a guide to deal with problems that arise along the course of the project. The core function of a management framework is to provide direction and also chart the time at which activities need to be performed.
2. Different management frameworks emphasize on different points. But they are usually compatible with each other. Most of the management approaches do not believe in doing one thing at a time and exclude or neglect other things in a project.

A comprehensive framework is one that covers all the vital areas that a project manager needs to look into. A management framework approach is holistic in all aspects and it must have all the five arts of business management framework supporting each other. It is the ideal approach to address technical management issues. It becomes even more effective when it is supplemented with other arts of business management framework, related to the domain of a particular problem.

In the subsequent units, you will study:

- The usage and incorporation of principles and techniques, comprehensive coverage of technical management related problems, inclusive of areas such as people, process, technology leverage, organization and leadership.
- Emphasis on practices based on lessons from real-life case studies and solutions therein.

- Being up-to-date on technical management knowledge that reflects current technology and business reality (e.g., the importance of scripting language, design pattern, and the reality of outsourcing).
- Greater emphasis on the approach that focuses on achieving long-term success in projects by improving the process, people/team, infrastructures and core knowledge.

4.3 PRINCIPLES OR LAWS OF PROJECT MANAGEMENT

Project management is the art of directing and coordinating the human, and material resources throughout the project by using modern management techniques. The main purpose of project management is to achieve the predetermined objectives of scope, cost, time, quality and the satisfaction of the participant.

Project management includes developing and implementing a plan for the project while considering the available resources, such as manpower, material and cost in the organization. Project management involves the following activities:

- Planning and analysing the objectives of the project
- Measuring and controlling the risk-involved in the project
- Estimating the organizational resources required in the project
- Assigning tasks to the employees related to the project
- Directing and motivating employees to improve their performance
- Organizing project activities
- Formulating the project
- Forecasting trends in the project
- Completing the project on time
- Keeping up the quality of the project

Characteristics of Project Management

Project management is concerned with achieving a specific goal in a given time by using the resources available for that specific period. Project management requires attention for goal-oriented systems, the environment, sub-systems and their relationships. This is what makes project management a 'systems approach' to the management.

The application of principles and practices from the classical, behavioural and systems viewpoints to the unique requirements of projects has led to a new set of concepts which may be called the 'Project Viewpoint'. Cleland and King have identified the following characteristics of project management:

- The project manager is the single focal point for bringing together all the necessary resources for achieving the project objectives. He/she formally heads the project organization and operates independently the normal chain of commands. The project organization reflects the cross-functional, goal-oriented nature of the project.

- Since each project requires a variety of skills and resources, many functional areas may perform the work in a combined form. The project manager is responsible for integrating the people from different functional disciplines working on the project.
- The project manager will negotiate directly with the functional managers for support. The functional managers are responsible for the activities of the individuals and for the personnel coming under the scope of their functional groups. However, the project manager has to concentrate on integrating all the project activities and overseeing the activities from the beginning to end.
- The project manager focuses on delivering a particular product or service at a certain point of time and cost to the satisfaction of the technical requirements. In contrast, the functional units must maintain an ongoing pool of resources to reach the ultimate organizational goals instead of the limited project goals. Thus, conflicts may often arise between the project and functional managers over the optimum allocation of resources to a project.
- A project in an organizational structure has two chains of command. One is the vertical, functional reporting relationship and the other is the horizontal, project reporting mechanism.
- For rewarding incentives and distributing responsibilities, the decision-making, accountability, outcomes and rewards should be shared among all the members of the project team and the supporting functional units.
- Though the project organization is temporary, the functional units from which it is formed are permanent. Thus, when a project ends, the project team is scattered and the project personnel either return to their functional units or they are reassigned to new projects. Projects may originate from different areas of the organization. Product development and related projects tend to emerge from marketing whereas technological applications originate in Research and Development (R&D).
- Project management sets into motion numerous other support functions, such as personnel evaluation, accounting and information systems.

Prerequisites for Successful Project Management

In order to reduce the cost of constructing a project, organizations should consider various factors, such as cost and time for the successful competition among projects. Following are the prerequisites for a successful project management:

- **Adequate Project Formulation**: Project formulation is the process of converting project ideas into project proposals in a structured manner. Generally, project formulation suffers from the following shortcomings:
  - o Use of informal methods for estimating the costs and benefits, such as maintaining paper records instead of using computers
  - o Deliberate overestimation of benefits and underestimation of cost of constructing a project

  - o Faulty judgements due to lack of experienced managers and employees

    It is essential for an organization to avoid these shortcomings in order to have adequate and meaningful project formulation.

- **Project Organization**: A sound organization possesses the following characteristics:
  - o Proper working environment for employees
  - o Well-defined working methods and systems
  - o Proper rewards and penalties to employees for their performances and faults
- **Implementation Planning**: After taking investment-related decisions, it is necessary for an organization to do proper implementation planning before commencing the actual implementation. Proper implementation planning includes the following steps:
  - o Developing a plan for various activities, such as land acquisition, tender evaluation, recruitment of the staff, construction of buildings and creation of an industrial plant
  - o Estimation of the resource requirements, such as manpower, materials and money in project
- **Availability of Funds on Time**: It is important to have funds on time for taking advanced actions in the project activities. Timely availability of funds facilitates the organization to negotiate the cost of the project with suppliers and contractors.
- **Effective Monitoring**: In order to have a successful management of the project, a project monitoring system must be established in the organization. This is because effective monitoring helps in analysing the emerging problems and taking corrective actions for the project activities. Following are the factors that should be kept in mind while developing an effective system of monitoring:
  - o It should emphasize all the critical aspects, such as the finance of the project management.
  - o It should be simple and not overcomplicated as it may result in a lot of documentation and wastage of resources.

Principles of Project Management

Successful project management can be achieved by proper application of principles instead of implementing different kinds of techniques. Following are the seven principles of project management:

- To identify the project type that is suitable for the business. One needs to select the projects that are good for business.
- To understand the needs and expectations of the customers.
- To prepare the reasonable plans which defines the scope, cost and approach of the project. This helps in reducing unplanned areas in the project.

- To establish a good team with a good leader. This principle conveys that there should be proper working environment and communication flow between the project managers and team members.
- To define the status of the project. This helps improving the project quality and recognizing the various problems in it.
- To make a proper assumption for the project. This principle focuses on the verification of the critical items used in the project in order to reduce the risk.
- To take proactive actions in the problems of the project. This is because the problem usually gets worse over time which in turn increases the chances of risk.

Scope of Project Management

The scope of a project is determined by using product scope and project scope. Product scope explains all the functions and features that are to be included in a product or service of the project. On the other hand, project scope deals with the deeds to be done for delivering the needed product. The tools and techniques to manage the product scope change with the nature of the project.

The project manager uses various tools and techniques, such as product and cost-benefit analysis for developing the scope of a project. Once the project has been selected, the project manager and the client jointly prepare the scope of the project and deliverables.

Importance of Project Management

Organizations have to manage their projects effectively in order to create and maintain their reputation in the market. Many organizations fail to manage their projects properly due to the following reasons:

- Project is completed late or without fulfilling the demands of the client
- Project is not giving any valuable information
- Project lacks proper planning and organization of the activities
- The techniques and standards used are not advanced

A good project management offers various techniques and guidelines to manage employees and workloads. Project management provides the following benefits in an organization:

- **Saving Cost**: Project management offers a common methodology for managing the project, i.e., if the processes and procedures are planned once, then they can be used in all the future projects again. Consequently, it helps in saving the cost and time required in completing the project.
- **Improving Working Conditions**: If the projects are successful, the client will be more involved in the projects. This helps in improving the working environment of the organization, which in turn encourages the morale and confidence of the project team.
- **Improving Financial Management**: Better estimation of the actual costs involved in the project helps in managing the budget of the organization. This results in better financial predictability and cost control.

- **Resolving Problems**: Team members in a project spend a lot of time and energy in dealing with project problems. This is because the project team members do not know how to resolve the project problems. If the project is properly managed and planned, then the process of project management helps in solving the project problems quickly.
- **Determining Risk**: The process of project management helps in identifying and managing risks in the near future.
- **Improving the Product Quality**: The process of project management helps team members understand the needs of customers. Once customer needs are recognized, team members can implement quality control and assurance techniques to fulfil customer demands.

Need for Project Management

Modern project management ideas originated in the construction and aerospace industries in the USA and Western countries. This was because the environment and activities in those industries demanded flexible and imaginative forms of management. The spread of project management ideas has come about due to necessity rather than desire. The major reason for its slow growth can be attributed to the reluctance in accepting new approaches and techniques. The major problems identified by the managers, who attempted the new system, revolve around conflicts in authority and resources. The three major problems identified are as follows:

- Project priorities and competition for talent would interrupt the stability of the organization and interfere with its long-range interests by upsetting the normal business of the functional organizations.
- Long-range planning would suffer if the company gets more involved in meeting the schedules and fulfilling the requirements of temporary projects.
- Shifting people from project to project would disrupt the training of new employees and specialists.

Let us briefly consider some of the organizational factors influencing the need for project management.

(i) The first is the size of the organization. A small organization, such as a consulting firm, an engineering office or a small contractor who has a budget, a schedule and limited requirements to control quality and production could get along without any formal project organizational system. However, the quantitative methods and procedures of the project management approach are still necessary. On the other hand, a large organization executing a prestigious, complex, multi-disciplinary and capital-intensive project would certainly require a formal project management organizational system as well as quantitative techniques.

(ii) The second factor is the style of management needed to meet the complexities of a rapidly changing business environment. Most industrial, public service and government organizations have solved the problem of complexity due to a hierarchical management structure inherited from the military. The top management has been very comfortable with the hierarchy because of its simple 'one boss' reporting system. The hierarchy also lends itself to convenient subdivision of the organization into groups or departments, each of which represents a speciality, a discipline or a function.

While these so-called line, functional or disciplinary divisions often enhance efficiency and maximize productivity, they suffer from the following flaws:

- The ability of a specialized organization to work together and coordinate effectively with external agencies, such as the clients, vendors and regulatory agencies is critical to the success of the project. Line managers often suffer from 'tunnel vision' or lack of knowledge of the overall organizational goals. In addition, competition between line divisions may result in inefficiency or failure to communicate vital information.
- The responsibility for important external coordination may become mixed-up because of overlapping or inadequately defined roles. The allocation of responsibility for a job that overlaps several functional divisions in a project complicates the process of decision-making affecting the entire project. This may increase the possibility of inadequate or tardy responses to changing conditions which could make all the differences between the success or failure of a project.
- As an organization grows in size and complexity, it becomes increasingly difficult for the top management to connect itself with the day-to-day problems of each project.

A chief executive may face a project failure for any of these reasons, and in an attempt to determine the cause for the failure, the executive may find the divisional managers blaming each other. It can be a very disturbing experience for the top management to realize their inattentiveness over the official issues. A chief executive needs a single point of information and control if complex projects are to be successfully completed.

In determining the need for project management, one should examine the project and organization carefully and ask the following questions:

- Is the job very large?
- Is the job technically very complex?
- Is the job a true system in which it has many separate parts or sub-systems that must be integrated to complete the operation as whole?
- Is the job a part of a larger system and must be closely integrated, especially if the larger system has a project-oriented organization?
- Does the top management really feel the need for a single point of information and responsibility for the total job?
- Are strong budgetary and fiscal controls required?
- Are tight schedules and budgetary constraints foreseen?
- Are quick responses to changing conditions necessary?
- Does the job cross many disciplinary and organizational boundaries?
- Will the proposed job drastically disrupt the present organizational structure?

- Are more than two divisions involved? Is more than one division going to deal directly with the client or customer?
- Are there other complex projects being conducted concurrently with this one?
- Is there scope for a conflict between the line managers concerning this project?
- Is the organization committed to a firm completion date?
- Is it likely that changing conditions may seriously affect the project before its completion?
- Are there major items to be procured from outside the company?
- Are there major portions of the system which must be sub-contracted outside the organization?
- Is it necessary to have the project reviewed or approved by government regulatory agencies? Will these review processes and approvals generate problems and controversy?

The answers to the above questions help in the various project management considerations for an organization. From the above discussions, it can be concluded that project management can be applied to any ad hoc undertaking. This includes a broad range of activities, such as writing a research paper, remodelling a house or constructing a children's park. There are two situations in which project management should be used:

- The more unfamiliar or unique the undertaking, the greater the need for project management to ensure that nothing gets overlooked.
- The more numerous, interdisciplinary and interdependent activities in the undertaking, the greater the need for a project manager to ensure that everything is coordinated, integrated and completed.

Cleland and King have suggested five general criteria to decide when to use project management techniques and the corresponding organizational structures. These criteria are briefly described below:

- **Effort**: The magnitude of the effort should be more when the job requires more resources in an organization and for this purpose the project management techniques are necessary. For example, the Atlas missile programme requires major undertakings in the defence, aerospace, space, energy and transportation sectors. However, micro-level industrial activities may also need formal project management, e.g., in the case of relocation of facilities, merger of two companies, placing of a new product on the market, etc.
- **Coordination**: Even when a job lies primarily in one functional area, the task of coordinating its work with the other functional area is necessary. For example, the task of computer installation in a company may seem to be the sole concern of the Electronic Data Processing (EDP) department, as many corporate executives thought during the last two decades in India. Only a few smart executives and organizations realized early on in the game that during the process of computerization, there will be a continuous meshing of policies, procedures and resources of all the departments affected by computer installation. Often hundreds of

NOTES

people may be involved and the required coordination and integration might be more than what a single department, such as EDP can tackle efficiently and effectively.

- **Modification**: A project always requires modifications from time to time. Minor changes in products, such as annual automobile design changes can usually be accomplished without setting up a project team. On the other hand, undertaking the modernization of an automobile plant calls for non-routine efforts, such as revising the facilities layout, modifying the assembly line, replacing equipment, retraining employees and altering policies and work procedures. For this, project management requires to bring all the functional areas together. In a changing environment with rapid changes taking place in the economic, social and technological environment, more and more industrial organizations are seeking creative, innovative and flexible forms of management. Companies that operate in the computers, communications, electronics and pharmaceutical sectors are exposed to high innovations, rapid product changes, shifting markets and consumer behaviour. Other industries, such as those in biotechnology, petrochemicals and ceramics, though less volatile, also have highly competitive and dynamic environments.
- **Changing Environment**: Another aspect of the changing environment that is particularly relevant for the Indian economy is the government's policy of liberalization and transition to the free market mode. Changing environments present opportunities that organizations must capture swiftly. Project management provides flexibility and diversity needed to deal with changing goals and new opportunities. When a joint effort is required, project management attempts to build lateral relationships between functional areas in order to accelerate work and reconcile the conflicts inherent in multi-functional and multi-disciplinary organizations. The project manager links and coordinates the efforts of the divisions within the parent organization as well as those of the outside: sub-contractors, suppliers and customers.
- **Reputation**: The reputation of the undertaking and what is at stake may determine the need for project management. An unsuccessful project will result in either a loss of future contracts, damaged reputation, loss of market share or, in the worst case, financial ruin; therefore, there is a strong case for utilizing formal project management techniques and organizational form. For example, in the launching of American multinational Pepsi soft drinks and snack food operations in India or introduction of its new 1000 c.c. car by Maruti Udyog Ltd. or setting up of its joint venture in the form of Tata-IBM by IBM Corporation formally, each of the undertakings warranted the adoption of a formal project management approach. The obvious reason, in each of the above cases, is that the likelihood of successfully completing the undertaking is increased when a single competent individual is assigned responsibility for overseeing it. The project manager, with the assistance of technical support groups, can do much to reduce the problems inherent in large, complex undertakings.

4.4 SOFTWARE PROJECT

NOTES

Before starting a software project, it is essential to determine the tasks to be performed and to properly manage allocation of tasks among individuals involved in software development. Hence, planning is important as it results in effective software development.

Project planning is an organized and integrated management process that focuses on activities required for successful completion of a project. It prevents obstacles that arise in the project, such as changes in projects or organization's objectives, non-availability of resources, and so on. Project planning also helps in better utilization of resources and optimal usage of the allotted time for a project. The other objectives of project planning are:

- It defines the roles and responsibilities of the project management team members.
- It ensures that the project management team works according to business objectives.
- It checks feasibility of schedule and user requirements.
- It determines project constraints.

Several individuals help in planning the project. These include senior management and project management team. *Senior management* is responsible for employing team members and providing resources required for the project. The *project management team*, which generally includes project managers and developers, is responsible for planning, determining and tracking the activities of the project. Table 4.1 lists the tasks performed by individuals involved in the software project.

Table 4.1 *Tasks of Individuals involved in Software Project*

Senior Management	Project Management Team
• Approves the project, employs personnel and provides resources required for the project • Reviews project plan to ensure that it accomplishes business objectives • Resolves conflicts among team members • Considers risks that may affect the project so that appropriate measures can be taken to avoid them	• Reviews the project plans and implements procedures for completing the project • Manages all project activities • Prepares budget and resource allocation plans. • Helps in resource distribution, project management, issue resolution, and so on • Understands project objectives and finds ways to accomplish the objectives • Devotes appropriate time and effort to achieve the expected results • Selects methods and tools for the project

Principles of Project Planning

Project planning should be effective so that the project begins with well-defined tasks. Effective project planning helps to minimize the additional costs incurred on the project while it is in progress. For effective project planning, some principles are followed.

- **Planning is necessary**: Planning should be done before a project begins. For effective planning, objectives and schedule should be clear and understandable.

- **Risk analysis**: Before starting the project, senior management and the project management team should consider the risks that may affect the project. For example, the user may desire changes in requirements while the project is in progress. In such a case, the estimation of time and cost should be done according to those requirements (new requirements).
- **Tracking of project plan**: Once the project plan is prepared, it should be tracked and modified accordingly.
- **Meet quality standards and produce quality deliverables**: The project plan should identify processes by which the project management team can ensure quality in software. Based on the process selected for ensuring quality, the time and cost for the project is estimated.
- **Description of flexibility to accommodate changes**: The result of project planning is recorded in the form of a project plan, which should allow new changes to be accommodated when the project is in progress.

Project planning comprises *project purpose*, *project scope*, *project planning process* and *project plan*. This information is essential for effective project planning and to assist project management team in accomplishing user requirements.

Project Purpose

A software project is carried out to accomplish a specific purpose, which is classified into two categories, namely project objectives and business objectives. The commonly followed project objectives are:

- **Meet user requirements**: To develop the project according to user requirements after understanding them
- **Meetscheduledeadlines**:Tocompletetheprojectmilestonesasdescribed intheprojectplanontimeinordertocompletetheprojectaccordingtoschedule
- **Be within budget**: To manage the overall project cost so that the project is within budget

  Produce quality deliverables: To ensure that quality is considered for accuracy and overall performance of the project

Business objectives ensure that the organizational objectives and requirements are accomplished in the project. Generally, these objectives are related to business process improvements, customer satisfaction and quality improvements. The commonly followed business objectives are to:

- **Evaluate processes**: Evaluate the business processes and make changes when and where required as the project progresses
- **Renew policies and processes**: Provide flexibility to renew the policies and processes of the organization in order to perform the tasks effectively
- **Keep the project on schedule**: Reduce the downtime (period when no work is done) factors, such as unavailability of resources during software development
- **Improve software**: Use suitable processes in order to develop software that meets organizational requirements and provide competitive advantage to the organization

Project Scope

With the help of user requirements, project management team determines the scope of the project before the project begins. This scope provides a detailed description of functions, features, constraints and interfaces of the software that are to be considered. *Functions* describe the tasks that the software is expected to perform. *Features* describe the attributes required in the software as per the user requirements. *Constraints* describe the limitations imposed on the software by hardware, memory, and so on. *Interfaces* describe the interaction of software components (like modules and functions) with each other. Project scope also considers the software performance, which in turn depends on its processing capability and response time required to produce the output.

Once the project scope is determined, it is important to properly understand it in order to develop the software according to user requirements. After this, the project cost and duration are estimated. If the project scope is not determined on time, the project may not be completed within the specified schedule. This in turn can delay the completion of the project. The project scope gives the following information:

- List of elements included and excluded in the project
- Description of processes and entities
- Determination of functions and features required in software according to user requirements

Note that the project management team and senior management should communicate with users to understand their requirements and develop software according to those requirements and expected functionalities.

Project Planning Process

The project planning process involves a set of interrelated activities followed in an orderly manner to implement user requirements in software and includes the description of a series of project planning activities and individual(s) responsible for performing these activities. In addition, the project planning process comprises the following:

- Objectives and scope of the project
- Techniques used to perform project planning
- Effort (in time) of individuals involved in project
- Project schedule and milestones
- Resources required for the project
- Risks associated with the project

Project planning process comprises several activities that are essential for carrying out a project systematically. These activities refer to the series of tasks that are performed over a period of time for developing the software. These activities include estimation of time, effort and resources required, and risks associated with the project.

NOTES

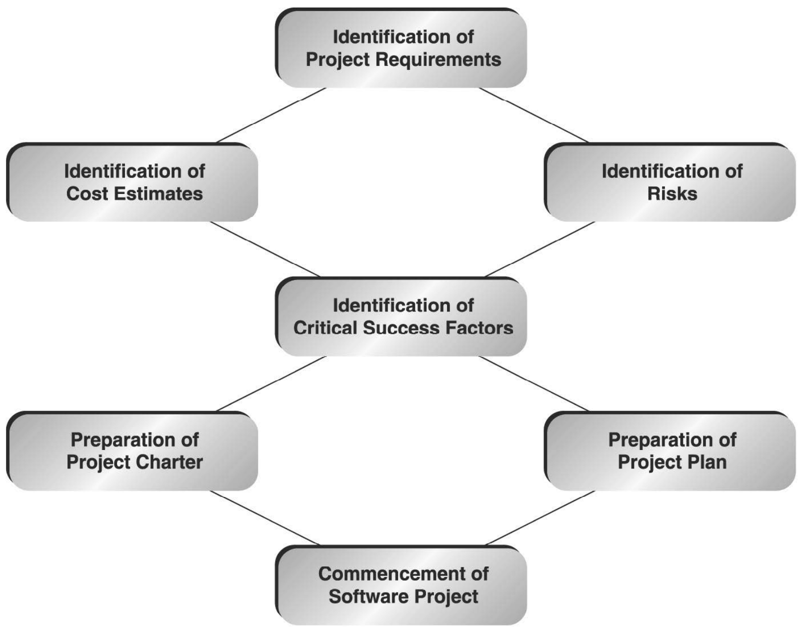

Identification of Project Requirements
Identification of Cost Estimates
Identification of Risks
Identification of Critical Success Factors
Preparation of Project Charter
Preparation of Project Plan
Commencement of Software Project

Fig. 4.1 *Project Planning Activities*

Figure 4.1 shows several activities of project planning which can be performed both in a sequence and in a parallel manner. Project planning process consists of various activities.

- **Identification of project requirements**: Before starting a project, it is essential to identify the project requirements as the identification of project requirements helps in performing the activities in a systematic manner. These requirements comprise information, such as project scope, data and functionality required in the software, and roles of the project management team members.
- **Identification of cost estimates**: Along with the estimation of effort and time, it is necessary to estimate the cost that is to be incurred on a project. The cost estimation includes the cost of hardware, network connections and the cost required for the maintenance of hardware components. In addition, cost is estimated for the individuals involved in the project.
- **Identification of risks**: Risks are unexpected events that have an adverse effect on a project. A software project involves several risks (like technical risks and business risks) that may affect the project schedule and increase the cost of the project. Identifying risks before a project begins helps in understanding their probable extent of impact on the project.
- **Identification of critical success factors**: For making a project successful, critical success factors are followed. Critical success factors refer to the conditions that ensure greater chances of success of a project. Generally, these factors include support from management, appropriate budget, appropriate schedule and skilled software engineers.
- **Preparation of project charter**: A project charter provides brief description of the project scope, quality, time, cost and resource constraints as described during the project planning. The management prepares it for approval from the sponsor of the project.

- **Preparation of project plan**: A project plan provides information about the resources that are available for the project, individuals involved in the project and the schedule according to which the project is to be carried out.
- **Commencement of the project**: Once the project planning is complete and resources are assigned to team members, the software project commences.

Once the project objectives and business objectives are determined, the project end date is fixed. The project management team prepares the project plan and schedule according to the end date of the project. After analysing the project plan, the project manager communicates the project plan and end date to the senior management. The progress of the project is reported to the management from time to time. Similarly, when the project is complete, senior management is informed about it. In case of delay in completing the project, the project plan is re-analysed and corrective action is taken to complete the project. The project is tracked regularly and when the project plan is modified, the senior management is also informed.

Project Plan

As stated earlier, a project plan stores the outcome of project planning. It provides information about the end date, milestones, activities and deliverables that are required by the project. In addition, it describes the responsibilities of project management team and the resources required for the project. It also includes the description of hardware and software (such as compilers and interfaces), and lists the methods and standards to be used in it. These methods and standards include algorithms, tools and review techniques, design language, programming language and testing techniques.

A project plan helps a project manager to understand, monitor and control the development of software project. This plan is used as a means of communication between users and the project management team. There are various advantages associated with a project plan.

- It ensures that software is developed according to user requirements, objectives and scope of the project.
- It identifies the role of each project management team member involved in the project.
- It monitors the progress of the project according to the project plan.
- It determines the available resources and the activities to be performed during software development.
- It provides an overview to management about the costs of the software project that is estimated during project planning.

Note that there are differences in the contents of two or more project plans, depending on the kind of project and user requirements. A project plan is divided into several sections, namely:

- **Introduction**: It describes the objectives of the project and provides information about the constraints that affect the software project.

NOTES

- **Project organization**: It describes the responsibilities assigned to the project management team members for completing the project.
- **Risk analysis**: It describes the risks that can possibly arise during software development as well as explains how to assess and reduce the effect of risks.
- **Resource requirements**: These specify the hardware and software that are required to carry out the software project. Cost estimation is done according to these resource requirements.
- **Work breakdown**: It describes the activities into which the project is divided. It also describes the milestones and deliverables of the project activities.
- **Project schedule**: It specifies the dependencies of activities on each other. Based on this, the time required by the project management team members to complete the project activities is estimated.

In addition to these sections, there are several plans that may be a part or linked to a project plan. These plans include *quality assurance plan*, *verification and validation plan*, *configuration management plan*, *maintenance plan* and *staffing plan*.

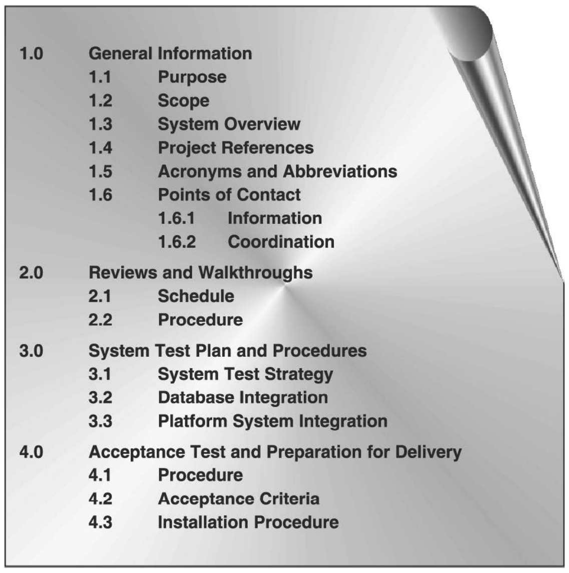
Quality assurance plan

The quality assurance plan describes the strategies and methods that are to be followed to accomplish the following objectives:

- Ensure that the project is managed, developed and implemented in an organized way.
- Ensure that project deliverables are of acceptable quality before they are delivered to the user.

Verification and validation plan

The verification and validation plan describes the approach, resources and schedule used for system validation.



1.0 General Information
- 1.1 Purpose
- 1.2 Scope
- 1.3 System Overview
- 1.4 Project References
- 1.5 Acronyms and Abbreviations
- 1.6 Points of Contact
  - 1.6.1 Information
  - 1.6.2 Coordination

2.0 Reviews and Walkthroughs
- 2.1 Schedule
- 2.2 Procedure

3.0 System Test Plan and Procedures
- 3.1 System Test Strategy
- 3.2 Database Integration
- 3.3 Platform System Integration

4.0 Acceptance Test and Preparation for Delivery
- 4.1 Procedure
- 4.2 Acceptance Criteria
- 4.3 Installation Procedure

Fig. 4.2 *Verification and Validation Plan*

Figure 4.2 shows the verification and validation plan, which comprises the following sections:

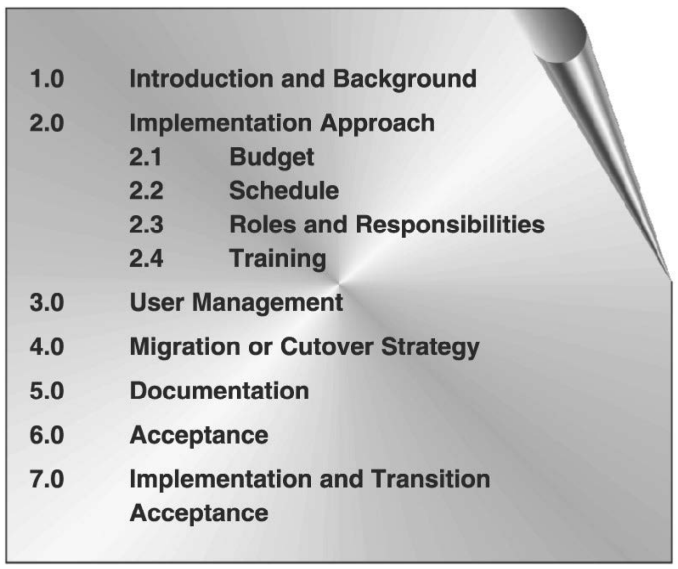
- **General information**: It provides description of the purpose, scope, and system overview and project references. *Purpose* describes the procedure to verify and validate the components of the system. *Scope* provides information about the procedures to verify and validate as they relate to the project. *System overview* provides information about the organization responsible for the project and other information, such as system name, system category, operational status of the system and system environment. *Project references* provide the list of references used for the preparation of the verification and validation plan. In addition, this section includes acronyms and abbreviations and points of contact. *Acronyms and abbreviations* provide a list of terms used in the document. *Points of contact* provide information to users when they require assistance from organization for problems, such as troubleshooting, and so on.
- **Reviews and walkthroughs**: These provide information about the schedule and procedures. *Schedule* describes the end date of milestones of the project. *Procedures* describe the tasks associated with reviews and walkthroughs. Each team member reviews the document for errors and consistency with the project requirements. For walkthroughs, the project management team checks the project for correctness according to software requirements specification (SRS).
- **System test plan and procedures**: These provide information about the system test strategy, database integration and platform system integration. *System test strategy* provides overview of the components required for integration of the database and ensures that the application runs on at least two specific platforms. *Database integration* procedure describes how database is connected to the graphical user interface (GUI). *Platform system integration* procedure is performed on different operating systems to test the platform.
- **Acceptance test and preparation for delivery**: These provide information about procedure, acceptance criteria and installation procedure. *Procedure* describes how acceptance testing is to be performed on the software to verify its usability as required. *Acceptance criteria* describes that software will be accepted only if all the components, features and functions are tested including the system integration testing. In addition, acceptance criteria checks whether the software accomplishes user expectations, such as its ability to operate on several platforms. *Installation procedure* describes the steps on how to install the software according to the operating system being used for it.

Configuration management plan

The configuration management plan defines the process, which is used for making changes to the project scope. Generally, the configuration management plan is concerned with redefining the existing objectives of the project and deliverables (software products that are delivered to the user after completion of a software development phase).

Maintenance plan

The maintenance plan specifies the resources and processes required for making the software operational after its installation. Sometimes, the project management team (or software development team) does not carry out the task of maintenance once the software is delivered to the user. In such a case, a separate team known as software maintenance team performs the task of software maintenance. Before carrying out maintenance, it is necessary for users to have information about the process required for using the software efficiently.



1.0 Introduction and Background

2.0 Implementation Approach
- 2.1 Budget
- 2.2 Schedule
- 2.3 Roles and Responsibilities
- 2.4 Training

3.0 User Management

4.0 Migration or Cutover Strategy

5.0 Documentation

6.0 Acceptance

7.0 Implementation and Transition Acceptance

Fig. 4.3 *Maintenance Plan*

Figure 4.3 shows the maintenance plan, which comprises various sections that are listed here.

- **Introduction and background**: It provides a description of the software to be maintained and the services required for it. It also specifies the scope of maintenance activities that are to be performed once the software is delivered to the user.
- **Budget**: It specifies the budget required for carrying out software maintenance and operations activities.
- **Roles and responsibilities**: It specifies the roles and responsibilities of the team members associated with the software maintenance and operation. It also describes their skills required to perform maintenance and operations activities. In addition to software maintenance team, software maintenance comprises user support, user training and support staff.
- **Performance measures and reporting**: It identifies the performance measures required for carrying out software maintenance. In addition, it describes how measures required for enhancing the performance of services (for the software) are recorded and reported.
- **Management approach**: It identifies the methodologies that are required for establishing maintenance priorities of the projects. For this purpose, the management either refers to the existing methodologies or identifies new

methodologies. The management approach also describes how users are involved in software maintenance and operations activities as well as how users and project management team communicate with each other.

- **Documentation strategies**: It provides a description of the documentation that is prepared for user reference. Generally, documentation includes reports, information about problems occurring in software, error messages and the system documentation.
- **Training**: It provides information about training activities.
- **Acceptance**: It defines a point of agreement between the project management team and software maintenance team after the completion of implementation and transition activities. Software maintenance begins after this.

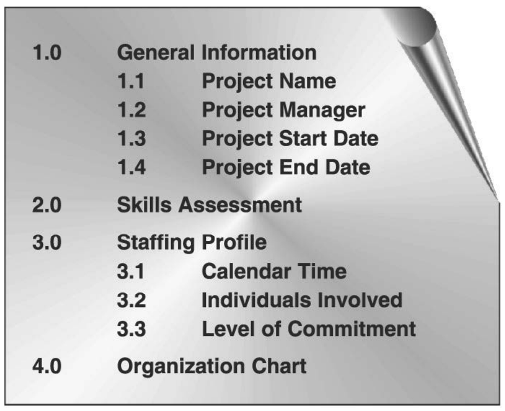
Staffing plan

The staffing plan describes the number of individuals required for a project. It includes selecting and assigning tasks to the project management team members. A staffing plan provides information about appropriate skills required to perform the task to produce the project deliverables and manage the project. In addition, it provides information of resources, such as tools, equipment and processes used by the project management team.

A staff planner, who is responsible for determining the individuals available for the project, performs the staff planning. The other responsibilities of a staff planner are:

- It recruits individuals. They can be the existing staff, staff on contract or newly employed staff. It is important for the staff planner to know the structure of the organization to determine the availability of staff.
- It determines the skills required to execute the tasks mentioned in the project schedule and task plan. In case staff with required skills is not available, staff planner informs project manager about the requirement.
- It ensures that the required staff with required skills is available at the right time. For this purpose, the staff planner plans the availability of staff after the project schedule is fixed. For example, at the initial stage of a project, staff may consist of project manager and few software engineers, whereas during software development, staff consists of software designers as well as the software developers.
- It defines the roles and responsibilities of the project management team members so that they can communicate and coordinate with each other according to the tasks assigned to them. Note that the project management team can be further broken down into a sub-team depending on the size and complexity of the project.

NOTES



1.0 General Information
- 1.1 Project Name
- 1.2 Project Manager
- 1.3 Project Start Date
- 1.4 Project End Date

2.0 Skills Assessment

3.0 Staffing Profile
- 3.1 Calendar Time
- 3.2 Individuals Involved
- 3.3 Level of Commitment

4.0 Organization Chart

Fig. 4.4 *Staffing Plan*

Figure 4.4 shows staffing plan, which comprises the following sections:

- **General information**: It provides information, such as name of the project and project manager who is responsible for the project. In addition, it specifies the start and end dates of the project.
- **Skills assessment**: It gives information that is required for assessment of skills. This information includes the knowledge, skill and ability of team members, who are required to achieve the objectives of the project. In addition, it specifies the number of team members required for the project.
- **Staffing profile**: It describes the profile of the staff required for the project. The profile includes calendar time, individuals involved and level of commitment. *Calendar time* specifies the period of time, such as month or quarter required to complete the project. *Individuals* who are involved in the project have specific designations, such as project manager and the developer. *Level of commitment* is the utilization rate of individuals, such as work performed on full-time and part-time basis.
- **Organization chart**: It describes the organization of project management team members. In addition, it includes information, such as name, designation and role of each team member.

4.5 ESTIMATION OF BUILDING A SYSTEM

Estimation models use derived formulas to predict effort as a function of LOC or FP. Various estimation models are used to estimate the cost of a software project. In these models, the cost of the software project is expressed in terms of effort required to develop the software successfully. These cost estimation models are broadly classified into two categories:

- **Algorithmic models**: Estimation in these models is performed with the help of mathematical equations that are based on historical data or theory. In order to estimate costs accurately, various inputs are provided to these algorithmic models. These inputs include software size and other parameters.

To provide accurate cost estimation, most of the algorithmic cost estimation models are calibrated to the specific software environment. The various algorithmic models used are *COCOMO, COCOMO II* and *software equation.*

- **Non-algorithmic models**: Estimation in these models depends on the prior experience and domain knowledge of project managers. Note that these models do not use mathematical equations to estimate the cost of software project. The various non-algorithmic cost estimation models are *expert judgement, estimation by analogy* and *price to win.*

Note: Here only COCOMO and software equation will be discussed.

Constructive Cost Model

In the early 1980s, Barry Boehm developed a model called **CO**nstructive **CO**st **MO**del (COCOMO) to estimate the total effort required to develop a software project. The COCOMO model is commonly used as it is based on the study of already developed software projects. While estimating total effort for a software project, costs of development, management and other support tasks are included. However, costs of secretarial and other staff are excluded. In this model, size is measured in terms of thousands of delivered lines of code (KDLOC).

In order to estimate effort accurately, COCOMO model divides projects into three categories.

- **Organic projects**: These projects are small in size (not more than 50 KDLOC) and thus easy to develop. In organic projects, small teams with prior experience work together to accomplish user requirements that are less demanding. Most people involved in these projects have thorough understanding of how the software under development contributes in achieving the organization's objectives. Examples of organic projects include simple business system, inventory management system, payroll management system and library management system.
- **Embedded projects**: These projects are complex in nature (size is more than 300 KDLOC) and the organizations have less experience in developing such type of projects. Developers also have to meet stringent user requirements. These software projects are developed under tight constraints (hardware, software and people). Examples of embedded systems include software system used in avionics and military hardware.
- **Semi-detached projects**: These projects are less complex as the user requirements are less stringent compared to embedded projects. The size of semi-detached project is not more than 300 KDLOC. Examples of semi-detached projects include operating system, compiler design and database design.

The various advantages and disadvantages associated with COCOMO model are listed in Table 4.2.

Table 4.2 *Advantages and Disadvantages of COCOMO Model*

Advantages	Disadvantages
• Easy to verify the working involved in it. • Cost drivers are useful in effort estimation as they help in understanding impact of different parameters involved in cost estimation. • Efficient and good for sensitivity analysis. • Can be easily adjusted according to the organization's needs and environment.	• Difficult to accurately estimate size, in the early phases of the project. • Vulnerable to misclassification of the project type. • Success depends on calibration of the model according to the needs of the organization. This is done using historic data that is not always available. • Excludes overhead costs, travel cost and other incidental cost.

The constructive cost model is based on the hierarchy of three models, namely *basic model*, *intermediate model* and *advance model.*

Basic model

In the basic model, *only* the size of a project is considered while calculating effort. To calculate effort, use the following equation (known as *effort equation*):

$$E = A \times (\text{size})^{B} \qquad \text{... (4.1)}$$

Where E is the effort in person-months and size is measured in terms of *KDLOC*. The values of constants 'A' and 'B' depend on the type of the software project. In this model, values of constants ('A' and 'B') for three different types of projects are listed in Table 4.3.

Table 4.3 *Values of Constants for Different Projects*

Project Type	A	B
Organic project	2.4	1.05
Semi-detached project	3.0	1.12
Embedded project	3.6	1.20

For example, if the project is an organic project having a size of 30 KDLOC, then effort is calculated using equation (5):

$$E = 2.4 \times (30)^{1.05}$$

$$E = 85 \text{ PM}$$

Intermediate model

In the intermediate model, parameters like software reliability and software complexity are also considered along with the size, while estimating effort. To estimate total effort in this model, a number of steps are followed, which are listed as follows:

1. Calculate an initial estimate of development effort by considering the size in terms of KDLOC.
2. Identify a set of 15 parameters derived from attributes of the current project. All these parameters are rated against a numeric value, called *multiplying factor*. The effort adjustment factor (EAF) is derived by multiplying all the multiplying factors with each other.
3. Adjust the estimate of development effort by multiplying the initial estimate calculated in Step 1 with EAF.

NOTES

To understand the earlier mentioned steps properly, let us consider an example. For simplicity reasons, an organic project whose size is 45 KDLOC is considered. In the intermediate model, the values of constants (A and B) are listed in Table 4.4. To estimate total effort in this model, a number of steps are followed, which are listed as follows:

1. An initial estimate is calculated with the help of effort equation (4.1). This equation shows the relationship between size and the effort required to develop a software project. This relationship is given by the following equation:

$$E_i = A \times (\text{size})^B \qquad \text{... (4.2)}$$

   Where E_i is the estimate of *initial effort* in person-months and size is measured in terms of *KDLOC*. The value of constants 'A' and 'B' depend on the type of software project (organic, embedded and semi-detached). In this model, values of constants for different types of projects are listed in Table 4.4.

Table 4.4 *Values of Constants for Different Projects*

Project Type	A	B
Organic project	3.2	1.05
Semi-detached project	3.0	1.12
Embedded project	2.8	1.20

   Using the equation (4.2) and the value of constant for organic project, initial effort can be calculated as follows:

$$E_i = 3.2 \times (45)^{1.05} = 174 \text{ PM}$$

2. Fifteen parameters are identified. These parameters are called *cost driver attributes*, which are rated as very low, low, nominal, high, very high or extremely high. For example, in Table 4.5, reliability of a project can be rated according to this rating scale. In the same table, the corresponding multiplying factors for reliability are 0.75, 0.88, 1.00, 1.15 and 1.40.

Table 4.5 *Effort Multipliers for Cost Drivers*

Cost Drivers	Description	Rating					
		Very Low	Low	Nominal	High	Very High	Extra High
RELY	Required software reliability	0.75	0.88	1.00	1.15	1.40	-
DATA	Database size	-	0.94	1.00	1.08	1.16	-
CPLX	Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
TIME	Execution time constraint	-	-	1.00	1.11	1.30	1.66
STOR	Main storage constraint	-	-	1.00	1.06	1.21	1.56
VIRT	Virtual machine volatility	-	0.87	1.00	1.15	1.30	-
TURN	Computer turnaround time	-	0.87	1.00	1.07	1.15	-
ACAP	Analyst capability	1.46	1.19	1.00	0.86	0.71	-
AEXP	Applications experience	1.29	1.13	1.00	0.91	0.82	-
PCAP	Programmer capability	1.42	1.17	1.00	0.86	0.70	-
VEXP	Virtual machine experience	1.21	1.10	1.00	0.90	-	-
LEXP	Language experience	1.14	1.07	1.00	0.95	-	-
MODP	Modern programming practices	1.24	1.10	1.00	0.91	0.82	-
TOOL	Software tools	1.24	1.10	1.00	0.91	0.83	-
SCED	Development schedule	1.23	1.08	1.00	1.04	1.10	

Next, the multiplying factors of all cost drivers considered for the project are multiplied with each other to obtain EAF. For instance, using cost drivers listed in Table 4.6, EAF is calculated as: 0.8895 (1.15 × 0.85 × 0.91 × 1.00).

Table 4.6 *Cost Drivers in a Project*

Cost Drivers	Rating	Multiplying Factor
Reliability	High	1.15
Complexity	Low	0.85
Application Experience	High	0.91
Programmer Capability	Nominal	1.00

3. Once EAF is calculated, the effort estimates for a software project is obtained by multiplying EAF with initial estimate (E_i). To calculate effort use the following equation:

```
Total effort = EAF×Ei
For this example, the total effort will be 155 PM.
```

Advance model

In advance model, effort is calculated as a function of program size and a set of cost drivers for each phase of software engineering. This model incorporates all characteristics of the intermediate model and provides procedure for adjusting the phase-wise distribution of the development schedule.

There are four phases in advance COCOMO model, namely requirements planning and product design (RPD), detailed design (DD), code and unit test (CUT) and integration and test (IT). In advance model, each cost driver is rated as very low, low, nominal, high and very high. For all these ratings, cost drivers are assigned multiplying factors. Multiplying factors for analyst capability (ACAP) cost driver for each phase of advanced model are listed in Table 4.7. Note that multiplying factors yield better estimates because the cost driver ratings differ during each phase.

Table 4.7 *Multiplying Factors for ACAP in Different Phases*

Rating	RPD	DD	CUT	IT
Very Low	1.80	1.35	1.35	1.50
Low	0.85	0.85	0.85	1.20
Nominal	1.00	1.00	1.00	1.00
High	0.75	0.90	0.90	0.85
Very High	0.55	0.75	0.75	0.70

For example, a software project (of organic project type), with a size of 45 KDLOC and rating of ACAP cost driver as nominal is considered (that is 1.00). To calculate effort for code and unit test phase in this example, only ACAP cost drivers are considered. Initial effort can be calculated by using equation (4.2):

$$E_i = 3.2 \times (45)^{1.05} = 174 \text{ PM}$$

Using the value of E_i, final estimate of effort can be calculated by using the following equation:

$$E = E_i \times 1$$

That is, $$E = 174 \times 1 = 174 \text{ PM}$$

Software Equation

In order to estimate effort in a software project, software equation assumes specific distribution of efforts over the useful life of the project. Software equation is a multivariable model, which can be derived from data obtained by studying several existing projects. To calculate effort, use the following equation:

$$E = [\text{Size} \times B^{0.333}/P]^{3} \times (1/t^{4})$$

Where,

P =	productivity parameter. The productivity parameter indicates the maturity of overall process and management practices. This parameter also indicates the level of programming language used, skills and experience of software team and complexity of software application.
E =	efforts in person-months or person-years.
t =	project duration in months or years.
B =	special skills factor. The value of B increases over a period of time as the importance and need for integration, testing, quality assurance, documentation and management increases. For small programs with sizes between 5 KDLOC and 15 KDLOC, the value of B is 0.16 and for programs with sizes greater than 70 KDLOC, the value of B is 0.39.

Note that in the foregoing given equation, there are two independent parameters, namely an estimate of size in KDLOC and project duration in calendar months or years.

4.6 SOFTWARE METRICS

Software metrics help project managers to gain an insight into the efficiency of a software process, project and product. This is possible by collecting quality and productivity data and then analysing and comparing these data with past averages in order to know whether quality improvements have occurred. Also, when metrics are applied in a consistent manner, it helps in project planning and project management activity. For example, schedule-based resource allocation can be effectively enhanced with the help of metrics.

Difference in Measures, Metric and Indicators

Metric is often used interchangeably with measure and measurement. However, it is important to note the differences between them. *Measure* can be defined as quantitative indication of amount, dimension, capacity or size of a product and process attributes. *Measurement* can be defined as the process of determining the measure. *Metrics* can be defined as the quantitative measures that allow software engineers to identify the efficiency and improve the quality of the software process, project and product.

To understand the difference, let us consider an example. A measure is established when a number of errors are (single data point) detected in a software component. Measurement is the process of collecting one or more data points. In other words, measurement is established when many components are reviewed and tested individually to collect the measure of the number of errors in all these components. Metrics are associated with individual measure in some manner. That is, metrics are related to the detection of errors found per review, or the average number of errors found per unit test.

Once measures and metrics have been developed, *indicators* are obtained. These indicators provide a detailed insight into the software process, software project or intermediate product. Indicators also enable software engineers or project managers to adjust software processes and improve software products, if required. For example, measurement dashboards or key indicators are used to monitor progress and initiate change. Arrayed together, indicators provide snapshots of a system's performance.

Measured Data

Before data is collected and used, it is necessary to know the type of data involved in software metrics. Table 4.8 lists different types of data identified in metrics along with their description and the possible operations that can be performed on them.

- **Nominal data:** Data in the program can be measured by placing it under a category. This category of program can be a database program, an application program or an operating system program. For such data, operation of arithmetic type and ranking of values in any order (increasing or decreasing) is not possible. The only operation that can be performed is the determination of whether program 'X' is the same as program 'Y.'
- **Ordinal data:** Data can be ranked according to data values. For example, experience in application domain can be rated as very low, low, medium or high. Thus, experience can easily be ranked according to its rating.
- **Interval data:** Data values can be ranked and substantial differences between them can also be shown. For example, a program with complexity level 8 is said to be four units more complex than a program with complexity level 4.
- **Ratio scale:** Data values are associated with a ratio scale, which possesses an absolute zero and allows meaningful ratios to be calculated. For example, program lines expressed in lines of code.

Table 4.8 *Type of Data Measured*

Type of Data	*Possible Operations*	*Description of Data*
Nominal	=, ≠	Categories
Ordinal	<, >	Ranking
Interval	+, -	Differences
Ratio	/	Absolute zero

It is desirable to know the measurement scale for metrics. For example, if metrics values are used to represent a model for a software process, then metrics associated with ratio scale may be preferred.

Guidelines for Software Metrics

Although many software metrics have been proposed over a period of time, ideal software metrics is the one which is easy to understand, effective and efficient. In order to develop ideal metrics, software metrics should be validated and characterized effectively. For this, it is important to develop metrics using some specific guidelines:

- **Simple and computable:** Derivation of software metrics should be easy to learn and should involve average amount of time and effort.
- **Consistent and objective:** Unambiguous results should be delivered by software metrics.
- **Consistent in the use of units and dimensions:** Mathematical computation of the metrics should involve the use of dimensions and units in a consistent manner.
- **Programming language independent:** Metrics should be developed on the basis of analysis model, design model or program's structure and should not be dependent on the programming language used.
- **High quality:** Effective software metrics should lead to a high-quality software product.
- **Easy to calibrate:** Metrics should be easy to adapt according to project requirements.
- **Low cost:** Metrics should be developed at a reasonable cost.
- **Robust:** Metrics should be relatively insensitive to small changes in the process, project or product.
- **Value:** The value of metrics should increase or decrease with the value of software characteristics it represents. For this, the value of metrics should be within a meaningful range. For example, metrics can be in a range of 0 to 5.
- **Validation:** Metrics should be validated before being used for making any decisions.

4.7 RESOURCE TRACKING AND SIMULATION/ PROJECT SCHEDULING

It is essential to perform project scheduling to effectively manage the tasks of a software project. Project scheduling provides details such as start date and end date of the project, milestones and tasks for the project. In addition, it specifies the resources (such as people, equipment and facilities) required to complete the project and the dependencies of tasks of the project on each other. An appropriate project schedule prepared according to the project plan not only aims to complete the project on time but also helps to avoid the additional cost incurred when the project is delayed.

There are various factors that delay project schedule. The commonly noticed factors are:

- **Unrealistic deadlines:** Project schedule is affected when the time allocated for completing a project is impractical and not according to the effort required for it. Generally, this situation arises when the deadline is established by inexperienced individual(s) or without the help of the project management team. Here, the project management team is constrained to work according to that deadline and if the unrealistic deadline is not achieved, the project is considered delayed.
- **Changes in user requirements:** Sometimes, project schedule is affected when user requirements are changed after the project has started. This affects the project schedule, and thus, more time is consumed in the revision of the project plan and in the implementation of new user requirements.
- **Underestimation of resources:** If the estimation of the resources for the project is not done according to its requirements, the schedule is affected. The underestimation of resources leads to delay in performing the tasks of the project.
- **Lack of consideration of risks:** Risks should be considered during project planning and scheduling; otherwise, it becomes difficult for the project management team to prevent their effect during software development.
- **Lack of proper communication among team members:** Sometimes, there is no proper communication among the project management team members to resolve the problems occurring during software development. This in turn makes it difficult for the project management team to understand and develop the software according to user requirements and schedule.
- **Difficulties faced by team members:** Software projects can also be delayed due to unforeseen difficulties faced by the team members. For example, some of the team members may require leave for personal reasons.
- **Lack of action by project management team:** Sometimes, the project management team does not recognize that the project is getting delayed. Thus, they do not take the necessary action to speed up the software development process and complete the project on time.

Generally, the task of assigning the end date is done by the project sponsor or the user. While preparing the project schedule, the project manager assists the project sponsor by providing information about the project scope, deliverables and resources. In addition, the project manager provides an estimate of the time to be consumed to complete the project tasks. Preparing an accurate project schedule is a difficult task and requires thorough knowledge about the processes and time required to perform each process. Once the project schedule is fixed, the project manager is responsible for monitoring the progress of the project. If there is a need to revise the project schedule, the project manager communicates with the project management team members.

Principles of Project Scheduling

To carry out project scheduling appropriately, some principles are followed. These principles help the project management team to prepare the project schedule. The commonly followed principles are:

- **Compartmentalization:** Divide the project into several subtasks. The purpose of compartmentalization is to make the project manageable. Thus, it becomes easier to prepare the project schedule according to these subtasks.
- **Interdependency:** Determine the interdependency of one or more activities or tasks on each other. All the activities of the project are not independent. There are various activities that are performed sequentially (one after another), whereas some of the activities are executed together. In addition, some activities cannot begin until the activity on which they are dependent is completed.
- **Time allocation:** Determine the time to be allocated to each project management team member for performing specified activities. However, before allocating time, it is important to estimate the effort required by team members to complete the assigned tasks. In addition, the project management team members should be assigned a start date and an end date according to the work to be conducted on full-time or part-time basis.
- **Effort validation:** Ensure that the effort required to perform the assigned task is valid. In other words, it should verify that the task allocated to one or more project management team members is according to the effort required for each task. This is because every project management team has a defined number of members. Hence, the project manager should allocate the tasks according to the effort and time required to complete the task.
- **Defined responsibilities:** Specify the roles and responsibilities of every project management team member. Hence, the task should be allocated according to the skills and abilities of the team members.
- **Defined outcomes:** Specify the outcomes of every task performed by the project management team members. Generally, the outcome of a task is in the form of a product and these products are combined into deliverables.
- **Defined milestones:** Specify the milestones when products are complete and reviewed for quality.

Milestones

Milestones are formal representations of the progress of a project. Generally, milestones are planned when deliverables are provided. Milestones describe the end-point at which the software process activity is completed. After completion of a milestone, its output is described in the form of a document. This document comprises information about the completion of a phase of the project. Note that these documents are used as a reference for the project management team only and are not delivered to the user.

It is difficult to keep in mind all the tasks being performed during software development. Hence, each task is recorded in documents, which describe the work being done in that phase. With the help of these documents, it becomes easy for the project manager to check the status of the project. In addition, milestones have several advantages:

- They avoid losing control of the project according to the schedule.
- They help in completing the project according to the allocated budget.
- They report the status of the project to the management.

NOTES

Figure 4.5 shows examples of milestones in the requirements and design phases of the project. 'Milestone 1,' 'Milestone 2' and 'Milestone 3' represent the completion of tasks in the requirements phase. Similarly, 'Milestone 4,' 'Milestone 5,' 'Milestone 6' and 'Milestone 7' represent the completion of tasks in the design phase. Each milestone represents the completion of a specific task. For example, 'Milestone 1' represents the completion of the feasibility study, and 'Milestone 2' represents the completion of the requirements definition. Similarly, in the design phase, 'Milestone 4' represents the completion of architectural design, and so on.

In project scheduling, there are several aspects that need to be considered. These include *techniques of project scheduling*, *task network* and *tracking the schedule*. Techniques of project scheduling focus on checking the activities that are completed according to the project schedule. In addition, these techniques describe the information about activities in a graphical form so that it is easy for the project management team to understand the time and effort required for each activity. Task network focuses on how the entire software project can be broken into several manageable tasks which are understandable by the project management team. Tracking the schedule focuses on finding ways to complete the project according to the schedule.

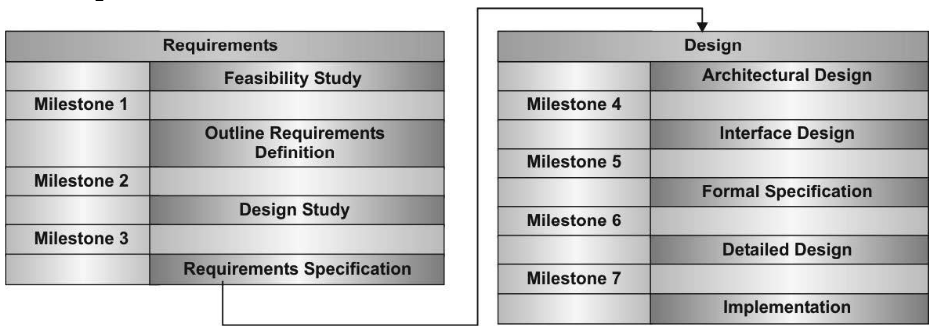


Fig. 4.5 *Milestones*

Techniques of Project Scheduling

Several techniques are used for keeping track of the project schedule. These techniques are applied after information is collected from the project planning activities. This information includes estimation of effort, selection of suitable process model, decomposition of tasks into multiple subtasks, and so on. The commonly followed techniques for project scheduling include *Gantt chart*, *PERT chart* and *critical path method*.

Gantt Chart

Gantt chart is a graphical representation of the project. It is used in project scheduling to depict the activities of the project. This chart shows the start and end dates of

NOTES

each activity in the project. In addition, it shows the number of weeks, months or quarters required to complete each activity. Due to this fact, Gantt chart is also known as *timeline chart.* It shows information about activities in the form of horizontal bars. Generally, a Gantt chart is prepared on a graph paper. In case a Gantt chart is big and complex, it is prepared using applications such as Microsoft Excel.

A Gantt chart helps the project manager by providing a graphical illustration of the project schedule. The advantages of using Gantt chart are:

- It represents the project in a graphical form.
- It reports the status of the project by showing the progress of each activity.
- It keeps a record of the activities being performed in the project.
- It depicts milestones after the completion of each activity.
- It describes the tasks assigned to the project management team members.

The schedule of two projects can vary from each other due to differences in user requirements. Hence, the Gantt chart also varies according to the project schedule. Figure 4.6 shows an example of a Gantt chart for the schedule of a project. The horizontal bars depict the total time span of the project activities. This time span is further divided into months, weeks and days. The time consumed to perform a specific activity is shown in increments. The vertical axis depicts the activities involved in the project. For example, the tasks shown on the vertical axis are preliminary investigation, writing report, conducting interviews, and so on. Note that the shaded parts of horizontal bars show the parts of activities that are completed, while the unshaded parts show the parts of activities that are not completed. Horizontal bars vary in lengths depending on the amount of time required for the completion of a specific activity. In addition, the activities can commence sequentially, in a parallel manner, or after the completion of one or more activities. Hence, the span of one or more activities can overlap. As the project progresses, the unshaded bars are shaded accordingly to indicate the completion of activities. The vertical line represents the report date.

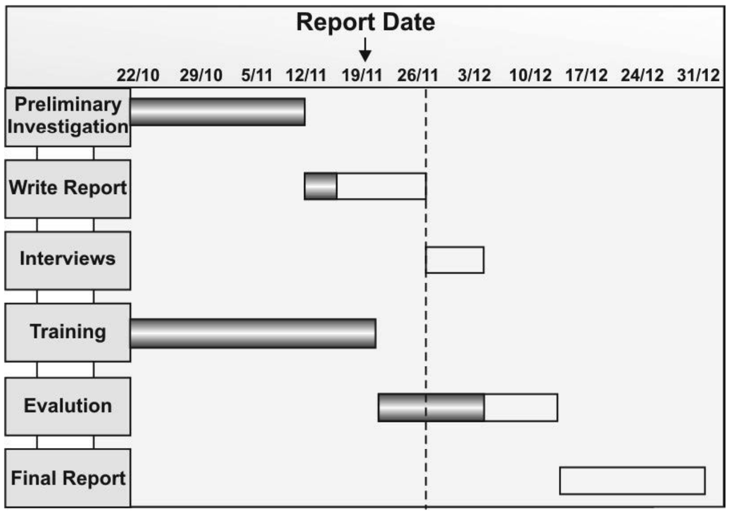


Fig. 4.6 Gantt Chart

NOTES

PERT Chart

The Program Evaluation and Review Technique (PERT) chart is used to schedule, organize and coordinate tasks within the project. The objective of PERT chart is to determine the *critical path*, which comprises critical activities that should be completed on schedule. This chart is prepared with the help of information generated in project planning activities, such as estimation of effort, selection of suitable process model for software development and decomposition of tasks into subtasks. The advantages of using a PERT chart are:

- It represents the project in a graphical form.
- It provides information about the expected completion time of the project.
- It describes the probability of completion of the project before the specified date.
- It specifies the activities that form the critical path.
- It specifies the start and end dates of the activities involved in project.
- It describes the dependencies of one or more tasks on each other.

Figure 4.7 shows an example of a PERT chart. The milestones are numbered as '1,' '2,' '3,' '4' and '5' and are represented either by circles or rectangles. When a milestone is completed, it is assigned a greater number than the previous milestones. Each milestone is linked with one or more arrows. The activities of the project are represented by 'A,' 'B,' 'C,' 'D,' 'E' and 'F.' The direction of arrows determines the sequence of activities. When activities are completed in sequence, they are known as *serial activities*. Here activities 'A,' 'C' and 'F' are performed in sequence. Similarly, activities 'B' and 'E' are serial activities. On the other hand, when two or more activities are being performed simultaneously, they are known as *concurrent activities* or *parallel activities*. Here, activities 'A' and 'B,' and activities 'C' and 'D' are performed concurrently. Each activity is allocated a specific amount of time, which is depicted by '*t*.' Here activity 'A' requires three weeks to get completed, activity 'B' requires four weeks, and so on.

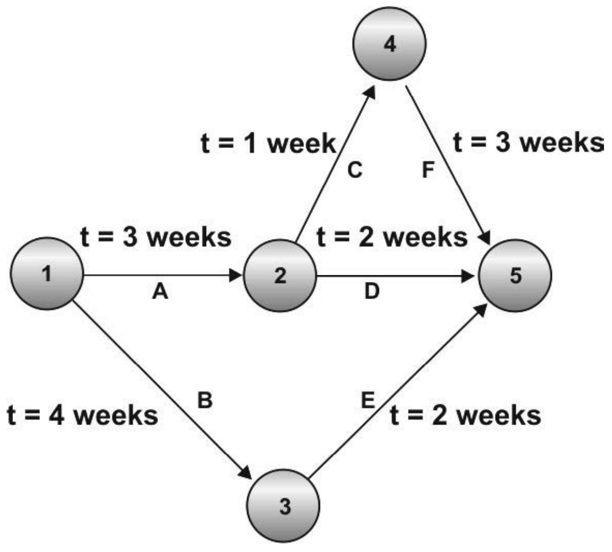


Fig. 4.7 *PERT Chart*

To create a PERT chart, follow the steps listed below:

1. **Identify activities and milestones:** In this step, the activities and milestones that are required to complete the project are identified and described. Once the tasks and milestones are specified, it is easy to understand the sequence and duration of each activity.
2. **Identify sequence of activities:** In this step, the sequence of activities is determined. The sequence describes the dependency of one activity on another. These activities can either be serial or concurrent. Based on the sequence and dependency of activities, the relationships among activities are depicted. Note that this step is sometimes combined with step 1.
3. **Prepare PERT chart:** In this step, the PERT chart is prepared.
4. **Estimate the time consumed in activities:** In this step, the amount of time consumed in carrying out each activity is estimated. The time can be estimated in months, weeks or days. Generally, the time estimates followed to determine the time consumed in the activities are:
   - **Optimistic time**: It is the shortest time in which an activity can be completed.
   - **Most likely time**: It is the completion time having the highest probability.
   - **Pessimistic time**: It is the longest time that an activity may require for completion.
5. **Determine critical path:** In this step, the critical path for the completion of activities is specified. Critical path determines the calendar time required to complete a series of activities according to the project schedule. Note that the speed-up or delay in activities that are outside the critical path do not affect the total project time.
6. **Update PERT chart:** In this step, the PERT chart is modified as changes take place in the project and on completion of each activity. This chart is also updated when there is delay in the completion of activities or when additional resources are required to complete the project on time.

Critical Path Method (CPM)

Critical path method is a technique that determines those activities which have the least scheduling flexibility (i.e., critical activities). Note that if the critical activities are delayed, the entire project is delayed. After determining these activities, CPM specifies the project schedule according to the activities that lie on the critical path. The advantages of using critical path method are:

- It represents the project in a graphical form.
- It predicts the time required to complete the project.
- It specifies the critical activities.
- It specifies how to speed up the project so that it is completed on schedule.
- It specifies the optimal plan for speeding up the project.

Figure 4.8 shows the CPM diagram. It comprises activities and events which form a network. The activities are shown in circles (also known as nodes) and named as 'A' 'B,' 'C,' 'D,' 'E' and 'F.' The events begin from 'start' node and end at 'finish' node. The lines (or arcs) between the activities show the events. The time required to complete each activity is indicated along with it. First of all, activities 'A' and 'B' begin. Two activities, namely, 'C' and 'D,' are dependent on 'A.' Similarly, activity 'E' is dependent on 'B.' In other words, the activities 'C,' 'D' and 'E' cannot begin until activities 'A' and 'B' are complete. After the activities 'F,' 'D' and 'E' are complete, they reach the 'finish' node.

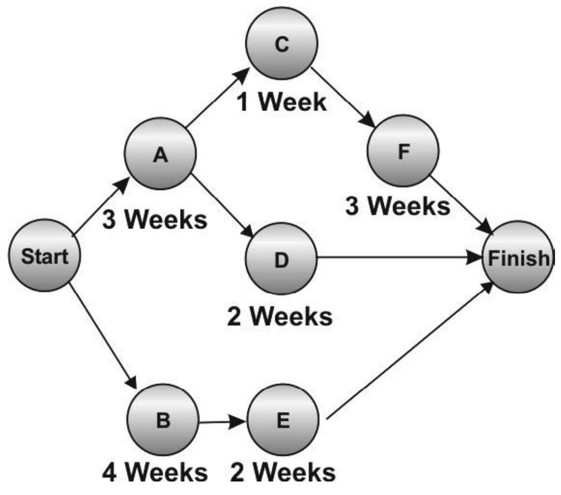


Fig. 4.8 *CPM Diagram*

To create a CPM diagram, follow these steps:

1. **Determine individual activities:** In this step, the activities determined with the help of work breakdown structure are described.
2. **Determine sequence of activities:** Here, the sequence of activities and the time taken to complete each activity are described. These activities are carried out both in series or in parallel. There are some activities which are dependent on other activities. For these activities, the dependency is determined according to which the sequence is specified.
3. **Prepare CPM diagram:** In this step, a CPM diagram is prepared after determining the critical activities and their sequence.
4. **Estimate activity completion time:** The completion time of each activity is estimated with the help of knowledge gained through experience by the organization or individuals. This is essential because the estimation of the completion time for the activity should be correct and there should be no variation in the completion time.
5. **Identify critical path:** Critical path is the path through the project network where no activity is slack. The amount of time for which a non-critical path activity can be delayed without delaying the project is known as *slack time*. Slack time for an activity is the time between its earliest start time and its latest start time or between its earliest finish time and its latest finish time. Critical path is identified by determining certain parameters of each activity. These parameters are:

- **Earliest start time (ES)**: It is the earliest start time when an activity can begin. It is assumed that the previous activities on which the activity depends are complete.
- **Earliest finish time (EF)**: It is equal to the sum of earliest start time for the activity and the time required to complete that activity.
- **Latest finish time (LF)**: It is the latest time at which the activity can be completed without delaying the project.
- **Latest start time (LS)**: It is equal to the difference of latest finish time and the time required to complete the activity.

6. **Update CPM diagram:** In this step, the information about actual completion time of tasks is gathered as the project progresses or when changes take place in it. According to the time taken to complete an activity, the CPM diagram is updated. In addition, the CPM diagram can be updated when there is a new critical path in the project.

Example of CPM

Consider an example of developing software. All the activities of software development are listed in Table 4.9, which describes the start time of each activity along with its duration. In addition, the type of activity and the dependent activity are specified.

Table 4.9 *Activities of Software Development*

Activity	*Activity Description*	*Start Time*	*Duration*	*Dependent on*
A	Analysis	Week 1	5 days	-
B	Selection of hardware platform	Week 1	2 days	A
C	Detailed analysis of modules	Week 1	2 weeks	A
D	Detailed analysis of supporting modules	Week 3	2 weeks	C
E	Hardware installation	Week 2	1 week	B
F	Programming of core modules	Week 3	3 weeks	C, E
G	Programming of supporting modules	Week 5	3 weeks	D, E
H	Quality assurance of core modules	Week 6	1 week	F
I	Quality assurance of supporting modules	Week 8	1 week	G
J	Documentation	Week 9	2 weeks	H, I
K	Users' training	Week 11	1 week	J

In Figure 4.9, events within the project are shown in circles. The circles are numbered to distinguish one activity from the rest of the activities. An arrow between two events shows an activity. The description of each activity and its duration are depicted along the arrow.

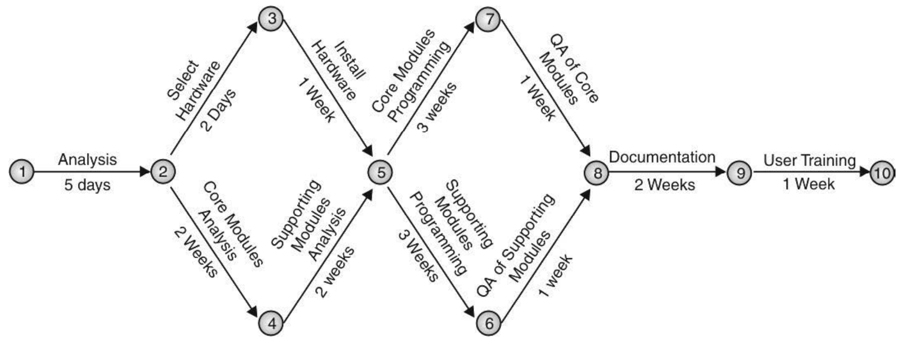


Fig. 4.9 *CPM Diagram for Software Development*

NOTES

Note that an activity cannot begin until the activity on which it is dependent is completed. For example, activities 'B' and 'C' (see Table 4.9) cannot begin until activity 'A' is completed. Similarly, activities 'D' and 'E' cannot begin until activities 'C' and 'B,' respectively, are not completed.

It may be observed from Table 4.9 and Figure 4.9 that the time required for activities 'B' and 'E' is less than that for activities 'C' and 'D.' Thus, activities 'B' and 'E' have slack time of almost 3 weeks. However, activities 'C' and 'D' are critical and any delay in these activities will delay the start of further activities, which in turn will delay the project.

It is essential that events '2' to '4' and '4' to '5' must be started and completed on time, since they are on critical path. Similarly, events '5' to '6,' '5' to '7,' '6' to '8,' '7' to '8,' '8' to '9' and '9' to '10' are on critical path. The critical path activities should be managed to ensure that they are completed on time. In case the critical path activities are delayed, proper measures should be taken to get the software development process back on schedule.

Similarities between PERT Chart and CPM Diagram

Sometimes, both PERT charts and CPM diagrams are used in combination and considered as one technique known as *PERT/CPM* technique. They are used together as they have some similarities that are:

- They define the activities and tasks of the project. Both PERT and CPM have a single 'start' and single 'finish' activity.
- They define relationships among the activities. In addition, both of them specify the sequence of activities.
- They prepare a diagram connecting all the activities.
- They assign the time and cost estimates to each activity.
- They determine the critical path in the entire network.
- They use the diagram to plan, schedule, monitor and control the project.

Task Network

Task network, also known as *activity network*, is a graphical representation of the flow of tasks in a project. With the help of task network, the sequence and dependency of tasks are determined. The tasks are divided according to the project. Once the tasks are identified, the task dependency chart is prepared. After this, the critical path of the project is determined. Table 4.10 lists the dependencies of tasks in the CPM diagram shown in Figure 4.10.

Table 4.10 *Task Duration*

Task	*Duration (in weeks)*	*Dependency*
A	3	None
B	4	None
C	1	A
D	2	A
E	2	B
F	3	C

To depict the hierarchy of tasks involved in the project, *work breakdown structure* is used, which provides information about the tasks along with their subtasks.

Work Breakdown Structure (WBS)

Work breakdown structure is the process of dividing the project into tasks and logically ordering them in a sequence. It provides a framework for keeping track of the progress of the project and for determining the cost planned for these tasks. By comparing the planned cost and the actual cost (cost that has been expended), the additional cost required can be controlled.

WBS specifies only the tasks that are to be performed and not the process by which the tasks are to be completed. It also does not specify the individuals performing that task. This is because WBS is based on requirements and not on the way in which the tasks are carried out.

To break the tasks involved in a project, follow these steps:

1. **Break the project into general tasks:** The project can be divided into general tasks such as analysis, design, testing, and so on.
2. **Break the general tasks into smaller individual tasks:** When the general tasks are determined, they can further be divided into subtasks. For example, design can further be divided into interface design, modular design, and so on.

Figure 4.10 shows an example of WBS. Here, the project summary is divided into three tasks, namely, *design phase*, *programming phase* and *testing phase*. Also, each task contains several subtasks. For example, design phase is further divided into two subtasks, namely, *first design phase* and *second design phase*. Note that in the figure, each task is numbered to indicate the sequence of tasks and subtasks. It also indicates the time taken and the costs involved to complete these tasks and subtasks.

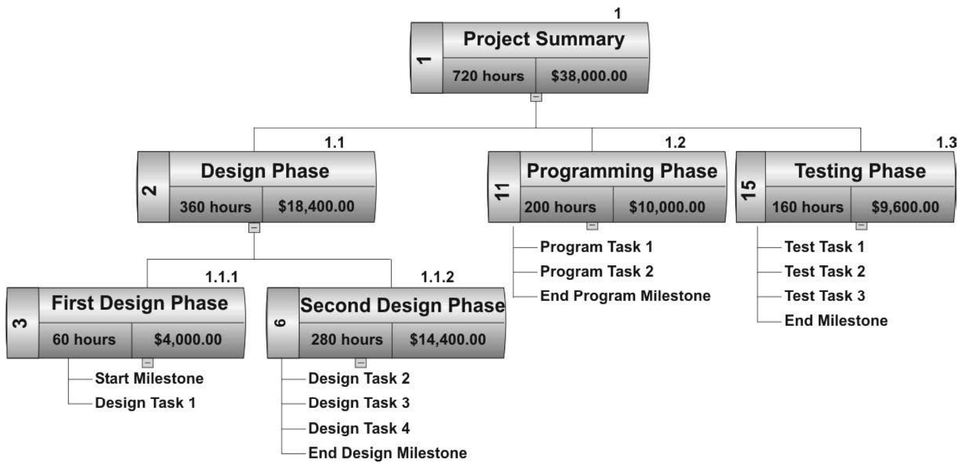


Fig. 4.10 *Example of Work Breakdown Structure*

Tracking the Schedule

As the project progresses, the project manager understands the activities to be completed and milestones to be tracked and controlled with the help of the project schedule. Tracking of the project schedule is done in the following ways:

- **Conducting periodic meetings with team members:** By conducting periodic meetings, the project manager is able to distinguish between completed and uncompleted activities or those that are yet to start. In addition, the project manager considers the problems in the project as reported by the team members.
- **Assessing the results of reviews:** Software reviews are conducted when one or more activities of the project are completed or when a particular development phase is completed. The purpose of conducting reviews is to check whether the software is developed according to user requirements.
- **Determining the milestones:** Milestones indicating the expected output are described. These milestones check the status of the project by comparing the progress of the activities with the estimated end date of the project.
- **Using earned value analysis to determine the progress of the project:** The progress of the project is determined quantitatively by the earned value analysis technique. This technique provides an estimate for every task, without considering its type, and the total hours required to accomplish the project. Based on this estimation, each activity is given an *earned value*, which is a measure of the progress and describes the percentage of the activities that have been completed.

Note: The completion of each task is indicated with the help of a milestone.

Project Monitoring

During the development of a project, it is important to check whether the plan is properly executed or not so that the management can modify the plan according to the requirements. A project works smoothly if all the requirements are fulfilled, and for this reason, it is important that all the methods required to monitor the project should be well-planned in advance. This helps in tracking the changes and deviations from the plan, thus helping the management in taking corrective action to handle those deviations.

To monitor the progress of activities against the plan and to ensure that the goals are achieved, a project monitoring plan is prepared. It is a document that provides guidelines for the proper execution of a project. There are various methods to assess the progress, cost and quality of the project. In this section, we will discuss some of the methods used for monitoring a project.

Time Sheets

One of the major tasks of the management include tracking the progress of the project as well as calculating the expenses incurred on the project. Progress can be monitored with the help of milestones set for the project and the earned value method (discussed later), while expenditures incurred are monitored by using time sheets. Time sheet is used to keep track of the time spent by each member of the project on different activities. This data represents the overall expenditure spent on different phases of the project at a given time.

Reviews

For effective monitoring and control, reliable information is needed so that the management can take effective decisions and can monitor the progress of a particular project. This information is provided by reviews that help to determine the progress of a project as per its planned schedule. Reviews serve three purposes—detecting defects, improving productivity and lowering costs. They are also a good tool in ensuring the quality of a software being produced. They can be used after each major development step to find and fix faults as soon as possible. They ensure that the product is complete and all its requirements have been met before the delivery.

Earned Value Method

Earned value method is a common measure for comparing the progress of the different activities of a project. It is used to assess the 'percent completeness' of a project. This method assigns a common value known as *earned value* for every task, which has an entry in *Summary Task Planning Sheet (STPS).* The total time spent to complete the project is estimated and every task is assigned an earned value on the basis of its estimated percentage contribution of the total. The total cost and time spent on a particular task can then be assessed from time sheets at any given time. By determining the difference between the earned value and the total cost spent, the project manager can determine the status of the project and can take necessary actions. At last, *Earned Value Summary Report* is prepared monthly or biweekly, which summarizes the earned value for each task and the actual effort spent on that task.

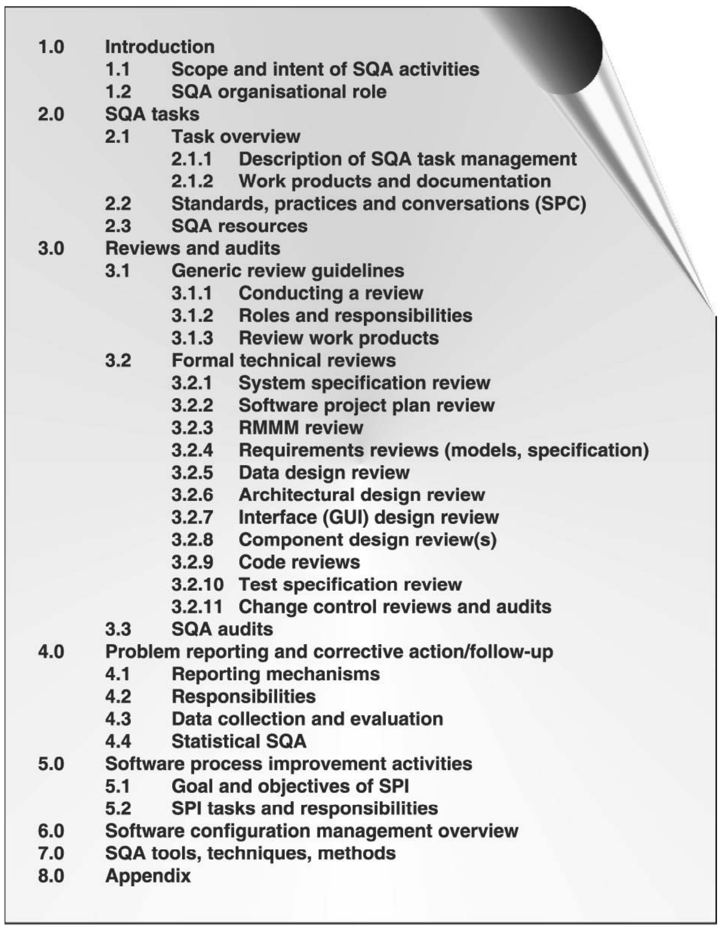
NOTES

Check Your Progress

1. What is project management?
2. Write the main purpose of project management.
3. What is project planning?
4. Define the term configuration management plan.
5. How will you define constructive cost model.
6. What is software metrics?

4.8 QUALITY ASSURANCE PLANNING

Quality planning is a structured process for defining the procedures and methods that are used to develop software. Quality planning starts in the early phases of software development. The software quality assurance (SQA) plan defines the activities for the SQA group. The objective of the SQA plan is to identify the quality assurance activities that are followed by the SQA group. Generally, the plan comprises documentation, such as project plan, test plans and user documents. Figure 4.11 shows an SQA plan designed by IEEE.



- 1.0 Introduction
  - 1.1 Scope and intent of SQA activities
  - 1.2 SQA organisational role
- 2.0 SQA tasks
  - 2.1 Task overview
    - 2.1.1 Description of SQA task management
    - 2.1.2 Work products and documentation
  - 2.2 Standards, practices and conversations (SPC)
  - 2.3 SQA resources
- 3.0 Reviews and audits
  - 3.1 Generic review guidelines
    - 3.1.1 Conducting a review
    - 3.1.2 Roles and responsibilities
    - 3.1.3 Review work products
  - 3.2 Formal technical reviews
    - 3.2.1 System specification review
    - 3.2.2 Software project plan review
    - 3.2.3 RMMM review
    - 3.2.4 Requirements reviews (models, specification)
    - 3.2.5 Data design review
    - 3.2.6 Architectural design review
    - 3.2.7 Interface (GUI) design review
    - 3.2.8 Component design review(s)
    - 3.2.9 Code reviews
    - 3.2.10 Test specification review
    - 3.2.11 Change control reviews and audits
  - 3.3 SQA audits
- 4.0 Problem reporting and corrective action/follow-up
  - 4.1 Reporting mechanisms
  - 4.2 Responsibilities
  - 4.3 Data collection and evaluation
  - 4.4 Statistical SQA
- 5.0 Software process improvement activities
  - 5.1 Goal and objectives of SPI
  - 5.2 SPI tasks and responsibilities
- 6.0 Software configuration management overview
- 7.0 SQA tools, techniques, methods
- 8.0 Appendix

Fig. 4.11 *Quality Plan*

NOTES

This plan comprises eight sections. The description of each section is given as follows:

- **Introduction**: It provides an overview of the entire SQA plan and SQA activities being performed in software project. In addition, it specifies the role of an organization in SQA.
- **Tasks**: These provide information about SQA tasks performed in SQA plan. It assigns SQA tasks to software engineers and to the SQA group. In addition, it describes SQA resources, such as hardware, software and tools to perform SQA tasks.
- **Reviews and audits**: These describe project reviews conducted by the SQA group. For performing reviews and audit, a set of guidelines is used. Furthermore, it provides information on formal technical reviews. Audits are used to assess SQA activities conducted in the project.
- **Problem reporting and corrective action/Follow-up**: This describes problem-reporting mechanisms that occur as a result of reviews and provides corrective actions to improve them. Reporting mechanisms include the process of reporting problems (such as errors in software) to SQA group. In addition, it collects erroneous data and finds measures to correct errors.
- **Software process improvement activities**: These describe the activities required for software process improvement (SPI). It also outlines the objectives and tasks of SPI.
- **Software configuration management (SCM) overview**: This overview of the SCM plan provides information, such as description of configuration management process and procedures to perform SCM.
- **SQA tools, techniques and methods**: It describes the SQA tools, techniques and methods that are used by SQA group.
- **Appendix**: It provides additional information about SQA plan.

4.9 RISK ANALYSIS

Risk analysis is a technique to identify and assess factors that may jeopardize the success of a project or achieving a goal. This technique also helps to define preventive measures to reduce the probability of these factors from occurring and identify countermeasures to successfully deal with these constraints when they develop to avert possible negative effects on the competitiveness of the company. One of the more popular methods to perform a risk analysis in the computer field is called Facilitated Risk Analysis Process (FRAP).

Risks can come from uncertainty in financial markets, threats from project failures (at any phase in design, development, production or sustainment life cycles), legal liabilities, credit risk, accidents, natural causes and disasters as well as deliberate attack from an adversary, or events of uncertain or unpredictable root cause. Several risk management standards have been developed. Risk analysis should be performed as part of the risk management process for each project.

Analysis and Factors of Risk Measurement

The following approaches are used to measure market risk:

1. The Notional Amount Approach

Under the notional amount approach, risk is measured as the adverse change in the notional or nominal amount of a security. In case of a portfolio of securities, the risk represents the change in the sum of the notional values of securities in the portfolio. For instance, if the portfolio consists of a five year government bond with a nominal value of ₹ 10 million and a ten year government bond with a nominal value of ₹ 15 million, the nominal value of the portfolio is considered to be ₹ 25 million. The risk represents the adverse change in the value of this portfolio.

This method is not appropriate for risk measurement as it simply adds the positions whether short (sale) or long (purchase). It also does not consider the correlation between prices of securities comprised in a portfolio. In case of derivative securities, there is a significant difference in the notional amount and the true amount of market exposure. Further, the risk exposure of options depends on whether these are in the money or out of money. In case of swaps, some swaps are written to hedge the risk arising from other swaps. Simply adding the notional amounts of such securities does not reflect on the true exposure to risk.

2. Factor Sensitivity Approach

Measures of factor sensitivity capture the sensitivity of an instrument or portfolio to changes in the value of primary risk factors such as interest rates, yield to maturity, volatility, stock price, stock index, and so on. For fixed-income products, a popular risk measure among traders is 'DV01', also known as 'value of 01'. 'DV01' is a trader's abbreviation for the change ('delta') in value consecutive to a change in yield of one basis point i.e., 1 per cent of a percentage point. The measure is consistent with the conventional 'duration' analysis of a bond.

The interest rate sensitivity of bond prices to parallel shifts in all the interest rates is measured by 'duration'. If the duration of a bond is 3, it means the bond value will change by 3 per cent, when the interest rates change by 1 per cent. The duration measure suffices only for small parallel changes in interest rates. When the shape of the yield curve also changes, a convexity adjustment is required. For stocks, the sensitivity of movement of stock prices to movements in the market index is measured by 'beta'. If a stock has a beta of 1.5, a change in the index return of 1 per cent results in a 1.5 per cent increase in the return from that stock. Beta is used as a risk measure in the popular Capital Asset Pricing Model (CAPM) to calculate the risk premium required from a stock by investors. Beta is the coefficient of regression resulting from regressing historical stock returns against similar index returns.

Options refer to the right and, not the obligation, to buy or sell some underlying asset. The value of options depend on a number of parameters that form a part of the Black-Scholes model. These parameters include the value of the underlying asset, the horizon to maturity, the volatility of the underlying asset and the risk-free interest rate. Option prices are sensitive to all these parameters and these sensitivities are known as 'Greek letters'.

The sensitivity, with respect to the underlying asset, is the 'delta' δ. Delta is low if a call option is 'out of the money' (asset price below strike price) as there is no gain from the exercise of the option. Delta gets closer to 1 if there is a gain from the exercise of the option, i.e., when the option is 'in the money'. For instance, if the asset price is ₹ 110 and the strike price is ₹ 100, the gain also increases by Re 1, if the asset price increases by Re 1.The sensitivity varies between 0 and 1.

Gamma (γ) measures the degree to which the option's delta changes as the price of the underlying asset changes. The higher the gamma, the more valuable the option is to its holder. For a high gamma option, when the underlying price increases, the delta also increases, so that the option appreciates more in value than for a gamma neutral position. Conversely, when the underlying price declines, the delta also falls and the option loses less in value than if the position was gamma-neutral. The reverse is true for short positions in options; high gamma positions expose their holders to more risk than gamma-neutral positions.

The option is also sensitive to the time for maturity because the longer the horizon, the higher the chances that the stock moves above the strike price. This sensitivity is denoted by theta (θ). The shorter the time to maturity, the lower is the value of the option. The change in the value of the option due to passage of time is the 'time decay' of the option value.

The volatility of the underlying asset also affects the value of the option. Vega (υ) measures the sensitivity of the option value to changes in the volatility of the underlying asset. A higher vega typically increases the value of the option to its holder.

Rho (ρ) measures the change in value of an option in response to a change in interest rate, more specifically a change in the zero-coupon rate of the same maturity as the option. Since the payoff from an option occurs in the future, discounting is required to calculate its value today. Therefore, the higher the value of rho, the lower is the value of the option to the holder.

Although the Greek letters individually provide a measure of some risk, these cannot be aggregated to provide a measure of the overall risk of a portfolio. Sensitivities cannot be aggregated for different types of risk on the same instrument. For instance, delta and gamma on the same option cannot be aggregated. Similarly, sensitivities cannot be aggregated for the same risk on different instruments. For instance, delta of a stock option and delta of a currency option cannot be aggregated to determine the portfolio risk.

Since the sensitivities cannot be aggregated, they cannot be used to assess the magnitude of the overall loss that might arise from a change in the risk factors. Therefore, sensitivities cannot be used directly to measure capital at risk. Sensitivities can also not be used for risk control. Position limits expressed in terms of delta, gamma and vega are often ineffective since they do not translate easily into a 'maximum loss acceptable' for the position.

3. Value at Risk

Value at Risk (VaR) summarizes the expected maximum loss (or worst loss) over a target horizon (a single day or any other period) within a given confidence level. VaR is the answer to the following question:

What is the maximum loss over a given time period such that there is a low probability, say a 1 per cent probability that the actual loss over the given period will be larger?

For instance, if we say that a position has a daily VaR of ₹10 lakh at the 99 per cent confidence level, we mean that the realized daily losses from the position will, on an average, be higher than ₹ 10 lakh on only one day every 100 trading days.

VaR is the loss corresponding to (100-C)th percentile of distribution of change in the value of the portfolio over a specified number of days where C is the confidence level. This is illustrated in Figure 4.12.

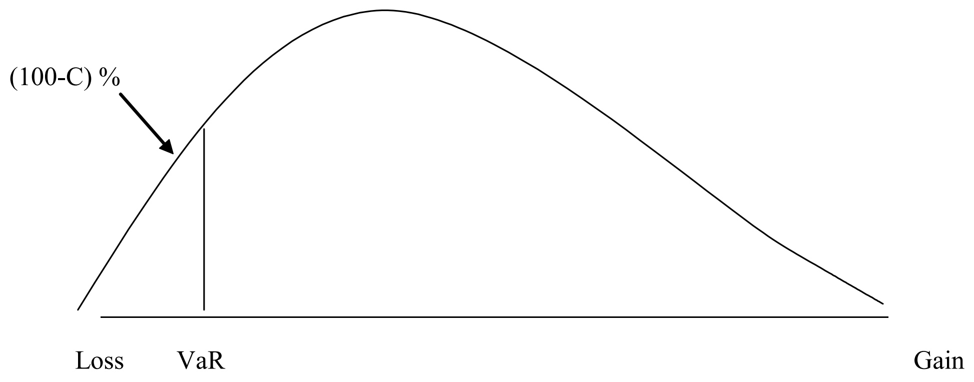


Fig. 4.12 Calculation of VaR from the Probability Distribution of the Change in the Portfolio Value

VaR depends on the chosen confidence level and the time horizon. The higher the confidence level, the greater is the VaR measure. The choice of the confidence level depends on the use of VaR. If it is simply used as a benchmark measure of downside risk for different trading desks or over time, the only requirement is that the confidence level should be the same for all. However, if the VaR measure is used to decide how much capital needs to be set aside to avoid bankruptcy, a higher confidence level should be used. A very high confidence level should be avoided as it adversely affects the verifiability of the VaR measure. If the confidence level is fixed at 99.99 per cent, there will be only 0.01 per cent probability (1 day in 10,000 trading days or once in 40 years) of the VaR being exceeded. This will make it impossible to verify if the true probability associated with VaR is in fact 99.99 per cent.

As in the case of confidence level, the VaR is higher for a longer horizon. The choice of the horizon depends on the characteristics of the portfolio. If the positions are subject to quick changes or if exposures change with changes in the prices of the underlying assets (as in the case of option deltas), a smaller horizon should be used. The horizon should also be short if the purpose of VaR measure is to provide an accurate benchmark measure of downside risk. However, a long horizon should be used where the purpose is to decide on the amount of capital to be set aside to avoid bankruptcy. The frequency at which profit and loss is measured also influences the choice of the horizon. Banks generally calculate profit and loss on a daily basis while others use a longer period. The minimum horizon should be equal to this interval.

Parametric and Non-Parametric VaR

When VaR is derived from a distribution that is constructed using historical data, it is called non-parametric VaR. If the VaR is derived assuming a particular theoretical distribution of the returns, it is called parametric VaR. In this case, historical data is used to estimate the parameters of the assumed distribution function. If a normal distribution is assumed, *VaR* over a horizon *H* and for a confidence level *c*, is given by the following equation:

$$VaR(H{:}c) = \alpha\,\sigma\,V$$

where,

α = The standard normal variate corresponding to the confidence level chosen.

σ = The standard deviation of returns of the asset on a given horizon.

V = The current market value of the asset.

Types of VaR

One-day VaR is derived from the daily distribution of the portfolio values. However, ten days is the time horizon set by the regulators for the purpose of reporting regulatory capital. The 10-day VaR can be derived either from the corresponding distribution over a 10-day horizon or the 10-day VaR can be approximated by multiplying the daily VaR by the square root of time (here, ten days).

$$VaR\,(10; c) = \sqrt{10}\;VaR\,(1;c)$$

This approximation can be used provided the daily returns are independent and identically distributed (i.i.d.). The result will not hold if the stock returns exhibit mean reversion, are serially correlated, or more generally, are not i.i.d.

Calculating Value at Risk

The models used to calculate *VaR* have been discussed in this section.

Parametric Normal Models

1. Portfolio Normal Approach

The portfolio normal approach calculates VaR as a multiple of the standard deviation of the portfolio returns. VaR over a horizon H and for a confidence level c, is given by the following equation:

$$VaR(H{:}c) = \alpha\,\sigma_p\,V$$

where,

α = the standard normal variate corresponding to the confidence level chosen.

σ = the standard deviation of returns of the asset on a given horizon.

V = the current market value of the asset.

This approach assumes a normal distribution of portfolio returns. It has been empirically observed that return distributions are not normal. Return distributions exhibit 'fat tails'. A fat-tailed distribution is characterized as one that has more observations far away from the mean than is the case in a normal distribution. So where a normal distribution tails off quickly to reflect rarity of unlikely events, the tail of a fat-tailed distribution remains relatively thick. Fat tails in distributions are particularly worrying to risk managers because they imply that extraordinary losses occur more frequently than a normal distribution would imply.

However, even if the returns of an individual factor do not follow a normal distribution, the returns of a well-diversified portfolio across many risk factors might still exhibit a normal distribution. This effect is explained by the central limit theorem which tells us that the independent random variables of well-behaved distributions will possess a mean that converges, in large samples, to a normal distribution. This result implies that a risk manager can assume that a portfolio has a normal returns distribution, provided that the portfolio is well diversified and the risk factor returns are sufficiently independent from each other.

This approach is very simple to apply and does not require the estimation of a large number of correlations among asset returns. It can, however, be used if the portfolio either consists of a small number of assets whose returns are normally distributed or the portfolio is made of a large number of positions whose limiting distribution is the normal distribution. This approach is not appropriate where the portfolios have a large number of non-linear assets such as options.

2. Asset Normal Approach

This method also calculates VaR as a multiple of the standard deviation of the portfolio returns like the portfolio normal approach. The only difference is that the standard deviation of the portfolio returns is calculated using a set of portfolio weights and the covariance matrix of position returns. This approach is derived from the modern portfolio theory of Harry Markowitz.

This method assumes that the individual asset returns are normally distributed. Since the portfolio return is a linear combination of asset returns, it is also normally distributed. The variance–covariance matrix is constructed using observed correlations and variances of assets.

Consider a portfolio that consists of N securities each having a weight w_i ($i = 1, 2, —, N$) and return r_i. The portfolio return is given by the following equation:

$$R_p = \sum_{i=1}^{N} w_i r_i$$

The portfolio variance will be as follows:

$$\sigma_p^2 = w\Sigma w^T$$

where,

σ_p^2 = The portfolio variance.

w = ($w_{1—}w_N$) is $1 \times N$ weight vector.

Σ = The variance–covariance matrix.

w^T = The transpose of w

The one day and ten day VaRs for the portfolio that have a value V at 99 per cent confidence level are, respectively:

$$VaR_p(1;99) = 2.33 * \sigma_p V$$

$$VaR_p(10;99) = \sqrt{10} VaR_p(1;99) = 2.33 * \sqrt{10} * \sigma_p V$$

The following are the limitations of this approach:

- It assumes that asset returns are normally distributed.
- It is a linear method, and therefore, not suited to assets that have a non-linear pay-off.
- It requires computation of a large number of correlations and variances.

Illustration 4.1:

The daily mean returns and standard deviations of two stocks S_1 and S_2 and the correlation coefficient between their rates of return are as follows:

$$\mu_1 = 0.15\%$$
$$\mu_2 = 0.04\%$$
$$\sigma_1 = 2.50\%$$
$$\sigma_2 = 2.00\%$$
$$\rho = 0.20$$

Assume that the portfolio is composed of 100 shares of S_1 and 150 shares of S_2, valued at ₹ 50 and ₹ 100, respectively.

The value of the portfolio is:

$$V = n_1S_1 + n_2S_2 = ₹\ 20{,}000$$

And, the relative investments in both stocks are, respectively:

$$w_1 = \frac{n_1S_1}{V} = 0.25$$

$$w_2 = \frac{n_2S_2}{V} = 0.75$$

so that the one-day mean rate of return and the standard deviation of the rate of return on the portfolio are as follows:

$$\mu_p = w_1\mu_1 + w_2\mu_2 = 0.0675\%$$

$$\sigma_p^2 = w\Sigma w^T$$

$$\sigma_p^2 = \begin{bmatrix}0.25 & 0.75\end{bmatrix}\begin{bmatrix}0.025^2 & 0.025\times0.02\times0.2\\ 0.02\times0.025\times0.2 & 0.02^2\end{bmatrix}\begin{bmatrix}0.25\\ 0.75\end{bmatrix}$$

$$= 0.0003$$

$$\sigma_p = 1.73\%$$

The one-day VaR at 99 per cent confidence level is as follows:

$V_aR_p\ (1;99) = 2.33\ \sigma_pV = ₹\ 806$

3. Delta Normal Approach

Even with a limited number of assets, modelling all individual asset returns within a portfolio requires a large amount of computation. A simplified approach consists of identifying the basic market rates and market prices that affect the value of assets. These basic market rates are known as market factors or risk factors.

Since the number of market factors is much smaller than that of individual assets, it is more efficient to derive asset returns from the time path of market factors that influence their values. This necessitates the modelling of relations between asset returns and market factor returns. The process requires the following two steps:

(i) **Mapping**: Mapping involves identifying those market factors which influence values (for instance, interest rates, stock indexes, volatilities). In most of the cases, relations between asset values and market factors have a linear approximation. In some cases, the relationship is stochastic, indicating the presence of an error term reflecting the asset-specific risk that must be accounted for. In other cases such as bonds and simple derivatives, the relation results from a closed-form formula.

(ii) **Modelling the Sensitivity of Individual Asset Values to Market Parameters:** For stocks, the sensitivity measure used is beta. Stocks also carry specific risk unrelated to the market parameter. Basis point measure (DV01), duration and convexity are used as sensitivity measures in the case of bonds and loans. Greek letters constitute the sensitivity measures in the case of options. These sensitivity measures are also known as delta parameters because they link a specific asset's price changes to market factors' price changes. The standalone risk of an asset results directly from the sensitivities of individual assets to market factors and volatilities of market factors.

$$\sigma(r_i) = S_i \, x \, \sigma\left(\Delta mk / mk_0 \right)$$

where,

$\sigma(r_i)$ = The volatility of return of asset *i.*

S_i = The sensitivity of return of asset i to returns of market factor *mk.*

$\sigma(\Delta\, mk/mk_0)$ = The volatility of market factor.

Assuming a normal distribution of asset returns,

$VaR = \alpha\, \sigma(r_i)\, V_i$

where,

α = Value of standard normal variate at the given confidence level.

V_i = The initial value of asset *i*.

For instance, if the volatility of market factor $\sigma(\Sigma mk/mk_0)$ is 16 per cent and the sensitivity of returns of asset i to market factor returns is 1.5, the volatility of returns of asset i would be,

$$1.5 \times 16\% = 24\%$$

If the initial value of the asset is ₹ 1,000, then the VaR at 99 per cent confidence level is 2.33 × 0.24 × 1,000 = ₹ 559.

The Delta VaR technique significantly simplifies the VaR calculation and is well suited for portfolios with no or very few options. It, however, suffers from the following limitations:

- It assumes constant sensitivities of asset returns to market factor returns. The technique does not work when sensitivities cannot be considered constant; for instance, in the case of options. In the case of stocks also, if only sensitivities are used, the risk unrelated to market parameters or the stock specific risk will stand ignored.
- Use of proxy mapping to market parameters generates basis risk. Basis risk implies that the relationship assumed between asset values and market factors may not hold in the future.
- It ignores the actual distributions of market factors and assumes that these are normal.

Portfolio Risk Using Delta Normal Approach

The goal of modelling portfolio risk is to obtain the distribution of portfolio returns at the horizon. The distribution of portfolio returns depends on the correlation between individual asset returns. Since market factors drive the value of individual assets, it is necessary to model random deviations in market factors that comply with their correlation structure. To model correlations, the common principle for all portfolio models is to relate the individual risks of each transaction to a set of common factors.

For market risk, the market values of individual transactions are sensitive to risk factors that alter their values. When all risk factors vary, the dependence of individual asset returns on this set of common factors generates correlation between them. It is, therefore, necessary to capture the interdependencies between market factors to correlate future values of market factors and assets.

In the case of bonds and derivatives, pricing models relate, in a deterministic way, the prices of the assets to the market factors that drive them. In such cases, it is sufficient to correlate the market factors to obtain correlated individual asset returns or values using the pricing models with correlated market factor values as inputs.

When the relationship to market factors is stochastic, for instance, between stock returns and equity indexes, there is an error term representing the random component of individual asset returns unrelated to market factors, which is also to be measured. In such a case, a diagonal model is used. The interdependence between market factors is represented in a variance–covariance matrix. The process of measuring the portfolio market risk comprises of the following two steps:

(i) Imposing the variance–covariance structure on market factor deviations.

(ii) Modelling the portfolio return distribution when all market factors vary in compliance with such variance–covariance structure.

Under the Delta VaR technique, a linear relationship between asset values and underlying market factors is assumed. This restrictive assumption can be bypassed using a full-fledged simulation approach. However, in both the cases, the prerequisite is to have the correlations and the variance–covariance matrix of all relevant market parameters.

Correlations and variances of individual asset returns and market factors are observable. Therefore, the first technique for obtaining the matrix is to measure these through direct observations, usually on historical basis. The difficulties with this method are that the number of assets is large and the observed variances and covariances may not form a consistent matrix.

It is more efficient to derive correlations from the correlations of factors driving these returns. This option requires a prior mapping of asset returns to market factors. For market risk, the risk factors are market factors or other risk drivers. These factors and risk drivers are not necessarily identical. For instance, it is possible to identify a number of factors that influence the equity index returns or the interest rates. Equity indexes and interest rates are direct drivers of asset returns. On the other hand, various factors influence risk drivers without interacting directly with returns as risk drivers do. They serve to correlate the distribution of risk drivers. The common dependence of risk drivers, such as interest rates on a set of common factors, makes them correlate. Hence, we need to distinguish the following three levels:

$$\text{Factors} \longrightarrow \text{Risk drivers} \longrightarrow \text{Individual risks}$$

In many instances, risk factors are identical to risk drivers for market risk. They are all market parameters such as yield curves, foreign exchange rates, equity indexes and their volatilities. Since market parameters are directly observable, there is no need to model them. To model correlations between individual assets and risk drivers, factor models are used. A single factor model for stocks is as follows:

$$r_i = \alpha_i + \beta_i r_m + \varepsilon_i$$

where,

r_i	= Return on asset *i*.
α_i	= Constant term
β_i	= Asset beta representing sensitivity of asset returns to market index returns.
r_m	= Return on the market index.
ε_i	= Error term representing asset returns unrelated to market index returns.

This model simplifies the construction of the variance–covariance matrix. The covariance between any two stocks under the one-factor model is given by the following equation:

$$Covariance(r_1, r_2) = \beta_1\, \beta_2\, \sigma^2 r_m$$

The variance of the stock is as follows:

$$\sigma^2(r_i) = \beta_i^2 \sigma_{rm}^2 + \sigma^2 \varepsilon_i$$

The variance and covariance terms calculated form the inputs for a diagonal variance–covariance matrix for the entire portfolio which is given as follows. The covariance terms occupy the off-diagonal positions and the variance terms occupy the diagonal position.

NOTES

$$\Sigma = \begin{bmatrix} \beta_1^2\sigma_{rm}^2 + \sigma^2\varepsilon_1 & \beta_1\beta_2\sigma_{rm}^2 --- & \beta_1\beta_N\sigma_{rm}^2 \\ \beta_2\beta_1\sigma_{rm}^2 & \beta_2^2\sigma_{rm}^2 + \sigma^2\varepsilon_2 --- & \beta_2\beta_N\sigma_{rm}^2 \\ \\ \beta_N\beta_1\sigma_{rm}^2 & \beta_N\beta_2\sigma_{rm}^2 & \beta_N^2\sigma_{rm}^2 + \sigma^2\varepsilon_N \end{bmatrix}$$

In addition to the single factor model, multiple factor models (such as the Arbitrage Price Theory or the APT model) are also available to derive the variance–covariance matrix. The variance–covariance structure so developed is then used to arrive at a VaR measure for the portfolio.

Illustration 4.2:

The following are estimates of two stocks.

Stock	Expected annual return	Beta	Firm-specific standard deviation
A	13%	0.8	30%
B	18%	1.2	40%

The market has a standard deviation of 22 per cent and the risk-free rate is 8 per cent. The portfolio has the following proportions:

Stock A: 30%

Stock B: 45%

Treasury bills: 25%

Thus, the variance of the portfolio is as follows:

$$\sigma_p^2 = w\Sigma w^T$$

The variance–covariance matrix 'Σ' is derived as follows:

$$\Sigma = \begin{bmatrix} 0.8^2 x\, 0.22^2 + 0.3^2 & 0.8x1.2x0.22^2 & 0.8x0x0.22^2 \\ 1.2x0.8x0.22^2 & 1.2^2 x0.22^2 + 0.4^2 & 1.2x0x0.22^2 \\ 0x0.8x0.22^2 & 0x1.2x0.22^2 & 0^2 x0.22^2 + 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0.121 & 0.0465 & 0 \\ 0.0465 & 0.230 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The weight vector 'w' is given by $\begin{bmatrix} 0.30 & 0.45 & 0.25 \end{bmatrix}$.

The transpose of the weight vector w^T is given as follows:

$$\begin{bmatrix} 0.30 \\ 0.45 \\ 0.25 \end{bmatrix}$$

The portfolio variance (σ_p^2) is equal to

$$\begin{bmatrix}0.30 & 0.45 & 0.25\end{bmatrix}\begin{bmatrix}0.121 & 0.0465 & 0\\ 0.0465 & 0.230 & 0\\ 0 & 0 & 0\end{bmatrix}\begin{bmatrix}0.30\\ 0.45\\ 0.25\end{bmatrix}$$

$= 0.0698$

The portfolio risk (σ_p) = $\sqrt{0.0698}$

$= 0.264$ or 26.4%

The standard deviation of the portfolio is the annual standard deviation. It can be converted into daily standard deviation by applying the square root of time rule.

4. Delta–Gamma Approach

The Delta-Portfolio, Asset-Portfolio and Delta-Normal approaches are linear methods. These are not appropriate when applied to portfolios that generate non-linear returns such as portfolios containing options, callable or convertible bonds and mortgage-backed securities with large convexity, and so on. Two different approaches are used in such cases—delta-gamma methods and simulation techniques. This approach takes into account both the delta and gamma of options contained in the portfolio.

Measuring Volatilities

The volatility is a basic ingredient of all processes leading to the horizon value distribution. Volatility captures the variations around the average of any random parameter or target variable, both upward and downward; for instance, losses for credit risk. It characterizes the stability or instability of any random variable. It is a common statistical measure of the dispersion around the average of any random variable such as market parameters, earnings and so on. Volatility is the standard deviation of the value of these variables.

Variance (or standard deviation) measures historical volatility, which is a static measure of variability of security returns around their mean. It does not utilize current information to update the estimate.

The historical volatility estimates rely on strong assumptions of the distributional properties of security returns, i.e., they are independently and identically distributed (**i.i.d.**). The identically distributed assumption means that the mean and variance of returns do not vary across time. The independence assumption means that speculative price changes are unrelated to each other at any point of time. These two conditions form the basis of the random walk model (successive price changes are not related). If these assumptions hold, volatilities over periods greater than a day are calculated using square root of the time rule. Assuming that changes over two time periods are independent, variance of value over period (0,2) should be the sum of variances over the period (0,1) and the period (1,2). If volatility σ is assumed to be constant, $\sigma_{0,1} = \sigma_{1,2} = \sigma$

$$\sigma^2(0,2) = \sigma^2 + \sigma^2 = 2\sigma^2$$

$$\sigma(0,2) = \sqrt{2}\sigma$$

However, these assumptions do not hold true in reality. Volatilities are not stable over time. If the square root of the time rule is followed, volatility would become infinite over a long horizon, which is unacceptable.

To deal with the issue of instability of volatility, some techniques use moving averages. Moving averages minimize deviations from the long-term mean. Both arithmetic and Exponentially Weighted Moving Averages (EWMA) are used.

Investigating the validity of the independent assumption has focused on testing for serial correlation in changes in price. The correlation of a variable with itself over successive time intervals is serial (auto) correlation. The general conclusion reached by past investigations is that successive price changes are auto correlated but are too weak to be of economic importance. This observation has led most investigations to accept the random walk hypothesis. However, evidence has shown that a lack of autocorrelation does not imply the acceptance of independent assumption. Some investigations have found that security returns are governed by non-linear processes that allow successive price changes to be linked through the variance. This phenomenon was observed in the pioneering investigations of the 1960s. More convincing evidence of this phenomenon was provided by later investigations, which observed nonstationarity (change over time) in the variances.

The time varying nature of the variance may be captured using conditional time series models such as ARCH (Auto Regressive Conditional Heteroscadasticity) and GARCH (Generalized Auto Regressive Conditional Heteroscedasticity). In an auto regressive model, the explanatory variables are lags of the dependent variable (ordinary regression has a dependent variable Y and explanatory variable X. In auto regression, the dependent variable is Y and the explanatory variable is the previous period value of Y). Heteroscadasticity means changing variance. Conditional means stochastic (random) volatility and unconditional means constant volatility. In case of unconditional volatility, the same distribution governs each data point while conditional distribution changes at each point in time.

ARCH models can scientifically validate a key characteristic of time series data that 'large changes tend to be followed by large changes and small changes tend to be followed by small changes'. This is often referred to as the clustering effect. The usefulness of ARCH models relate to their ability to deal with this effect by using squared past forecast errors to predict future variances.

There are many extensions of ARCH, of which GARCH is the most popular. The GARCH (1,1) model assumes that the conditional variance depends on the latest innovation (previous day's return) and one lag of the previous forecast of conditional variance. The model equation is as follows:

$$h_t = \alpha_0 + \alpha_1 r_{t-1}^2 + \beta h_{t-1}$$

The variance h_t is conditional variance using information up to time t–1, and r_{t-1} is the previous day's return, also called innovation.

The average, unconditional variance is found by setting $E[r_{t-1}^2] = h_t = h_{t-1} = h$. Solving for h, we find the following:

$$h = \frac{\alpha_0}{1 - \alpha_1 - \beta}$$

This model is stationary when the sum of the parameters α_1 and β is less than one. The sum is also called the persistence, as it defines the speed at which shocks to the variance revert to their long-run values.

Illustration 4.3:

Consider the following parameters of a GARCH process:

$\alpha_0 = 0.02$
$\alpha_1 = 0.04$
$\beta = 0.94$
$h_0 = 1.44$

The next day's return is 4 per cent. The new variance forecast is as follows:

$$h_1 = 0.02 + 0.04 \times 4^2 + 0.94 \times 1.44 = 2.01$$

The conditional volatility $\sqrt{h_0} = 1.20$ has increased to $\sqrt{h_1} = 1.42$ due to a large innovation (return) the previous day. If there is no change in the price the next day or the return is zero, the new variance forecast would be $h_2 = 0.02 + 0.04 \times 0^2 + 0.94 \times 2.01 = 1.91$ and volatility $\sqrt{h_2}$ would be 1.38.

Other variations of GARCH such as TGARCH and EGARCH differentiate between the effect of positive and negative changes in the previous period on the current volatility. These models assume that downward movements in the market are often followed by higher volatility than upward movements of the same magnitude.

ARCH and GARCH models have become widespread tools for dealing with heteroscedastic time series data. The goal of such models is to provide a volatility measure like a standard deviation that can be used in financial decisions concerning risk analysis, portfolio selection and derivative pricing.

A third technique is to use the implicit volatilities embedded in option prices. Pricing models of options are used to derive the implicit volatility. Implicit volatilities are based on future expectations and are, therefore, forward looking. However, as the option prices fluctuate, implicit volatilities turn out to be unstable.

Non-Parametric (Simulation) Models

Non-parametric models do not assume any particular theoretical distribution of returns. These models, though computationally more intensive than parametric models, overcome many of their limitations. These models rely on historical or Monte Carlo simulation. Accordingly, there are two classes of such models. They are given as follows:

(i) Historical Simulation Models.

(ii) Monte Carlo Simulation Models.

Simulation models are full valuation models. Every time a change of market factors is simulated, all the assets in the portfolio are simultaneously re-priced. There is no linear and quadratic approximation as in the case of parametric methods.

The simulation models, therefore, produce the true price change of the portfolio rather than an approximation. This feature proves to be very useful when the relationship between market factors and assets is non-linear as in the case of options.

Simulation models model asset returns instead of asset values as returns are much easier to model than prices. Suppose V_0 is the current value and V_H is the value at the horizon such that $V_H = V_0(1 + r)$, r being the return between the current date and the horizon. The change in value is given as follws:

$$\Delta V = (V_H - V_0) = V_0 (1 + r) - V_0 = V_0 \times r$$

Therefore, in order to measure risk, what is needed is the distribution of asset returns between now and the horizon.

Parametric methods calculate VaR analytically, using confidence intervals, assuming normality of asset returns or factor returns. In simulation models, VaR is calculated by building up a probability distribution of asset returns and identifying the required percentile.

Simulation models do not make any assumption about normality of asset returns or market factor returns. If the actual probability distribution of returns is not normal, conclusions based on normality assumptions are misleading. Historical simulation models use the actual distribution of returns taken from the past. Monte Carlo models have to choose a probability distribution to generate simulations.

1. Historical Simulation Approach

The historical simulation approach does not require any analytic assumptions about the distributions. VaR is derived from the empirical distribution generated by the historical realizations of the risk factors over a specified period of time. However, at least two or three years of data is necessary to produce meaningful results.

The historical simulation approach to VaR is conceptually simple. First, the changes that have been seen in the relevant market prices and rates (the risk factors) are analyzed over a specified historical period, say, one to four years. The portfolio under examination is then revalued, using changes in the risk factors derived from the historical data, to create the distribution of the portfolio returns from which the VaR of the portfolio can be derived. Each daily simulated change in the value of the portfolio is considered as an observation in the distribution. The following three steps are involved in the process:

(i) Select a sample of the actual daily risk factor changes over a given period of time; say 250 days (i.e. one year's worth of trading days).

(ii) Apply those daily changes to the current value of the risk factors and revalue the current portfolio as many times as the number of days in the historical sample.

(iii) Construct the histogram of portfolio values and identify the VaR that isolates the first percentile of the distribution in the left hand tail, assuming VaR is derived at the 99 per cent confidence level.

Illustration 4.4: Assume that the current portfolio is composed of 1,00,000 shares of stock of company and the current price is ₹ 26.30 per share.

Table 4.11 *Historical Market Values for Risk Factors over the Last 100 Days*

Day (t)	Stock Price (₹)
-100	25.00
-99	25.20
-98	24.90
-97	24.80
---	---
-3	26.75
-2	26.10
-1	26.25
0	26.30

Table 4.11 shows observations on the stock price for the last 100 days. Historical simulation requires re-pricing of the position using the historical distribution of the risk factors. Table 4.12 shows the value of the stock, assuming that the successive changes from now onwards are the same as those that happened over the last 100 days. The change in stock price between day -100 and day -99 is from ₹ 25.00 to 25.20. If the current price changes by the same percentage, it should be ₹ 26.30 × 25.20/25.00 = ₹ 26.51 (rounded off to ₹ 26.50). Proceeding in this manner, 100 alternate stock price scenarios are generated and changes in portfolio value from the current value are calculated.

Table 4.12 *Simulating Portfolio Values Using Historical Data (Current Value of the Portfolio: ₹ 2.63 m)*

Scenario	Alternate Price	Change from Current (₹2.63 m)
1	26.50	0.020
2	26.00	–0.030
3	26.20	–0.010
…	…	…
99	26.45	0.015
100	26.35	0.050

The last step consists of constructing the histogram of the portfolio returns based on the last 100 days of history, or equivalently sorting the changes in portfolio values to identify the first percentile of the distribution as shown in the Table 4.13:

Table 4.13 *Identifying the First Percentile of the Historical Distribution of the Portfolio Return*

Rank	Change from Current Value
100	– 0.10
99	– 0.08
98	– 0.05
…	…
2	+ 0.07
1	+ 0.08

Using Table 4.13, we identify the first percentile as 0.10. In this example, the estimate of VaR will be updated everyday using the data of the most recent 100 days.

The major attraction of historical simulation is that the method is completely non-parametric and does not depend on any assumption about the distribution of risk factors. We do not need to assume that the returns of the risk factors are normally distributed and independent over time. There is also no need to estimate volatilities and correlations as these are already reflected in the data set. Historical simulation also has no problem accommodating fat tails, since the historical returns already reflect actual synchronous moves in the market across all risk factors. It also allows the calculation of confidence intervals for VaR.

The main drawback of historical simulation is its complete dependence on a particular set of data and thus on the idiosyncrasies of this data set. The underlying assumption is that the past, as captured in the historical data set, is a reliable representation of the future. However, the historical period may cover events such as a market crash or conversely, a period of exceptionally low price volatility that are unlikely to be repeated in the future. Similarly, there may have been structural changes in the market such as the introduction of the Euro at the beginning of 1999. Another practical limitation of historical simulation is data availability. Employing small samples of historical data leaves gaps in the distribution of the risk factors and tends to underrepresent the occurrence of unlikely but extreme events. It cannot be used to conduct sensitivity analysis and is not always computationally efficient when the portfolio contains complex securities.

2. Monte Carlo Approach

Monte Carlo Simulation is similar to historical simulations except that we look forward using simulated values rather than historical values. The Monte Carlo methodology can be implemented by choosing any analytic multivariate distribution for the risk factors. The only limitation is the ability to estimate the parameters of the distribution, such as the mean, variance and the covariance. The Monte Carlo methodology is flexible and allows the analyst to choose distributions that exhibit fat tails and skewness (for instance, Student-t distribution). Processes with mean reversion, complex distributions such as mixtures of normal distributions or the mixture of a normal and a jump process, can also be dealt with easily.

Monte Carlo simulation consists of repeatedly simulating the random processes that govern market prices and rates. Each simulation (scenario) generates a possible value for the portfolio at the target horizon (for instance, 10 days). If we generate enough of these scenarios, the simulated distribution of the portfolio's values will converge towards the true, although unknown, distribution. The VaR can be easily inferred from the distribution. Monte Carlo simulation involves the following three steps:

(*i*) ***Specify All the Relevant Risk Factor*s:** As a first step, we have to select all the relevant risk factors. In addition, we have to specify the dynamics of these factors i.e., their stochastic processes and we need to estimate the parameters (volatilities, correlations, mean reversion factors for interest rate processes and so on.)

(*ii*) ***Construct Price Paths***: Monte Carlo simulations make use of stochastic processes as returns follow such processes. Stochastic processes specify the distribution of asset returns or of market parameters for each time interval, as a function of their expected values and volatilities. Implementing these stochastic processes allows us to model future outcomes and to find all future asset payoffs. Different stochastic processes apply to stock prices and interest rates. This process allows stock prices to drift away from initial values without imposing any bound on prices. However, the time paths of interest rates follow specific processes, because they tend to revert to a long-term average. Once the stochastic processes of risk drivers of value are specified, the technique used includes the following steps:

The entire time path of asset returns from now to the horizon is divided into small intervals by choosing intermediate time points. Random time paths are generated for asset returns and market parameter returns from now to each time point. The small interval returns are cumulated to find cumulative returns. These cumulative returns are used to revalue the asset at the horizon using the relationship between the final value of the asset and the return $V_H = V_0 (1 + r)$. As returns follow a normal distribution, calculation of a single return from now to the horizon allows the return to hit values lower than -100 per cent, resulting in non-acceptable negative values of the asset. For a sufficiently short horizon, the change in value is small that makes it more acceptable to use normal distributions. Normal distributions are easier to handle and are, therefore, preferred. A common stochastic process, used for stock prices is the Geometric Brownian Motion. It is defined by the following equation:

$$dV_t = \mu V_t dt + \sigma V_t dz_t$$

This equation states that the small change in value of a stock price dV_t is the sum of a term $\mu V_t dt$ which reflects the expected return μ per unit time, times the initial stock value, and proportional to the time interval dt, plus a random term. The random component of the return is $\sigma V_t dz_t$, proportional to the return volatility σ, the price V_t and a random 'noise' dz_t, which follows a standard normal variable (mean 0 and volatility 1). The process makes the instantaneous return dV_t/V_t, a direct function of its mean μ, and the volatility σ as shown by the following equation:

$$r = dV_t/V_t = \text{m}d_t + \text{s}dz_t$$

For instance, a time path of asset returns over a year can be generated using the process in the following manner:

Suppose the expected annual return is 10 per cent. The year can be divided into ten parts so that the expected return for each part is 1 per cent. Similarly, the annual volatility (say 20%) can be converted into volatility for each time interval ($20\% \times \sqrt{0.1}$). The values of expected returns and volatility are based on historical data. Random values of standard normal dz_t are generated using random number generators. With these inputs, the value of the return for this time interval can be generated using the following equation and cumulative returns along the entire path can be calculated:

$$r = dV_t/V_t = \mu d_t + \sigma dz_t$$

Starting with a price of ₹ 100, if three successive returns are -3 per cent, 1 per cent and 0.7 per cent respectively, the asset price at the end of the first interval will be ₹ 97, at the end of the second interval will be ₹ 97.97 and the end of the third interval will be ₹ 98.66. The change in the asset value after three intervals is (98.66–100) or -1.34. This process is repeated till the horizon.

The final asset value distribution at the end of the year (horizon) results from the cumulative returns along the time path. This is only one time path for the final asset value. A large number of such time paths (say 1,000) can be simulated. Using these 1,000 final asset values, a probability distribution of changes in the asset value can be constructed and the VaR at the desired confidence level calculated.

(iii) ***Value the Portfolio for Each Path (Scenario)***: Each path generates a set of values for the risk factors that are used as inputs into the pricing models, for each security composing the portfolio. This process is repeated many times, say 10,000 times, to generate the distribution at the risk horizon of the portfolio return. Then, VaR at the 99 per cent confidence level is simply derived as the distance to the mean of the first percentile of the distribution.

Modelling Correlations under Monte-Carlo Methods

In the case of a portfolio of assets, the returns of different assets are correlated since the different risk factors affecting returns on assets are also correlated. For instance, the risk of a portfolio comprising equity, bonds and options is related to two market factors—stock index returns and interest rates. These two factors are negatively correlated. It is, therefore, essential to consider correlation between asset returns while simulating.

When Monte Carlo simulation technique is used, the issue is to correlate random time paths of returns for different assets. The correlation is applied to the random innovation term dz_t. Uncorrelated random innovations are first generated. The uncorrelated random innovations are then transformed into correlated innovations using a technique known as Cholesky decomposition. These innovations are used as inputs in thefollowing equation to generate simulated asset returns:

$$r = dV_t/V_t = \mu d_t + \sigma dz_t$$

Advantages and limitations of Monte Carlo simulations

Monte Carlo is a powerful and flexible approach to VaR. It can accommodate any distribution of risk factors to allow for 'fait tail' distributions, where extreme events are expected to occur more commonly than in normal distributions, and 'jumps' or discontinuities in price processes. It can also accommodate complex portfolios that contain exotic options (such as Asian options and barrier options).

Monte Carlo, like historical simulation, allows the analyst to calculate the confidence interval of VaR, i.e., the range of likely values that VaR can take if we had to repeat the Monte Carlo simulation many times. The narrower this confidence interval, the more precise is the estimate of VaR. It is also easy to carry out sensitivity analysis under Monte Carlo by changing the market parameters used in analysis, such as the term structure of interest rates.

The major limitation of Monte Carlo simulation is the amount of computer resources it usually requires. Some reduction techniques are available for reducing the computational time. Still, Monte Carlo simulation remains very computer intensive and cannot be used to calculate the VaRs of very large and complex portfolios. At most, it can be used for portfolios of a limited size.

Uses and Limitations of VaR

The following are the uses of VaR:

- VaR is a summary measure. It provides an integrated measure of risk that takes into account all possible sources of risk. VaR provides a single number that can be used as a measure of capital requirement. It can also be used to measure risk-adjusted performance and to reward employees on the basis of the risk-adjusted return on capital generated by their activities.
- VaR takes into account the correlations between various risk factors. Depending on whether one risk increases or offsets another risk, VaR produces a higher or lower risk estimate. It, therefore, allows a firm to assess the benefits from portfolio diversification within a line of activity, across businesses (such as equity and fixed-income businesses).
- VaR helps in monitoring the risk taken by different business lines as their risk-limits can be set in terms of VaR. Therefore, employees can be restrained from taking more than the allowed risk.
- VaR is aggregative. Risk limits expressed in VaR units can easily be aggregated at the trading desk level, for each business line and for the entire firm.
- VaR measure is easily understood by senior management, the board of directors and regulators. They can decide whether they feel comfortable or not with the level of risk expressed in terms of VaR.
- New investment proposals can be evaluated using VaR. It helps in assessing the expected risk-adjusted return on capital of new investments and projects.
- VaR is the basis for calculating regulatory capital.
- The rating agencies take VaR calculations into account in establishing their rating of banks.

The following are the limitations of VaR:

- VaR is very useful for estimating the overall risk of an institution, but it does not describe the losses in the left tail. It only indicates the probability of such a value occurring. It also does not indicate which of the component risks contribute most to the total risk. For this purpose, other measures such as 'EVaR', 'Incremental VaR' and 'Delta Var' are used.

- The VaR model also assumes that the market conditions remain normal and do not account for extreme events that lie outside normal market conditions. Extreme conditions are characterized by large price changes, high volatility and breakdown in the correlations among risk factors. Stress testing and scenario analysis techniques are used to model extreme events.
- Another problem with VaR is that it is calculated within a static framework. It does not incorporate the liquidity risk and is therefore appropriate only for short time horizons. Further, VaR helps risk managers describe risk in terms of a loss level that might be breached on, say, one day in every 100 trading days, but it does not tell the manager anything about the magnitude of the potential loss in the tail of the distribution. For this purpose an alternative measure, extreme value at risk (EVaR), is used.

Extensions of VaR

The following are the extensions of VaR:

1. EVaR (Expected Shortfall)

VaR does not provide any indication of by how much any actual losses will exceed the VaR figure. To give an indication of the magnitude of the potential losses in the tail, we complement VaR by extreme value at risk, EVaR, i.e., the expected loss in the first quantile tail, provided that the loss exceeds VaR. For instance, if the VaR at 99 per cent confidence level is ₹ 5 million and there could be two losses in excess of VaR amounting to ₹ 7 million and ₹ 13 million, the EVaR is ₹ 10 million [(7+13)/2].

2. Incremental VaR

Incremental VaR or IVaR, measures the incremental impact on the overall VaR of the portfolio of adding or eliminating an asset, A, or a position. Thus, the formula to calculate IVaR is as follows:

IVaR(A) = VaR(portfolio with asset A) – VaR(portfolio without asset A)

IVaR can be positive if the asset is positively correlated with the rest of the portfolio and thus adds to the overall risk. It can also be negative, if the asset is negatively correlated with the rest of the portfolio.

3. Delta VaR or DVaR

DVaR measures the risk contribution of an asset to the overall risk of the portfolio. The sum of the risk contributions is equal to the overall risk of the portfolio. DVaR depends on the composition of the portfolio and the correlation structure of the assets.

Consider a portfolio P, composed of N assets, $i = 1,\ldots\ldots N$; p_i and A_i are, respectively, the unit price and the number of units of asset i in the portfolio. The portfolio value is as follows:

$$P(A_1, A_2, \ldots.. A_N) = \sum_{i=1}^{N} p_i A_i$$

The DVaR property is as follows:

$$VaR_p = \sum_i DVaR_i$$

and

$$DVaR_i = \frac{\partial VaR_p}{\partial A_i} A_i$$

NOTES

The total *VaR* of the portfolio is the sum of the risk contributions, $DvaR_i$, of all the assets contained in the portfolio. This is a useful decomposition since it highlights the most significant risks, i.e., the positions to which the portfolio is most sensitive. $DvaR_i$ is the product of the marginal change in risk per unit change in asset i, and the position size itself. $DvaR_i$ represents the risk contribution of asset i to the portfolio. Decomposition of the overall risk into its component parts helps identify the most significant risks and the hedge portfolios that can optimally reduce these risks, given hedging costs.

Check Your Progress

7. State about the milestones.
8. Define the term software quality.
9. How is the VaR calculated under the portfolio normal approach?
10. What is a fat-tailed distribution?
11. How is VaR calculated as per the asset normal approach?
12. What is the main aim of modeling portfolio risk?

4.10 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Project management is the art of directing and coordinating the human, and material resources throughout the project by using modern management techniques.
2. The main purpose of project management is to achieve the predetermined objectives of scope, cost, time, quality and the satisfaction of the participant.
3. Project planning is an organized and integrated management process that focuses on activities required for successful completion of the software project.
4. The configuration management plan defines the process, which is used for making changes to the project scope. Generally, the configuration management plan is concerned with redefining the existing objectives of the project and deliverables (software products that are delivered to the user after completion of a software development phase).
5. In the early 1980s, Barry Boehm developed a model called COnstructive COst MOdel (COCOMO) to estimate the total effort required to develop a software project. The COCOMO model is commonly used as it is based on the study of already developed software projects.
6. Software metrics help project managers to gain an insight into the efficiency of a software process, project and product. This is possible by collecting

quality and productivity data and then analysing and comparing these data with past averages in order to know whether quality improvements have occurred. Also, when metrics are applied in a consistent manner, it helps in project planning and project management activity.

7. Milestones are formal representations of the progress of a project. Generally, milestones are planned when deliverables are provided. Milestones describe the end-point at which the software process activity is completed. After completion of a milestone, its output is described in the form of a document. This document comprises information about the completion of a phase of the project.
8. Quality planning is a structured process for defining the procedures and methods that are used to develop software. Quality planning starts in the early phases of software development. The software quality assurance (SQA) plan defines the activities for the SQA group.
9. The portfolio normal approach calculates VaR as a multiple of the standard deviation of the portfolio returns.
10. A fat-tailed distribution is characterized as one that has more observations far away from the mean than is the case in a normal distribution.
11. According to the asset normal approach, VaR is calculated as a multiple of the standard deviation of the portfolio returns.
12. The goal of modelling portfolio risk is to obtain the distribution of portfolio returns at the horizon.

4.11 SUMMARY

- The main responsibility of a Software Project Manager is to plan and schedule software project development tasks.
- A software project manager is concerned about whether the product meets the required standards and be finally available for use after meeting the time and financial constraints.
- Project management is the art of directing and coordinating the human, and material resources throughout the project by using modern management techniques.
- The main purpose of project management is to achieve the predetermined objectives of scope, cost, time, quality and the satisfaction of the participant.
- Project planning is an organized and integrated management process that focuses on activities required for successful completion of the software project.
- The purpose of the project is to accomplish project objectives and business objectives. Project objectives include accomplishment of user requirements in software. In addition, the software should be completed according to schedule, within budget and incorporate quality in software. Business objectives include evaluating processes, renewing policies and processes, keeping the project on schedule and improving software.

- Project scope determines the limitations of the project. It includes functions, features, constraints and interfaces of the software.
- Project planning process involves a set of interrelated activities followed in an orderly manner to implement user requirements in software and includes the description of a series of project planning activities and individual(s) responsible for performing these activities.
- Project plan is a document that provides information about the end date, milestones, activities and deliverables required for the project. Various plans included in the project plan are quality assurance plan, verification and validation plan, configuration management plan, maintenance plan and staffing plan.
- In the early 1980s, Barry Boehm developed a model called COnstructive COst MOdel (COCOMO) to estimate the total effort required to develop a software project.
- The constructive cost model is based on the hierarchy of three models, namely basic model, intermediate model and advance model.
- Software metrics help project managers to gain an insight into the efficiency of a software process, project and product. This is possible by collecting quality and productivity data and then analysing and comparing these data with past averages in order to know whether quality improvements have occurred.
- Project scheduling provides details such as start date and end date of the project, milestones and tasks for the project. In addition, it specifies the resources (such as people, equipment and facilities) required to complete the project and the dependencies of tasks of the project on each other.
- Quality planning is a structured process for defining the procedures and methods that are used to develop software. Quality planning starts in the early phases of software development. The software quality assurance (SQA) plan defines the activities for the SQA group.
- Risk analysis is a technique to identify and assess factors that may jeopardize the success of a project or achieving a goal. This technique also helps to define preventive measures to reduce the probability of these factors from occurring and identify countermeasures to successfully deal with these constraints when they develop to avert possible negative effects on the competitiveness of the company.
- Market risk is the potential downside deviation of the market value of transactions and of trading portfolio during the liquidation period. Use of notional amounts and sensitivities to measure market risk are inadequate as they do not provide an overall measure of capital at risk.
- A popular and widely used measure of market risk is the VaR measure. VaR can be measured using both parametric and non-parametric methods. Nonparametric methods should provide more accurate estimates of VaR as these are based on full valuation. However, these are more computation intensive.

- VaR fails to provide a measure of expected loss when it exceeds the threshold at the required confidence level. It also does not indicate the risk contribution of different assets to the overall risk. For this purpose, additional risk measures, such as E-VaR, I-VaR and Delta VaR are needed.
- VaR models developed by banks have also to be subjected to stress tests, scenario analysis and back-tests to establish their robustness in assessing risk.

4.12 KEY TERMS

- **Project management:** It is the art of directing and coordinating the human, and material resources throughout the project by using modern management techniques.
- **Project planning**: It is an organized and integrated management process that focuses on activities required for successful completion of the project.
- **Project scope**: It provides a detailed description of functions, features, constraints and interfaces of the software that are to be considered.
- **Software metrics:** It help project managers to gain an insight into the efficiency of a software process, project and product.
- **PERT chart:** The Program Evaluation and Review Technique (PERT) chart is used to schedule, organize and coordinate tasks within the project.

4.13 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is the objective of project management?
2. Write the characteristics of project management.
3. What are the principles followed for project planning?
4. Define the term software equation.
5. State about the two models of cost estimations.
6. Write the guidelines for software metrics.
7. What is work breakdown structure?
8. What do you understand by tracking the schedule?
9. With the help of diagram, show the calculation of VaR from the probability distribution of the change in portfolio value.

Long-Answer Questions

1. Discuss about the various principles of software project management.
2. Explain the need for project management giving appropriate examples.
3. What are the various principles that are followed to carry out project scheduling appropriately?

4. Explain the various parameters that affect the cost of a software project.
5. Briefly explain the intermediate and advance cost model with the help of appropriate examples.
6. Discuss the difference between measures, metric and indicators.
7. Discuss briefly quality assurance plan with the help of diagram.
8. What are the methods of managing the project risk of an infrastructure project?
9. Explain in detail the methods of mitigating financing risk of an infrastructure projects.
10. Explain the stress testing, scenario analysis and back testing procedures applied to VaR.

4.14 FURTHER READING

Mali, Rajib. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hall of India.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Education.

Jalote, Pankaj. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Sommerville, Ian. *Software Engineering*. New Delhi: Addison Wesley.

Fenton, N.E. and F.L. Pfleeger. *Software Metrics*. Mumbai: Thomson Learning.

UNIT 5 SOFTWARE DESIGN

NOTES

Structure

5.0 INTRODUCTION

Software design is the process by which an agent creates a specification of a software artifact intended to accomplish goals, using a set of primitive components and subject to constraints. Software design may refer to either all the activity involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems. Software design usually involves problem-solving and planning a software solution. This includes both a low-level component and algorithm design and a high-level, architecture design.

The design process comprises a set of principles, concepts and practices, which allow a software engineer to model the system or product that is to be built. This model, known as the design model, is assessed for quality and reviewed before a code is generated and tests are conducted. The design model provides details about software data structures, architecture, interfaces and components, which are required to implement the system.

In this unit you will study about the software design, system design, software design quality, design description language, user interface design, data flow diagram, concurrent system design, object-oriented design, design quality assurance, design reviews, validation and verification.

5.1 OBJECTIVES

NOTES

After going through this unit you will be able to:

- Understand the basics of software design
- Define the system design
- Know about the design decomposition
- Explain the software design quality
- Discuss the design description language
- Elaborate on the user interface design
- Describe the data flow diagram
- Know about the concurrent system design
- Explain the object-oriented design
- Define the design quality assurance
- Understand the design reviews
- Discuss the validation and verification

5.2 SOFTWARE DESIGN

Software design is a software engineering phase in which blueprint is developed which serves as base for constructing the software system. IEEE defines software design as 'both a process of defining the architecture, components, interfaces and other characteristics of a system or component and the result of that process.'

During the software design phase, many critical and strategic decisions are made to meet the required functional and quality requirements of a system. These decisions are taken into account to successfully develop the software and carry out its maintenance in a systematic manner to improve the quality of the end product. Software design serves as a blue print for the solution of software system.

Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice (see Figure 5.1). Some of the commonly followed design principles are the following:

- **Software design should correspond to the analysis model:** Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.

- **Choose the right programming paradigm:** A programming paradigm describes the structure of the software system. Depending on the nature and type of the application, different programming paradigms such as procedure oriented, object-oriented, prototyping paradigms, etc. can be used. The paradigm should be chosen keeping in mind constraints such as time, availability of resources and nature of user's requirements.
- **Software design should be uniform and integrated:** In most cases, rules, format and styles are defined in advance to the design team before the starting of design process. The design is said to be uniform and integrated if the interfaces are properly defined among design components.
- **Software design should be structured to adapt change:** The fundamental design concepts (abstraction, refinement, modularity) should be applied to achieve this principle.
- **Software design should comprise minimal conceptual (semantic) errors:** The design team must ensure that major conceptual elements of design, such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.
- **Software design should be structured to degrade gently:** A software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.
- **Software design should represent closeness between the software and problem existing in the real world:** The design structure should be such that it always relates with the real-world problem.
- **Code reuse:** There is a common saying among software engineers,'do not reinvent the wheel'. Therefore, existing design hierarchies should be effectively reused to increase productivity.
- **Designing for testability:** A common practice that has been followed is to keep the testing phase apart from the design and implementation phases. That is, first the software is developed (designed and implemented) and then handed over to the testers who subsequently determine whether or not the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as if these types of errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should involve with analysts during the testing phase to prepare tests for determining whether the user requirements are being met or not.

NOTES

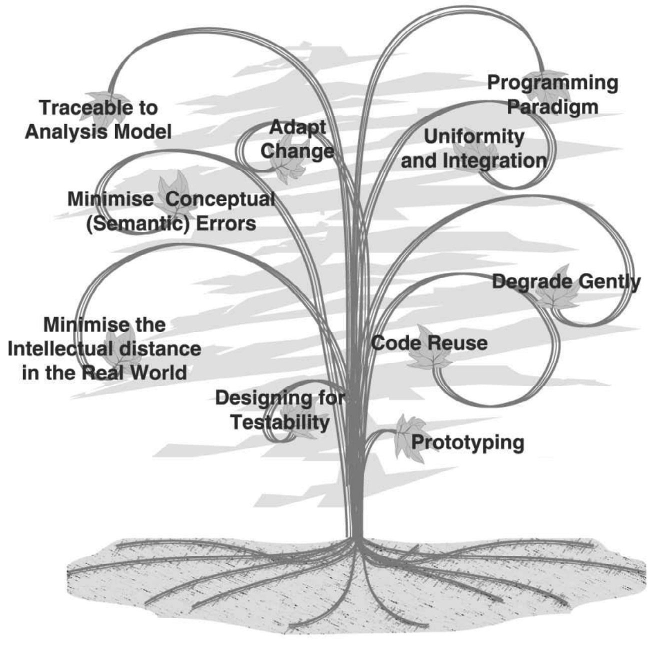

Traceable to Analysis Model
Adapt Change
Programming Paradigm
Uniformity and Integration
Minimise Conceptual (Semantic) Errors
Degrade Gently
Minimise the Intellectual distance in the Real World
Code Reuse
Designing for Testability
Prototyping

Fig. 5.1 *Design Principles of Software*

- **Prototyping:** Prototyping should be used when requirements are not completely defined in the beginning and the user interacts with the developer to expand and refine the requirements as the development proceeds. Using prototyping, a quick'mock-up' of the system can be developed with the currently known requirements to understand the system design. This mock up can be used as a effective means to give the users a feel of what the system will look like and demonstrate functions that will be included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer's requirements.

Design principles are often constrained by the existing hardware configuration, the implementation language, the existing file and data structures and the existing organizational practices. Also, the evolution of each software design should be meticulously designed for future evaluations, references and maintenance.

Application of Fundamental Design Concepts

Every software process is characterized by basic concepts along with certain practices or methods. Methods are the expression of the concepts as they apply to a particular situation. As new technology replaces older technology, many changes occur in the methods that are applied for the development of a software. However, the fundamental concepts underlining the software design process remain the same. The concepts discussed below provide the'underlying basis'for development and evaluation of software design.

Abstraction

Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components. IEEE defines abstraction as 'a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.' The concept of abstraction can be used in two ways, as a process and as an entity. As a process, it refers to a mechanism of hiding irrelevant details and representing only the essential features of an item so that one can focus on important things at a time. As an entity, it refers to a model or view of an item.

Each step in the software process is accomplished through various levels of abstraction. At the highest level of abstraction, a general solution of the problem is presented whereas at lower levels of abstraction, a detailed description about the solution is presented. For example, during system engineering, a software is viewed as an element of computer based engineering, while in the requirement analysis, the same software is viewed as a solution to a problem domain and as we move through the design phase, the level of abstraction is reduced to the source code generation.

There are three commonly used abstraction mechanisms in software design, namely, functional abstraction, data abstraction and control abstraction. All these mechanisms allow us to control the complexity of the design process by proceeding from the abstract design model to concrete design model in a systematic manner.

- **Functional abstraction:** This involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as'groups'. Within these groups there exist routines, which may be visible or hidden. Visible routines can be used within the containing groups as well as within other groups, whereas hidden routines are hidden from other groups and can be used within the containing group only.
- **Data abstraction:** This involves specifying data that describes a data object. For example, the data object window encompasses a set of attributes (window type, window dimension) that describe the window object clearly. In this abstraction mechanism, representation and manipulation details are ignored.
- **Control abstraction:** This means to express the final outcome without concerning the implementation details. For example, if and while statements in programming languages (like C and C++) are abstractions of machine code implementations, which involve conditional instructions. In architectural design level, this abstraction mechanism permits specifications of sequential subprogram and exception handlers without the concern for exact details of implementation.

Architecture

Software architecture refers to the structure of the system, which is composed of the various components of a program/system, the attributes (properties) of those components and the relationship amongst them. The software architecture enables the software engineers to analyze the software design efficiently. In addition, it also helps them in decision-making, and handling risks. The software architecture:

- provides an insight to all the interested stakeholders that enable them to communicate with each other;

- highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase;
- creates intellectual models of how the system is organized into components and how these components interact with each other.

Currently, software architecture is represented in an informal and unplanned manner. To support a particular architecture style, architectural concepts are often represented in the infrastructure and the lack of an explicit independent characterization of an architecture in the initial stages of a system configuration, limits the advantages of this design concept in the present scenario. Software architecture comprises two elements of design model: data design and architectural design.

Patterns

A pattern provides a description of the solution to a recurring design problem in such a way that the solution can be used again and again. Thus, each pattern represents a reusable solution to a recurring problem. The term pattern has been adopted in software from the work of the architect Christopher Alexander who explored patterns in architecture.

Types of Design Patterns

Patterns are used once the analysis model is developed. Patterns reflect low-level **strategies** for design of components in the system and high-level strategies, which impact the design of the overall system. Patterns are divided into the following three categories on the basis of their corresponding levels of abstraction:

- **Architectural patterns:** These patterns are high-level strategies that are concerned with the overall structure and organization of a software system. That is, they define the elements of a software system such as subsystems, components, classes, etc. In addition, they also indicate the relationship between elements along with the rules and guidelines for specifying these relationships. Architectural patterns are often considered equivalent to software architecture.
- **Design patterns:** These patterns are medium-level strategies that are used to solve design problems. They provide a means for the refinement of the elements (as defined by architectural pattern) of a software system, or the relationship among them. It addresses a specific element of the design such as relationship among components or mechanisms that affects component-to-component interaction. Design patterns often equate with *software* components with emphasis on reusability.
- **Idioms:** These patterns are low-level patterns, which Zare paradigm-specific and programming language-specific. They describe how different elements of the software system are implemented in a specific programming language. Software components are binary units of independent production, acquisition and deployment that interact to form a functioning program.

Modularity

Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as modules. A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules (see Figure 5.2). After developing the modules, they are integrated together to meet the software requirements. More the number of modules a system is divided into, greater will be the effort required to integrate the modules.

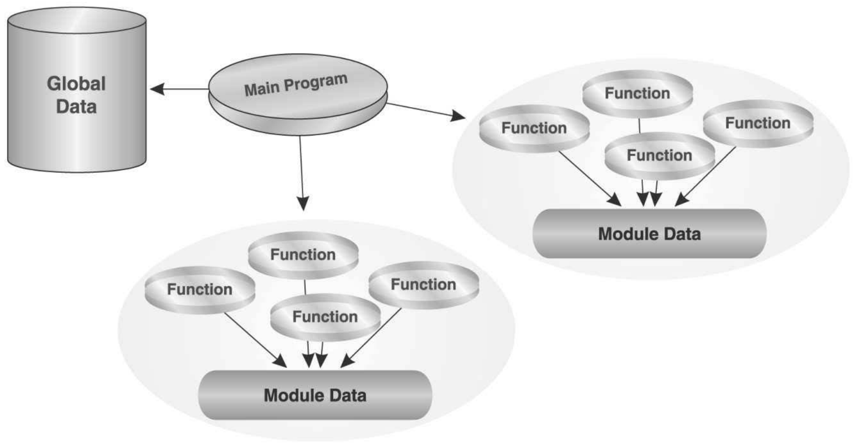


Fig. 5.2 *Modules in Software Programs*

Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conduct maintenance work without adversely affecting the functioning of the software.

Information Hiding

Modules should be specified and designed in such a way that the data structures and processing details of one module are not accessible to other modules (see Figure 5.3). They pass only that much information to each other, which is required to accomplish the software function. The way of hiding unnecessary details is referred to as information hiding. IEEE defines information hiding as'the technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus each module is a'black box' to the other modules in the system.' Information hiding is of immense use when modifications are required during the testing and maintenance phase.

Some of the advantages associated with information hiding are as follows:

- It leads to low coupling.
- It emphasizes communication through controlled interfaces.
- It decreases the probability of adverse effects.
- It restricts the affects of changes in one component on others.
- It results in higher quality software.

Stepwise Refinement

Stepwise refinement is a top-down design strategy used for decomposing a system from a high level of abstraction into a more detailed level (lower level) of abstraction. At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data. As we proceed towards the lower levels of abstraction, more and more details are available.

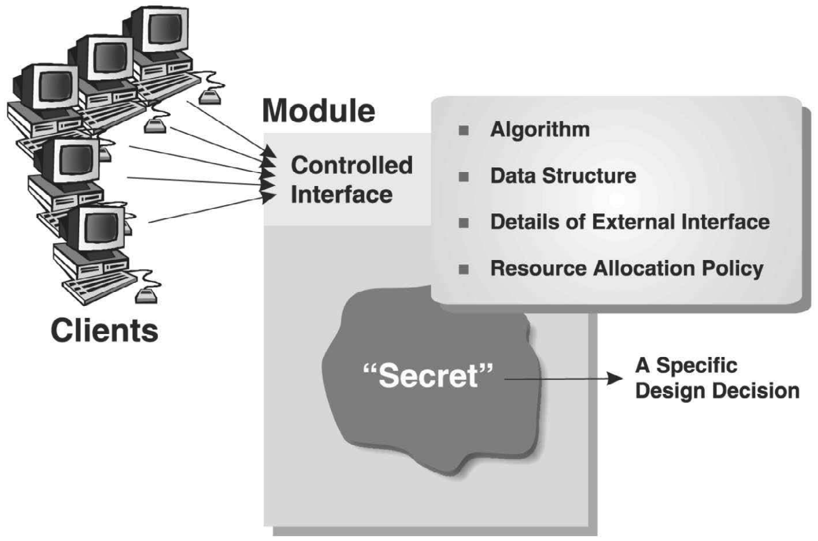

Module
Controlled Interface
Algorithm
Data Structure
Details of External Interface
Resource Allocation Policy
Clients
"Secret"
A Specific Design Decision

Fig. 5.3 *Information Hiding*

Software designers start the stepwise refinement process by creating a sequence of compositions for the system being designed. Each composition is more detailed than the previous one and contains more components and interactions. The earlier compositions represent the significant interactions within the system, while the later compositions show in detail how these interactions are achieved.

To have a clear understanding of the concept, let us consider an example of stepwise refinement. Every computer program comprises input, process and output:

- INPUT
  - o Get user's name (string) through a prompt
  - o Get user's grade (integer from 0 to 100) through a prompt and validate
- PROCESS
- OUTPUT

This is the first step in refinement. The input phase can be refined further as follows:

- INPUT
  - o Get user's name through a prompt
  - o Get user's grade through a prompt
  - o While (invalid grade)
  - o Ask again
- PROCESS
- OUTPUT

Note: Stepwise refinement can also be performed for PROCESS and OUTPUT phase.

Refactoring

Refactoring is an important design activity that simplifies the design of a module without changing its behaviour or function. Refactoring can be defined as a process of modifying a software system to improve the internal structure of the design without changing its external behaviour. During the refactoring process, the existing design is checked for any type of flaws like redundancy, poorly constructed algorithms and data structures, etc. in order to improve the design. For example, a design model might yield a component, which exhibits low cohesion (like a component performs only four functions that have a limited relationship with one another). Software designers may decide to refactor the component into four different components, each exhibiting high cohesion. This leads to easier testing and maintenance of the component.

Structural Partitioning

When the architectural style of a design follows a hierarchical nature, the structure of the program can be partitioned either horizontally or vertically. In **horizontal** partitioning, the control modules (as depicted in the shaded boxes in Figure 5.4 (a)) are used to communicate between functions and execute the functions. Structural partitioning provides the following benefits:

- The testing and maintenance of the software becomes easier
- The propagation of adverse affects becomes less
- The software can be extended easily

Besides these advantages, there is also some disadvantage of horizontal partitioning. It requires to pass more data across the module interface, which. makes the control-flow of the problem more complex. This usually happens in cases where data moves rapidly from one function to another.

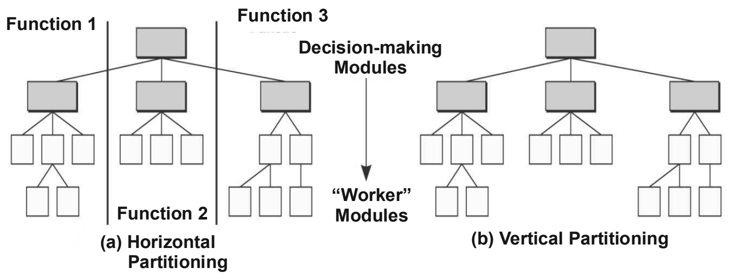


Fig. 5.4 *Horizontal and Vertical Partitioning*

In vertical partitioning, the functionality is distributed among the modules in a top-down manner. The modules at the top level called control modules perform the decision-making and do little processing whereas the modules at the low level perform all input, computation and output tasks.

Developing a Design Model

To develop a complete specification of design (design model), four elements are used (see Figure 5.5). These are:

1. **Data design:** This creates data structure by converting data objects specified during the analysis phase. The data objects, attributes and relationships defined in entity relationship diagrams provide the basis for

data design activity. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design elements.

2. **Architectural design:** This specifies the relationship between structural elements of a software, design patterns, architectural styles and the factors affecting the way in which architecture can be implemented.
3. **Procedural design:** This converts the structural elements of software architecture into a procedural description of software components.
4. **Interface design:** This depicts how a software communicates with the system that interoperates with it and with the end-users.

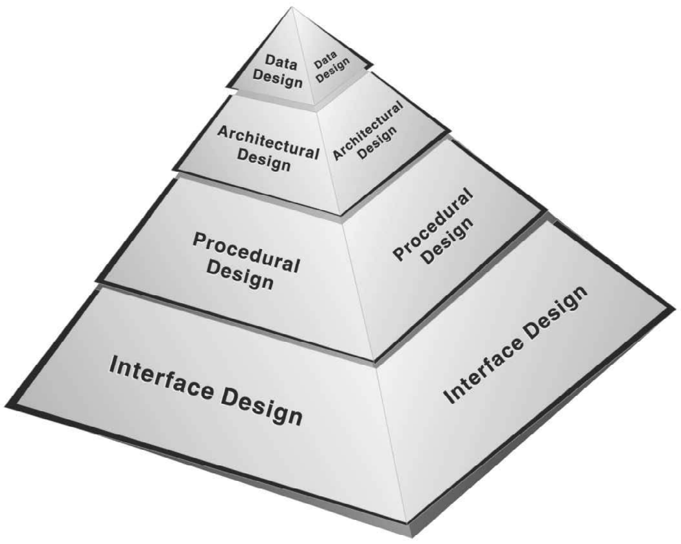


Fig. 5.5 *Design Model and Its Elements*

5.3 SYSTEMS DESIGN

Architectural design (also called system design) acts as a preliminary'blueprint' from which a software can be developed. After completion of the first iteration of architectural design, procedural design takes place.

Architectural Design

Requirements of the software should be transformed into an architecture that describes software's top-level structure and identifies its components. This is accomplished through architectural design. IEEE defines architectural design as 'the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system'. This framework is established by examining the software requirement document and designing a model for providing implementation details. These details are used to specify the components of the system along with their inputs, outputs functions, and the interaction between them. An architectural design performs the following functions:

- It defines an abstraction level at which the designers can specify the functional and performance behaviour of the system.

- It acts as a guideline for enhancing the system (whenever required) by describing those features of the system that can be modified easily without affecting the system integrity.
- It evaluates all top-level designs.
- It develops and documents top-level design for the external and internal interfaces.
- It develops preliminary versions of user documentation.
- It defines and documents preliminary test requirements and the schedule for software integration.

Sources of architectural design are as follows:

- Information regarding the application domain for the software to be developed
- Using data-flow diagrams
- Availability of architectural patterns and architectural styles

Architectural design occupies a pivotal position in software engineering. It is during architectural design that crucial requirements such as performance, reliability, costs, etc. are addressed. This task is cumbersome as the software engineering paradigm is shifting from monolithic, stand-alone, built-from-scratch systems to componentized, evolvable, standards-based and product line-oriented systems. Also, a key challenge for designers is to know precisely how to proceed from requirements to architectural design. To avoid these problems, designers adopt strategies such as reusability, componentization, platform-based, standards-based and so on.

Although the architectural design is the responsibility of developers but some other people like user representatives, systems engineers, hardware engineers and operations personnel are also involved. All these stakeholders must also be consulted while reviewing the architectural design in order to minimize the risks and errors.

Architectural Design Representation

Architectural design can be represented using various models, such as:

- **Structural model:** Illustrates architecture as an ordered collection of programs components.
- **Framework model:** Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.
- **Dynamic model:** Apecifies the behavioural aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment.
- **Process model:** Focuses on the design of the business or technical process, which must be implemented in the system.
- **Functional model:** Represents the functional hierarchy of a system.

Architectural Design Output

The output of the architectural design process is an architectural design document (ADD), which consists of a number of graphical representations that consist of software models along with associated descriptive text. These models include static model, an interface model, a relationship model and a dynamic process model that shows how the system is organized into a process at run-time. This document gives the developers' a solution to the problem stated in the software requirements specification (SRS). It considers only those requirements in detail that affect the program structure. In addition to ADD, other outputs of the architectural design are as follows:

- Progress reports, configuration status accounts and audit reports
- Various plans for detailed design phase, which include:
  - o a software project management plan
  - o a software configuration management plan
  - o a software verification and validation plan
  - o a software quality assurance plan

Architectural Styles

Architectural styles define a group of interlinked systems that share structural and semantic properties. In short, the objective of using architectural styles is to establish a structure for all the components present in a system. If an existing architecture is to be re-engineered, then imposition of an architectural style results in fundamental changes in the structure of the system. This change also includes re-assignment of the functionality performed by the components.

By applying certain constraints on the design space, we can make different style-specific analyses from an architectural style. In addition, if conventional structures are used for an architectural style, the other stakeholders can easily understand the organization of the system.

A computer-based system (a software is part of this system) exhibits one of the many available architectural styles. Each style describes a system category that includes:

- Computational components (such as clients, server, filter, database) that provide a specific part of the services of the system.
- A set of connectors (such as remote procedure call, internet protocols, pipes, etc.) that provide a means for interaction between these components.
- Constraints that define integration of components to form a system.
- A semantic model, which enables the software designer to identify the characteristics of the system as a whole by studying the characteristics of its components.

Some of the commonly used architectural styles are data-flow architecture, object-oriented architecture, layered system architecture, data-centered architecture, and call and return architecture. The use of an appropriate architectural style promotes design reuse, leads to code reuse and supports interoperability.

Data-Flow Architecture

Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations. Each component known as filter, transforms the data and sends this transformed data to other filters for further processing using the connector known as pipe, (see Figure 5.6(a)). Each filter works as an independent entity, that is, it is not concerned with the filter which is producing or consuming the data. A pipe is a unidirectional channel which transports the data received on one end to the other end. It does not change the data in any way and it merely supplies the data to the filter on the receiver end.

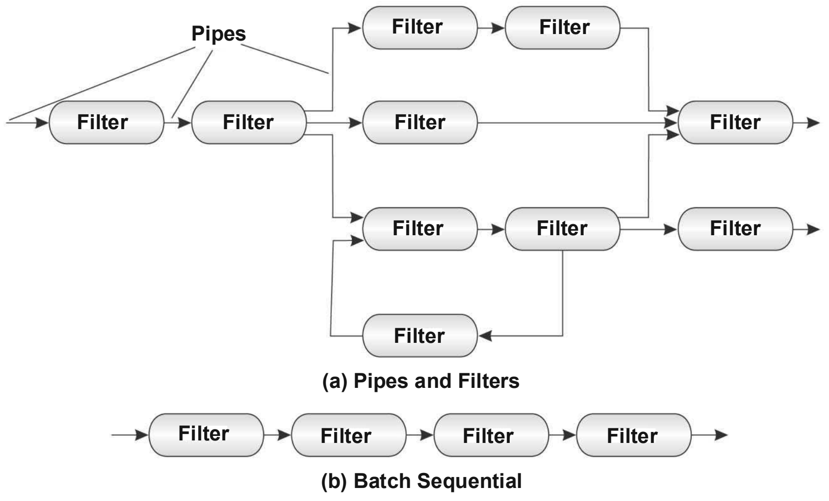


Fig. 5.6 *Dataflow Architecture*

Most of the times, the data-flow architecture degenerates to a batch sequential system. In this system, a batch of data is accepted as input and then a series of sequential filters are applied to transform this data. One common example of this architecture is Unix shell programs. In these programs, Unix processes act as filters and the file system through which Unix processes interact act as pipes. Other well-known examples of this architecture are compilers, signal-processing systems, parallel programming, functional programming and distributed systems. Some advantages associated with the data-flow architecture have been as follows:

- It supports reusability.
- It is maintainable and modifiable.
- It supports concurrent execution.

Some disadvantages associated with the data-flow architecture are:

- It often degenerates to batch sequential system.
- It does not provide enough support for applications requiring user interaction.
- It is difficult to synchronize two different but related streams.

Object-Oriented Architecture

In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data. In this style, components are represented as objects and they interact with each other through methods (connectors). This architectural style has two important characteristics:

- Objects maintain the integrity of the system.
- An object is not aware of the representation of other objects.

Some of the advantages associated with the object-oriented architecture are:

- As implementation detail of objects is hidden from each other, they can be changed without affecting other objects.
- It allows designers to decompose a problem into a collection of independent objects.

Layered Architecture

In layered architecture, several layers (components) are defined with each layer performing a well-defined set of operations. These layers are arranged in a hierarchical manner, each one built upon the one below it (see Figure 5.7). Each layer provides a set of services to the layer above it and acts as a client to the layer below it. The interaction between layers is provided through protocols (connectors) that define a set of rules to be followed during interaction. One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organization for Standardization) communication system (see Figure 5.7).

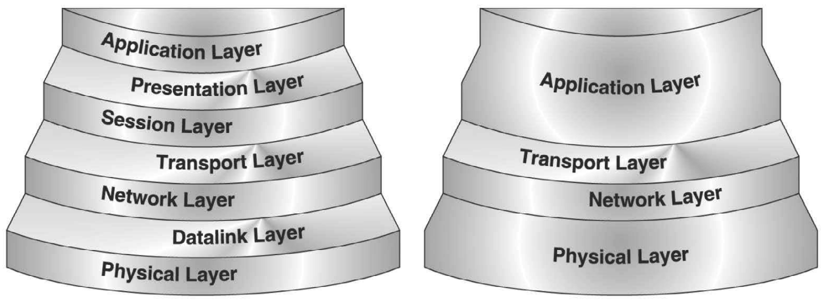


Fig. 5.7 *OSI and Internet Protocol Suite*

Data-Centred Architecture

A data-centred architecture has two distinct components: a central data structure or data store (central repository) and a collection of client software. The data store (e.g. a database or a file) represents the current state of data and the client software perform several operations like add, delete, update, etc. on the data stored in data store (see Figure 5.8). In some cases, the data store allows the client software to access the data independent of any changes or the actions of other client software. However, in a variation of this style the data store is transformed into a blackboard that notifies the client software when the data (of their interest) changes. In addition, the information can be transferred among clients through the blackboard component.

In this architectural style, existing components can be deleted and new clients can be added to the architecture without affecting the overall architecture. This is because client components operate independently.

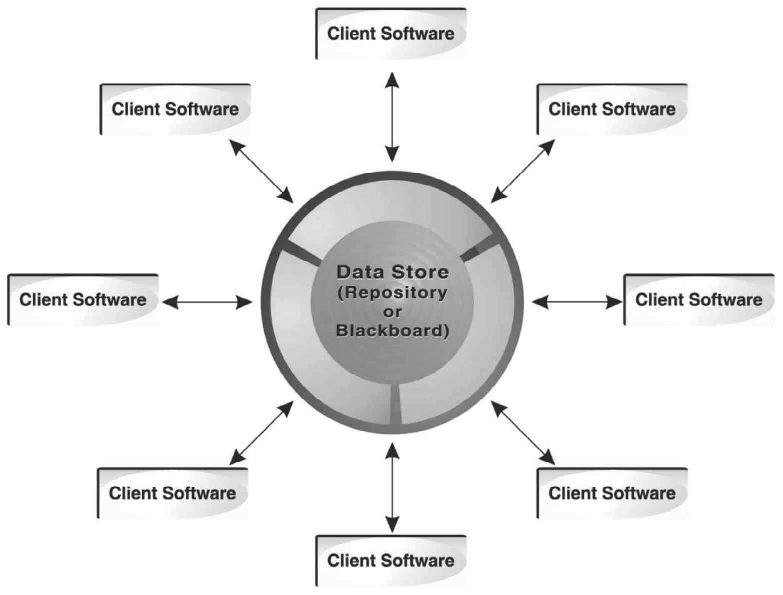


Fig. 5.8 *Data-Centred Architecture*

Some of the advantages of a data-centered system are as follows:

- Clients are relatively independent of each other.
- Data store is independent of the clients.
- It adds scalability (i.e., new clients can be added easily).
- It supports modifiability.
- It achieves data integration in component-based development using blackboard.

Call and Return Architecture

A call and return architecture enables software designers to achieve a program structure, which can be easily modified. This style consists of the following two substyles:

- **Main program/subprogram architecture:** In this, function is decomposed into a control hierarchy where the main program invokes a number of program components, which in turn may invoke other components (see Figure 5.9).

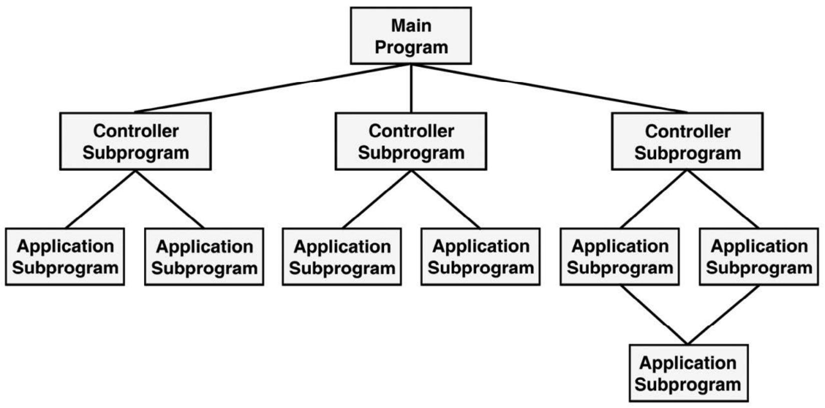


Fig. 5.9 *Main Program/Subprogram Architecture*

- **Remote procedure call architecture:** In this, components of main or subprogram architecture are distributed over a network across multiple computers.

Procedural Design

As soon as the first iteration of architectural design is complete, procedural design takes place. Procedural design is created by transforming the structural elements defined by the software architecture into procedural descriptions of software components. These components are derived from the analysis model where data-flow-oriented element (present in the analysis model) serves as the base for the derivation. A component also known as module, resides within the software architecture and serves one of the following three roles:

- A control component, which coordinates the invocation of all other components present in the problem domain.
- A problem domain component, which implements a complete or partial function as required by the user.
- An infrastructure component supports functions, which in turn support the processing required in the problem domain.

Procedural design is used to define the data structures, algorithms, interface description and communication mechanisms allocated to each module. A module or component can be defined as a modular building block for the software. However, the meaning of component differs according to how software engineers use it. The modular design of the software should exhibit the following sets of properties:

- **Provide simple interface:** Simple interfaces reduce the number of interactions that must be considered when verifying that a system performs its intended function. Simple interfaces also make it easier to reuse components in different circumstances. Reuse is a major cost saver. Not only does it reduce the time spent in coding, designing and testing but also allows development costs to be amortized over many projects. Numerous studies have shown that reusing software design is by far the most effective technique for reducing software development costs.
- **Ensure information hiding:** The benefits of modularity automatically do not follow the act of subdividing a program. Each module should encapsulate information that is not available to the rest of a program. This reduces the cost of subsequent design changes. For example, a module may encapsulate related functions which can benefit from a common implementation or which are used in many parts of a system.

Modularity has become an accepted approach in every engineering discipline. With the introduction of modular design, complexity of software design has considerably reduced; change in the program is facilitated that has encouraged parallel development of systems. To achieve effective modularity, design concepts like functional independence are considered to be very important.

Functional Independence

Functional independence is the refined form of the design concepts of modularity, abstraction and information hiding. Functional independence is achieved by developing a module in such a way that it uniquely performs given sets of function without interacting with other parts of the system. A software that uses the property of functional independence is easier to develop because its functions can be categorized in a systematic manner. Moreover, independent modules require less maintenance and testing activity as secondary effect caused by design modification are limited with less propagation of errors. In short, it can be said that functional independence is a key to a good software design and a good design results in high-quality software.

There exist two qualitative criteria for measuring functional independence, namely, coupling and cohesion. Coupling is a measure of relative interconnection among modules whereas cohesion measures the relative functional strength of a module (see Figure 5.10).

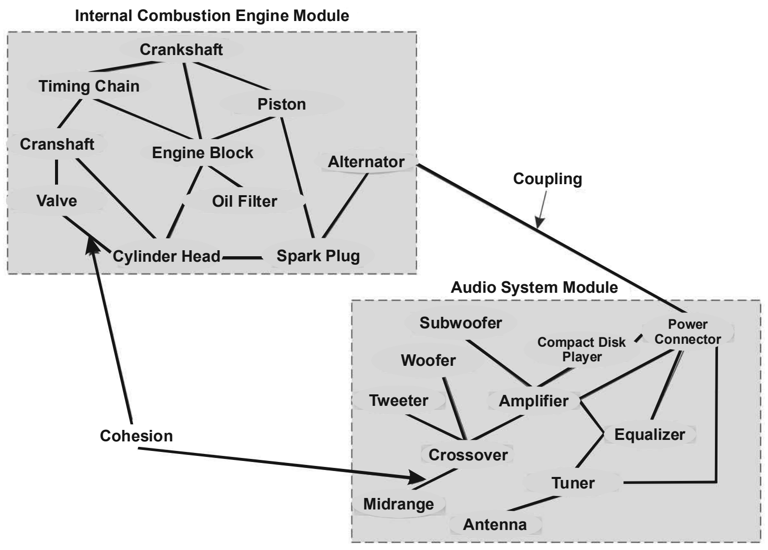


Fig. 5.10 *Coupling and Cohesion*

Coupling

Coupling is the measure of interdependence between one module and another. It depends on the interface complexity between components, the point at which an entry or reference is made to a module and the kind of data that passes across an interface. For better interface and well-structured system, modules should have low coupling which minimizes the 'ripple effect' where changes in one module cause errors in other modules. Module coupling is categorized into the following types.

- **No direct coupling:** Two modules are 'no direct coupled' when they are independent of each other. In Figure 5.11, Module 1 and Module 2 are 'not directly coupled'.
- **Data coupling:** Two modules are 'data coupled' if they communicate by passing parameters. In Figure 5.11, Module 1 and Module 3 are data coupled.

NOTES

- **Stamp coupling:** Two modules are 'stamp coupled' if they communicate through a passed data structure that contains more information than necessary for them to perform their functions. In Figure 5.11, data structure is passed between Modules 1 and 4. Therefore, Module 1 and Module 4 are stamp coupled.

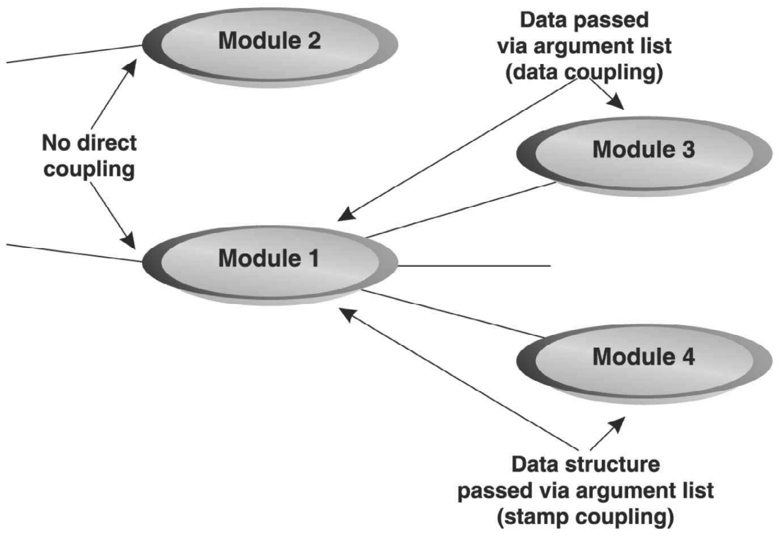

Module 2
Data passed via argument list (data coupling)
No direct coupling
Module 3
Module 1
Module 4
Data structure passed via argument list (stamp coupling)

Fig. 5.11 *No Direct, Data and Stamp Coupling*

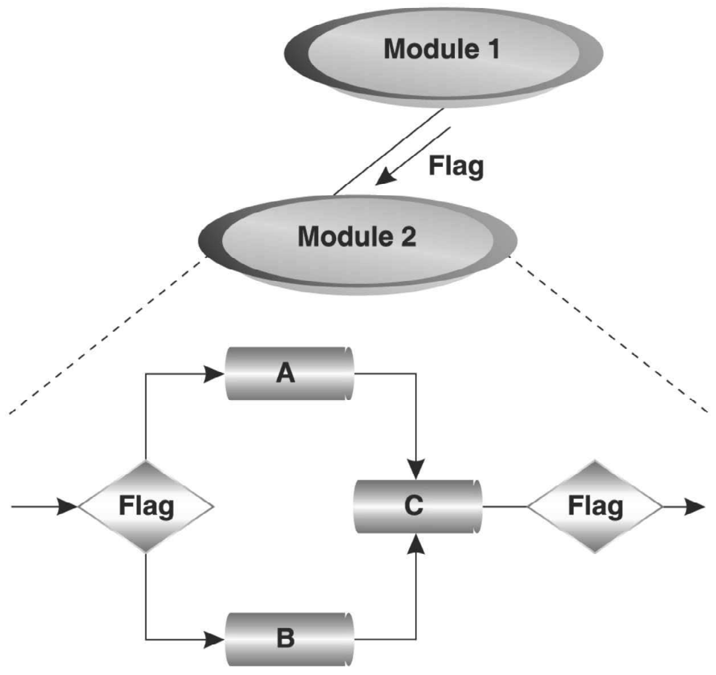


Fig. 5.12 *Control Coupling*

- **Control coupling:** Two modules are 'control coupled' if they communicate (pass a piece of information intended to control the internal logic) using at least one 'control flag'. The control flag is a variable that controls decisions in subordinate or superior modules. In Figure 5.12, when Module 1 passes the control flag to Module 2, Module 1 and Module 2 are said to be control coupled.

- **Content coupling:** Two modules are 'content coupled' if one module changes a statement in another module, one module references or alters data contained inside another module, or one module branches into another module. In Figure 5.13, Modules B and Module D are content coupled.
- **Common coupling:** Two modules are common coupled if they both share the same global data area. In Figure 5.13, Modules C and N are common coupled.

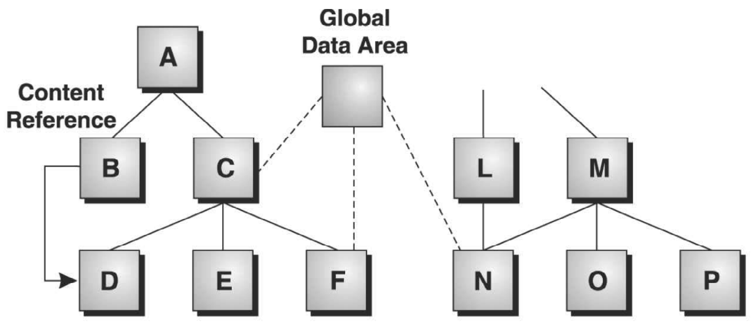


Fig. 5.13 Content and Common Coupling

Cohesion

The measure of strength of the association of elements within a module is known as cohesion. A cohesive module performs a single task within a software procedure, which has less interaction with procedures in the other part of the program. In practice, designers should avoid a low level of cohesion when designing a module. Generally, low coupling results in high cohesion and vice versa. The various types of cohesion are:

- **Functional cohesion:** In this, the elements within the modules contribute to the execution of one and only one problem-related task (Figure 5.14).

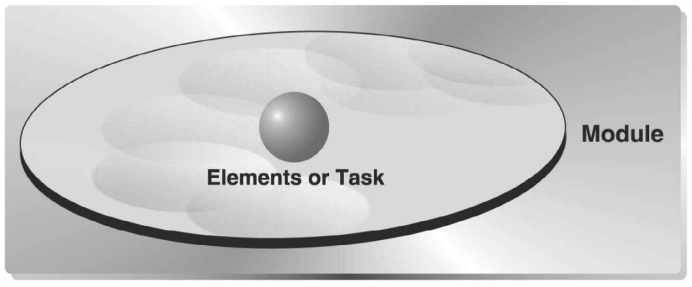


Fig. 5.14 Functional Cohesion

- **Sequential cohesion:** In this, the elements within the modules are involved in activities in such a way that output data from one activity serves as input data to the next activity (Figure 5.15).

NOTES

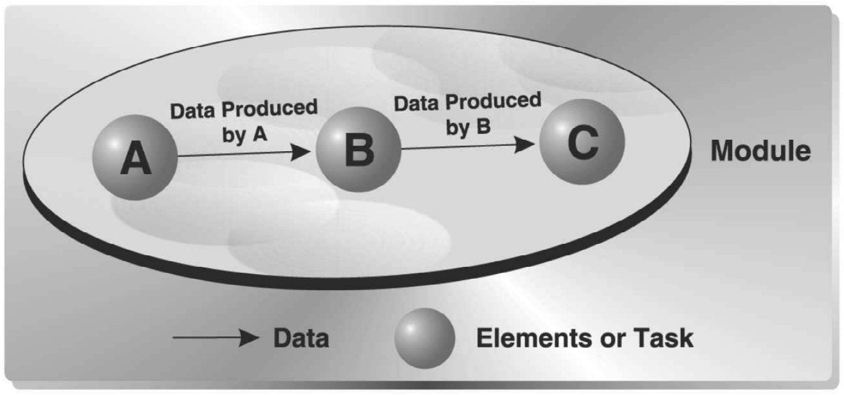


Fig. 5.15 *Sequential Cohesion*

- **Communicational cohesion:** In this, the elements within the modules perform different functions, yet, each function references the same input or output information (Figure 5.16).

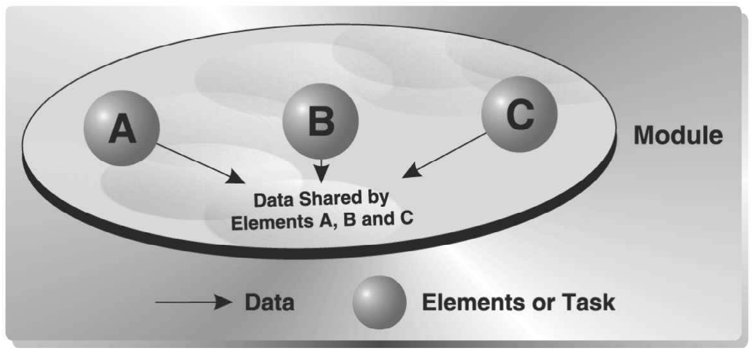


Fig. 5.16 *Communicational Cohesion*

- **Procedural cohesion:** In this, the elements within the modules are involved in different and possibly unrelated activities (Figure 5.17).

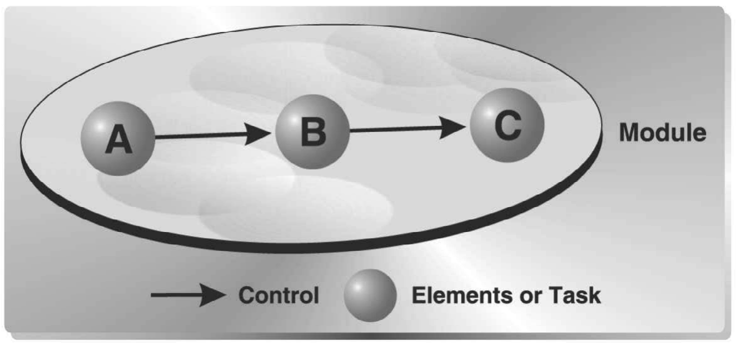


Fig. 5.17 *Procedural Cohesion*

NOTES

- **Temporal cohesion:** In this, the elements within the modules contain unrelated activities that can be carried out at the same time (Figure 5.18).

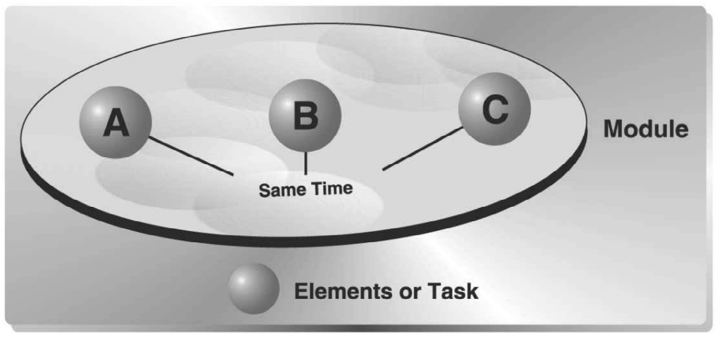


Fig. 5.18 *Temporal Cohesion*

- **Logical cohesion:** In this, the elements within the modules perform similar activities, which are executed from outside the module (Figure 5.19).

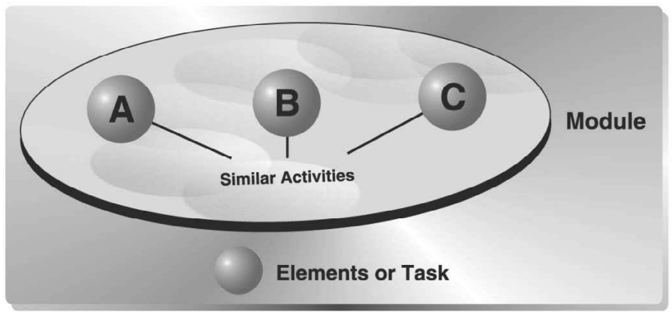


Fig. 5.19 *Logic Cohesion*

- **Coincidental cohesion:** In this, the elements within the modules perform activities with no meaningful relationship to one another (Figure 5.20).

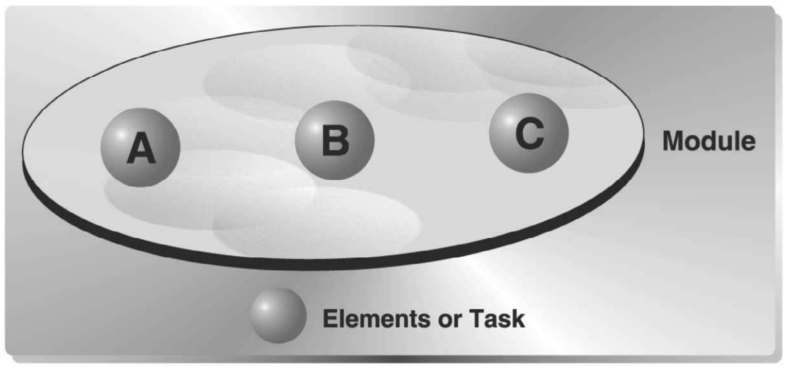


Fig. 5.20 *Coincidental Cohesion*

After having discussed various types of cohesions, Figure 5.21 illustrates the procedure, which can be used in determining the types of module cohesion for software design.

NOTES

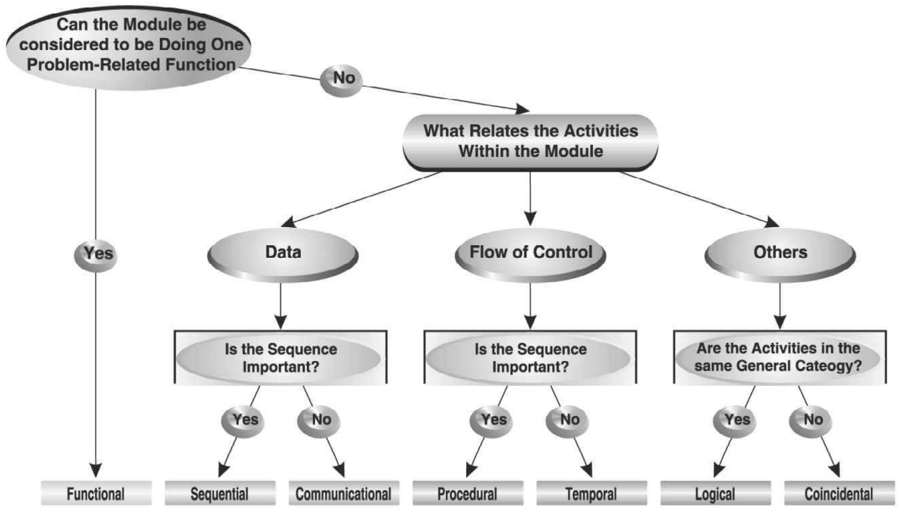

Can the Module be considered to be Doing One Problem-Related Function
No
What Relates the Activities Within the Module
Yes
Data
Flow of Control
Others
Is the Sequence Important?
Is the Sequence Important?
Are the Activities in the same General Cateogy?
Yes
No
Yes
No
Yes
No
Functional
Sequential
Communicational
Procedural
Temporal
Logical
Coincidental

Fig. 5.21 *Selection Criteria of Cohesion*

5.3.1 Design Decomposition

The two basic approaches that can be used to design a software are function-oriented and object-oriented. These approaches are fundamentally different and are applicable at different stages in the design phase.

Function-Oriented Design

Function-oriented design is an approach that is based on the decomposition of design into a set of interacting modules with each module having a specific function. For decomposing the design, a top-down design strategy called stepwise refinement is used.

Stepwise refinement decomposes the system design from a high level of abstraction into a more detailed level (lower level) of abstraction. At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data. As we proceed towards the lower levels of abstraction, more and more details are available.

Software designers start the stepwise refinement process by creating a sequence of compositions for the system being designed. Each composition is more detailed than the previous one and contains more components and interactions. The earlier compositions represent the significant interactions within the system, while the later compositions show in detail how these interactions are achieved.

To have a clear understanding of the concept, let us consider an example of stepwise refinement. Every computer program comprises *input*, *process* and *output*.

- INPUT
  - Get user's name (string) through a prompt
  - Get user's grade (integer from 0 to 100) through a prompt and validate
- PROCESS
- OUTPUT

This is the first step in refinement. The input phase can be refined further as follows.

- INPUT
  - Get user's name through a prompt
  - Get user's grade through a prompt
  - While (invalid grade)
    Ask again
- PROCESS
- OUTPUT

Note: Stepwise refinement can also be performed for PROCESS and OUTPUT phase.

Object-Oriented Design

Object-oriented design transforms the analysis model created using object-oriented analysis into the design model of a system that serves as a basis for object-oriented implementation. It gives a detailed description of how a system (solution domain) can be built using objects. The objects in an object-oriented design are related to the solution to the problem that is being solved.

The basic aim of this phase is to identify the classes in the system and their relationships. The classes and their relationships are frequently represented by the class diagrams. A *class diagram* represents a collection of classes, along with the association and relationships among them. It is a pictorial representation of the detailed system design. As we know case diagrams talk about 'what are the requirements' of a system. However, the class diagrams are designed to convert this 'what' to a 'how' for each requirement. The class diagrams give a static view of a system and reference by the developers while implementing the system.

5.3.2 Software Design Quality

Software quality refers to the extent to which the software is able to perform correctly as well as meets the required features and functions. For this, a planned and systematic set of activities is performed. IEEE defines software quality as *(1) the degree to which a system, component, or process meets specified requirements. (2) the degree to which a system, component, or process meets customer or user needs or expectations*.

Different individuals judge software quality on different basis. This is because they are involved with the software in different ways. Users, for example, want the software to perform according to their requirements. Similarly, developers involved in designing, coding and maintenance of the software evaluate the software quality by looking at the internal characteristics of the products, before delivering it to the user. However, according to the international standard, a software product is associated with various quality attributes which are as follows:

- **Functionality**: It refers to the degree of performance of the software against its intended purpose.
- **Reliability**: It refers to the ability of the software to perform a required function under given conditions for a specified period.

- **Usability**: It refers to the degree to which the software is easy to use.
- **Efficiency**: It refers to the ability of the software to use system resource in the most effective and efficient manner.
- **Maintainability**: It refers to the ease with which a software system can be modified to add capabilities, improve system performance or correct errors.
- **Portability**: It refers to the ease with which software developers can transfer software from one platform to another, without (or with minimum) changes. In simple terms, it refers to the ability of software to function properly on different hardware and software platforms without making any changes in it.

In addition to the above-mentioned attributes, *robustness* and *integrity* are also important. Robustness refers to the extent to which the software can continue to operate correctly despite the introduction of invalid input, while integrity refers to the extent to which unauthorized access or modification of software or data can be controlled in the computer system.

5.3.3 Design Description Languages

In the simplest way, the software design is specified using a natural language (like English). However, this approach may lead to a specification document that contains ambiguities, contradictions and incomplete and vague statements. Communicating design specifications in such a way can result in errors during the implementation phase where the software design is converted into the source code. Another way is to use a formal language like a programming language for specifying the detailed design. This approach is good enough for communicating the design to the developers but is not useful for the users. This is because the users are usually non-technical persons and it is difficult for them to understand the syntax of the programming language.

Therefore, we need a semi-formal method that specifies the software design precisely, completely and unambiguously. Programming design language (PDL) is one such method. PDL, also known as *structured English* or *pseudocode* is a language that is used to specify the system design using natural language (English) along with the constructs of a programming language. IEEE defines program design language as, 'A specification language with special constructs and sometimes verification protocols, used to develop, analyse and document a program design.' PDL is helpful to programmers as it ensures that the design specifications can be easily converted into the source code.

This language is mainly used to communicate the system design and can be extended to describe the logic design of the modules. The benefit of using PDL as a design notation is that it requires the programmer to focus only on the algorithms without taking care of the syntax of the programming language. Other benefits of using PDL are:

- It uses English-like statements that precisely describe specific operations.
- It supports the idea of iterative refinement.
- Program design is expressed in an easy-to-read format. Thus, the design can be converted easily into executable code.

- Reviews become easier since the source code is not examined.
- The language uses flow chart replacements, program documentation and technical communication at all levels.
- Continued use and refinement of the PDL has established it as the medium of choice for either creating or refining a detailed program design.

Developing Program Design Language (PDL)

PDL uses the keywords provided for all structured constructs of a programming language and the syntax of natural language to describe processing details. Some keywords to denote various programming processes are listed as follows:

- Input: INPUT, READ, OBTAIN, GET and PROMPT
- Output: PRINT, DISPLAY and SHOW
- Compute: COMPUTE, CALCULATE and DETERMINE
- Initialize: SET and INIT
- Add one: INCREMENT

Let us consider an example to understand the concept of program design language by designing a program to calculate interest of an account based upon the specified rate. To develop a PDL for this program, consider the points listed in Table 5.1.

Table 5.1 *Keywords and Description*

Keywords	Description
Known Values	Rate
Inputs	Principle, Time
Calculations	Interest = (Principle x Rate x Time)/100
Outputs	Interest

Using the keywords and their description listed, the PDL for the said program is given as:

```
PROMPT for Principle
READ Principle
PROMPT for Time
READ Time
CALCULATE Interest = (Principle x Rate x Time)/100
DISPLAY Interest
```

In addition to these keywords, PDL also supports some basic constructs of programming language like IF construct, CASE OF construct, and DO construct. The syntax for using each of these constructs is given here.

```
IF <condition> THEN
    <set A of statements>
ELSE
    <set B of statements>
END IF
```

```
    CASE OF <operator type>
    <statements>
END CASE
    DO WHILE [UNTIL/FOR] <condition>
<statements>
END DO
```

NOTES

Note: Various data structures such as arrays, tables, scalar, etc., can also be defined and used in PDL.

Check Your Progress

1. Define the term software design.
2. What is software architecture?
3. Give the definition of architectural design according to IEEE.
4. Define the term procedural design.
5. What is stepwise refinement in decomposition?
6. How does IEEE define PDL?

5.4 USER INTERFACE DESIGN

User interfaces determine the way in which users interact with the software. The user interface design creates an effective communication medium between a human and a computing machine. It provides easy and intuitive access to information as well as efficient interaction and control of software functionality. For this, it is necessary for the designers to understand what the user requires from the user interface (Figure 5.22).

Since the software is developed for the user, the interface through which the user interacts with the software should also be given prime importance. It is important to first know the person (user) for whom the user interface is being designed before designing the user interface. Direct interaction with end-users helps the developers to improve the user interface design because it helps designers to know the user's goals, skills and needs.

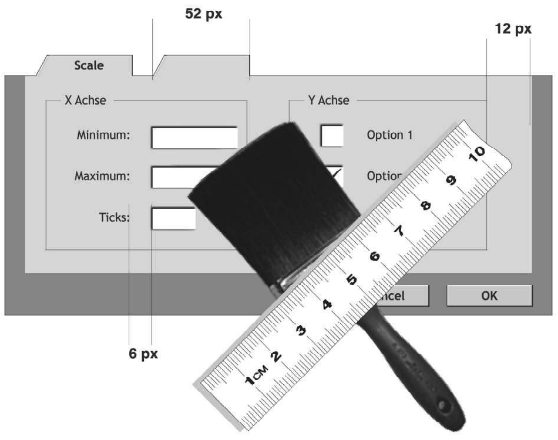


Fig. 5.22 *Simple User Interface Design*

User Interface Rules

Designing a good and efficient user interface is a common objective among software designers. But what makes a user interface looks 'good'? Software designers strive to achieve a good user interface by following three rules, namely, ease of learning, efficiency of use and aesthetic appeal.

Ease of Learning

It determines the degree of ease with which a user can learn to interact with the software. This is especially important for novice users. However, sometimes, experienced users also face problems during the learning stage when using a new version of the software or while expanding their knowledge of using the software. Here, the principle of state visualization is applied, which states that whenever a change is made in the behaviour of the software, it should be reflected in the interface appearance.

While designing the user interface, designers should focus on the ease of learning for the class of users having little or no knowledge, since only few users have expertise in using all the features of the user interface. Generally, to ease the task of learning, designers make use of the following tools:

- **Affordance:** Provides hints about the features of a tool and the suggestions for using these features. For example, the style of cap on some bottles indicates how to open it that is, by rotating it clockwise or anti-clockwise. If no such indication is provided, one has to struggle while opening the bottle. Therefore, in this case, the bottle cap is not just a tool that physically helps to open the bottle but it is also an affordance that shows how to open the bottle. Similarly, software designers should provide suggestion regarding the functioning of each part of the interface and how does it functions.
- **Consistency:** Designers should attempt to keep the interface design consistent internally as well as externally. Internal consistency means behaviour of the program including minor details like colour, font, etc. should be uniform while using different parts of the program. For example, the colour of pop-up menu should be uniform throughout the application. On the other hand, external consistency means the consistency of the program with the external environment. That is, the program should be consistent with the operating system in which it is running as well as with other applications that are running within that operating system.

Efficiency of Use

Once a user knows how to perform tasks, the next question is how efficiently the interface can be used by a user to accomplish a task? If the user is now engaged in performing the tasks instead of learning how to do, then only the efficiency can be evaluated reasonably.

To design an efficient interface, a thorough knowledge of behaviour of the users who are going to use the application is essential. For this, the designer should keep the following factors in mind:

- Frequency of using the interface

- The level of users (novice, intermediate, or expert)
- Frequency of performing a particular task

In addition to these factors, the following guidelines can also help in designing an efficient interface:

- User should be left to perform the minimal physical actions to accomplish a task. For this, the interface should provide the shortcut keys for performing the tasks that need to be commonly performed frequently, since it reduces the work of users.
- Users should be left to perform minimal mental effort as well. For this, the interface should not assume the user to remember much detail, instead it should provide the necessary instructions for performing a task. For example, the interface should provide the file browser to select a file, instead of assuming the user to remember the path of the required file.

Aesthetically Pleasing

Today, look and feel is one of the biggest USPs (Unique Selling Points) while designing a software. A user always prefers to buy things that look attractive as well as make him to feel better while using the product. Many software organizations focus specifically on designing software, which has an attractive look and feel so that they can lure customers/users towards their product(s).

In addition to the afore-mentioned goals, there exists a principle of metaphor which, if applied to the software's design result in a better and effective way of creating a user interface. This principle states that if the user interface for a complex software system is designed in such a way that looks like that of an already developed system, it becomes easier to understand. For example, the popular Windows operating system uses similar (not same) look and feel in all of its operating systems so that the users are able to use it in a user-friendly manner.

User Interface Design Process

The user interface design, like all other software design elements, is an iterative process. Each step in this design occurs a number of times, each time elaborating and refining information developed in the previous step. Although many different user interface design models have been proposed, all have the following steps in common:

- Using information stated in the requirements document, each task and actions are defined.
- A complete list of events (user actions), which causes the state of the user interface to change is defined.
- Each user action is assigned iteration.
- Each user interface state, as it will appear (look) to the user is depicted.
- On the basis of the information provided by the user interface, the users' interpretation about the state of system is indicated.

To sum up, the user interface design activity starts with the identification of user, task, and environmental requirements. After this, user states are created and analyzed to define a set of interface objects and actions. These object then form the basis for the creation of screen layout, menus, icons and much more.

While designing the user interface, the following points must be kept in mind:

- Follow the rules discussed under user interface. If an interface does not follow any of these rules to a reasonable degree, then it needs to be redesigned.
- Determine how interface will be implemented.
- Consider the environment (like operating system, development tools, display properties and so on).

Evaluating a User Interface Design

The user interface design process generates a useful interface; however, it cannot be expected from a designer to design a good quality interface in the first run. Each iteration of the steps involved in user interface design process leads to development of a prototype. The objective of developing a prototype is to capture the 'essence' of the user interface. The prototype is evaluated, discrepancies are detected and accordingly redesign takes place. This process carries on until a good interface evolves.

Evaluating a user interface requires its prototype to include the look and feel of the actual interface and should offer a range of options. However, it is not essential for the prototype to support the full range of software behaviours. Choosing an appropriate evaluation technique helps in knowing whether a prototype is able to achieve the desired user interface or not.

Evaluation Techniques

Evaluation of interface must be done in such a way that it can provide feedback to the next iteration. Each iteration of evaluation must specify what is good about the interface and where is a scope for improvement.

Some well-known techniques for evaluating user interface are: use it yourself, colleague evaluation, user testing and heuristic evaluation. Each technique has its own advantages and disadvantages, and the emphasis of each technique for various issues like ease of learning and efficiency of use varies. The rule of aesthetic pleasure largely varies from user to user and can be evaluated well by observing what attracts people:

- **Use it yourself:** This is the first technique of evaluation and in this technique the designer himself use the interface for a period of time to determine its good and bad features. It helps the designer to remove the bad features of the interface. This technique also helps in identifying the missing components of the interface and its efficiency.
- **Colleague evaluation:** Since the designers are aware of the functionality of the software, it is possible that they may miss out on issues of ease of learning and efficiency. Showing the interface to a colleague may help in solving these issues. If the prototype is not useful in the current state, colleagues might not spend sufficient time in using it to identify many efficiency issues.

- **User testing:** This testing is considered to be the most practical approach of evaluation. In this technique, users test the prototype with the expected differences recorded in the feedback. The communication between the users while testing the interface provides the most useful feedback. Before allowing the users to start testing, the designers choose some tasks expected to be performed by the users. In addition, they should prepare the necessary background details in advance. Users should spend sufficient time to understand these details before performing the test. This testing is considered the best way to evaluate ease of learning. However, it does not help much in identifying the inefficiency issues.
- **Heuristic evaluation:** In this technique of evaluation, a checklist of issues is provided, which should be considered each time the user interface is evaluated. Such evaluations are inherently subjective in a sense that different evaluators find different set of problems in the interface. Usually, an experienced designer detects more errors than a less experienced designer. However, we cannot expect from a single person to detect all the errors. Therefore, it is always beneficial to employ multiple persons in heuristic evaluation. The checklist provided in such evaluation includes the following issues:
  - The interface should include as minimum extraneous information as possible.
  - The interface should follow a natural flow and should lead the users' attention to the next expected action. For example, if the user is going to close an application without saving a task, then it must prompt the user to save or discard the changes.
  - There should be a consistency in the use of colour and other graphical elements. This helps the user to easily understand the information conveyed by the interface.
  - The information should be presented to the user in an ordered manner and the amount of information should be as minimum as per required by the user.
  - The messages should be displayed always from the user's perspective and not from the system's. For example, the messages like 'you have purchased...' should be displayed rather than the messages like 'we have sold you...'.
  - The interface should represent the sufficient information to the user whenever required thereby minimizing the load on user's memory.
  - The behaviour of program should remain consistent internally as well as externally. For example, for representing similar items, same types of visual descriptions are used.
  - The user must be provided with a clear feedback about the actions of a program.
  - The interface should provide shortcuts, like hot keys, icons, etc. that help the users to speed up their operations as well as improve the effectiveness of a user interface.

- The error messages are provided to communicate to the user that some error has occurred and these are the possible ways to correct it. These messages should describe exactly which part of the input has caused error.

If a user interface is carefully evaluated under these guidelines, almost all the problems related to learning and efficiency are disclosed.

5.5 DATA FLOW DIAGRAMS

Design notations are used to represent software design. These notations are important as they help designers to represent various aspects of software design like modules, abstraction, information hiding, concurrency, architecture of a program, etc., in a comprehensive manner. A design representation serves the purpose of designers and stakeholders in the following ways:

- The designers try to detect missing or inconsistent aspects of a proposed solution at the earlier stages of design.
- Other stakeholders (programmers testers or maintainers) try to understand the designer's intent.

A good design notation helps to clarify the relationships and interactions that exist in the design, while a poor notation generally complicates the design process by interfering with the software design practice. A software design notation can be diagrammatic, symbolic or textual. The various notations that are commonly used are flowcharts, data-flow diagrams, structure charts, HIPO diagram, decision table, and program design language.

Flowcharts

A flowchart is a graphical representation of an algorithm where sequence of steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows. The boxes represent operations and the arrows represent the sequence in which the operations are implemented. A flowchart helps to determine the major elements of a process by creating boundaries between ending of one process and starting of another process. In addition, it helps the programmer in understanding the logic of the program. Therefore, it is always not necessary to include all the required steps in detail. A properly developed flowchart provides a clear understanding of the process. Generally, a flowchart comprises various symbols, which are listed in Table 5.2.

Table 5.2 *Flowchart Symbols*

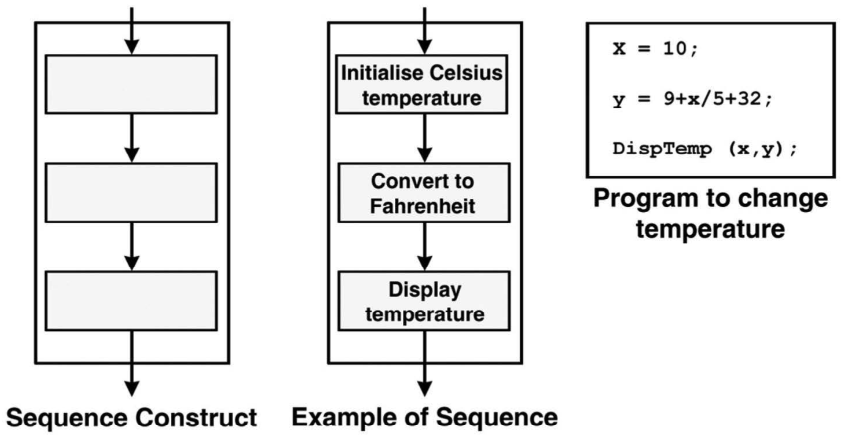
Name	Notation	Description
Flow line		Connects the flowchart symbols and shows the sequence of operations during the program execu tion
Terminator		Shows the starting and ending points of the program. A terminator has flow lines in only one direction, either in (a stop node) or out (a start node).

NOTES

Data input or output	Allows the user to input data and also allows results to be displayed.
Processing	Indicates an operation performed by the computer, such as a variable assignment or mathematical operation.
Decision	Indicates a decision structure. A diamond always has two flow lines out. One flow line out is labelled as 'yes' branch and the other is labelled as 'no' branch.
Predefined process	Denotes a group of previously defined statements. For example, 'Calculate n!' indicates that the program executes the necessary commands to compute n factorial.
Connector	Avoids crossing flow lines, making the flowchart easier to read. Connectors indicate where flow lines are connected. Note that connectors come in pairs, one with a flow line in and the other with a flow line out.
Off-page connector	Indicates the continuation of the flowchart on another page. Just like connectors, off-page connectors come in pairs too.

Structured flowcharts are generally used in situations where clarity of control flow is of utmost importance. These flowcharts are a graphical equivalent of the pseudocode description. They differ from traditional flowcharts in a way that in the former, only combinations of some basic structured constructs like sequence, selection, and iteration are used. The structured constructs used in structured flowcharts are listed as follows:

- **Sequence:** In this construct, a number of processes are performed one after the other. The processing boxes are connected by a line (arrow) of control in this construct (see Figure 5.23).

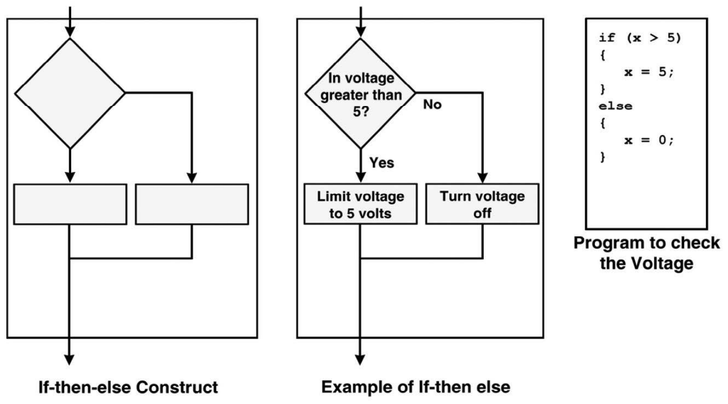


```
X = 10;
y = 9+x/5+32;
DispTemp (x,y);
```

Fig. 5.23 Sequence Construct

- **If–then–else:** In this, one of the processes is carried out. That is, if the then part is true, then the then-part processing takes place otherwise; the if-else part will invoke else-part processing (see Figure 5.24).

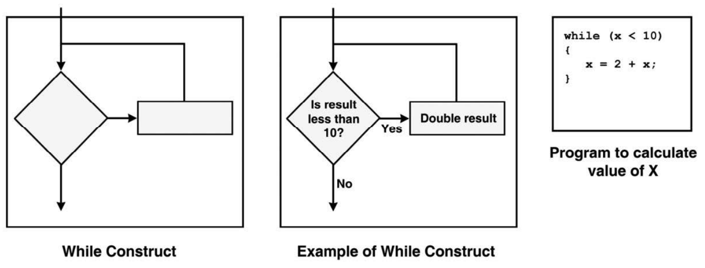
NOTES



```
if (x > 5)
{
   x = 5;
}
else
{
   x = 0;
}
```

Fig. 5.24 If–then–else Construct

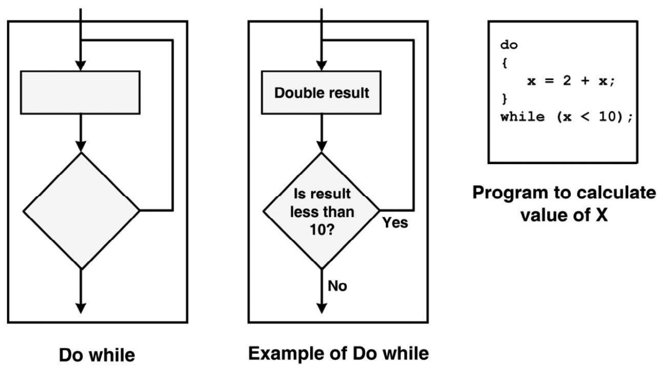
- **While:** This construct allows a process to be executed repeatedly as long as some condition is satisfied within a program. The condition is evaluated before the execution of the process. If condition is not satisfied for the first time, the process is not executed at all (see Figure 5.25).



```
while (x < 10)
{
   x = 2 + x;
}
```

Fig. 5.25 While Construct

- **Do–while:** This construct is similar to while except that in this, the process is executed at least once even if the condition is not satisfied for the first time (see Figure 5.26).

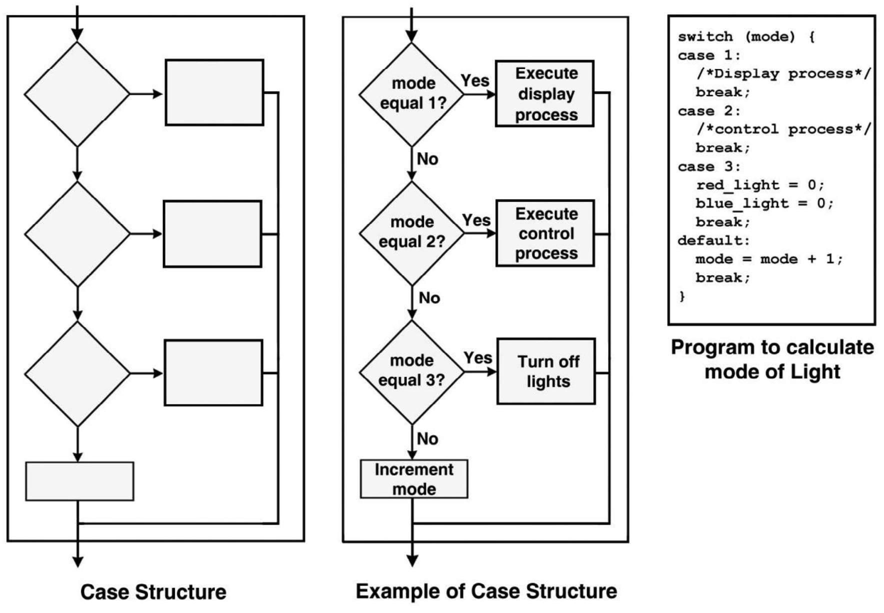


```
do
{
   x = 2 + x;
}
while (x < 10);
```

Fig. 5.26 Do–While Construct

- **Select-case:** This construct is useful when a number of if-then-else statements are to be represented. That is, select case is helpful in case we have a number of options to select from. In this construct, a specified parameter is tested until a true condition occurs and the case part of the construct is executed, as shown in Figure 5.27.

NOTES



```
switch (mode) {
case 1:
  /*Display process*/
  break;
case 2:
  /*control process*/
  break;
case 3:
  red_light = 0;
  blue_light = 0;
  break;
default:
  mode = mode + 1;
  break;
}
```

Fig. 5.27 *Case Constructs*

5.5.1 Structure Charts

Structure charts are the graphical representation of the high-level design, or architecture, of a computer program. As a design notation, structure charts show the control and data flow between modules. This flow includes showing all communications that takes place between the modules. This chart also helps the programmer to divide a problem into smaller parts so that the problem can be easily understood.

Programmers use structure charts to build a program in a manner similar to how an architect uses a 'blueprint' to build a house. In the design stage, the structure chart is drawn and is used by the client and the various software designers to communicate between them. During the actual building of the program (implementation), the chart is continually referred to as the 'master plan'. Often, this chart is modified as programmers learn new details about the program. After a program is completed, the structure chart is used to fix bugs and make changes (maintenance) to the module. A structure chart depicting an organization's structure is shown in Figure 5.28, where the structure comprises a rectangular box (represents an individual module) and various other symbols.

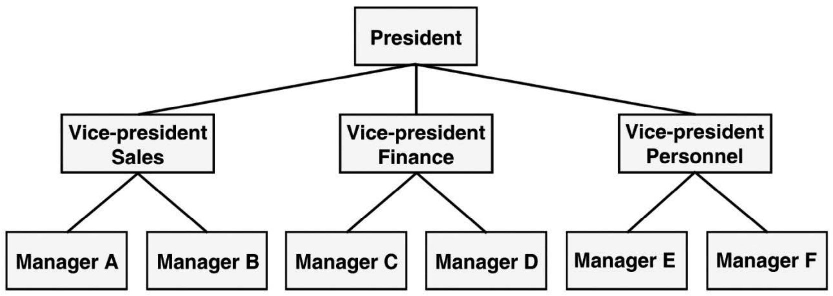


Fig. 5.28 *An Organization's Structure Chart*

The various symbols used in a structure chart are shown in Figure 5.29.

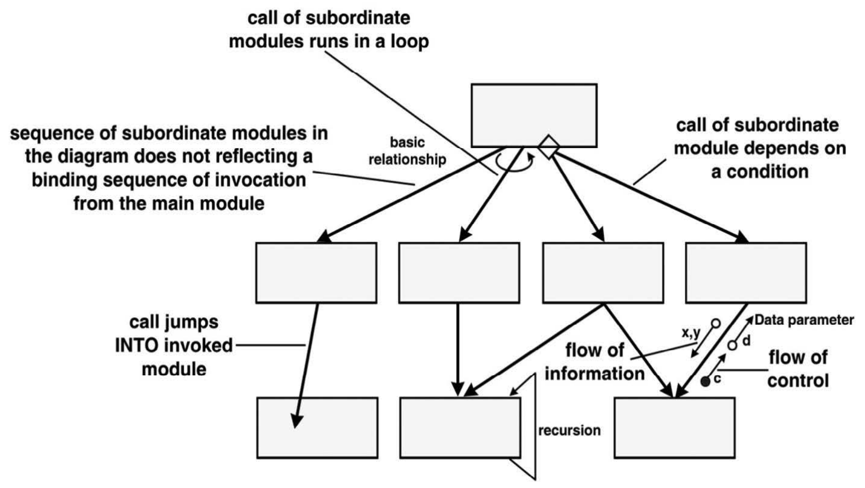

call of subordinate modules runs in a loop
sequence of subordinate modules in the diagram does not reflecting a binding sequence of invocation from the main module
basic relationship
call of subordinate module depends on a condition
call jumps INTO invoked module
flow of information
x,y
d
c
Data parameter
flow of control
recursion

Fig. 5.29 *Symbols in Structure Chart*

Developing a Structure Chart

To create a structure chart, perform the following steps:

1. Depict the relation of modules from top to bottom.
2. Conceptualize the major activities to be performed to solve the problem. These activities are represented by the nodes below the root, and connection between the root and the nodes is represented by drawing connecting lines.
3. Next, the programmer focuses on each activity individually, and conceptualizes how each module can be broken down into even smaller tasks. Eventually, the program is broken down to a point where the leaves of the tree represent simple methods that can be coded with just a few program statements.
4. Once the structure chart has been designed a process called bottom-up implementation and testing is used to implement each module.

Let us consider an example to understand the concept of structure charts by designing a program to read three characters and print those characters in ascending order. To construct a structure chart for this program, follow these steps:

1. Take three inputs-char_1, char_2, and char_3.
2. Construct an input–output diagram on the basis of the input–output table (Table 5.3).

Table 5.3 *Input–Output table*

Input	*Processing*	*Output*
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters	char_3
		Displays three characters

3. Group the activities into two main activities namely, Read_three_characters, and Sort_three_characters.
4. Next, create the structure chart (see Figure 5.30) showing both the activities.

NOTES

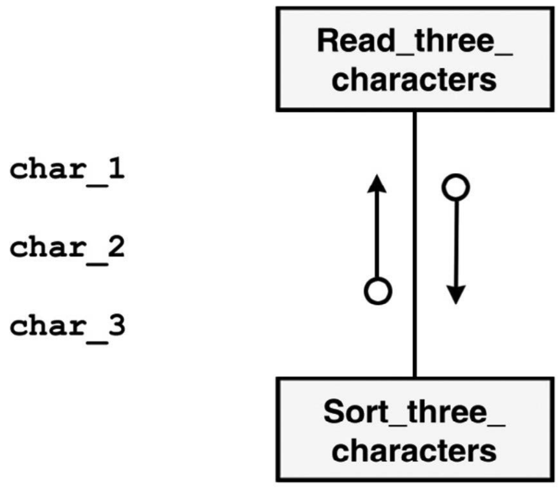


Fig. 5.30 *Structure Chart for the Program*

5.5.2 HIPO Diagram

HIPO (Hierarchy-Input-Process-Output) diagrams were designed by IBM in the 1970s as a design notation for representing system design. These diagrams further became an excellent input for detailed program design. In addition, system analysts have also used HIPO diagrams for representing a top-level view of the functions performed by a system and the decomposition of these functions into sub functions and so on. Thus, it can be said that HIPO diagram is as useful to analysts as the program design language is to programmers.

A HIPO diagram generally consists of the following:

- A collection of top-level diagrams.
- A collection of detailed diagrams.
- A visual table of contents (VTOC), which consists of tree or graph structured directory, summary of contents in each top-level diagram and a legend of symbol definitions.

HIPO diagrams are similar to the organization chart, which represents the management hierarchy of the organization. Therefore, HIPO diagrams can be used as a modelling tool in some environments. A sample HIPO diagram is shown in Figure 5.31, where each function is represented by a rectangular box.

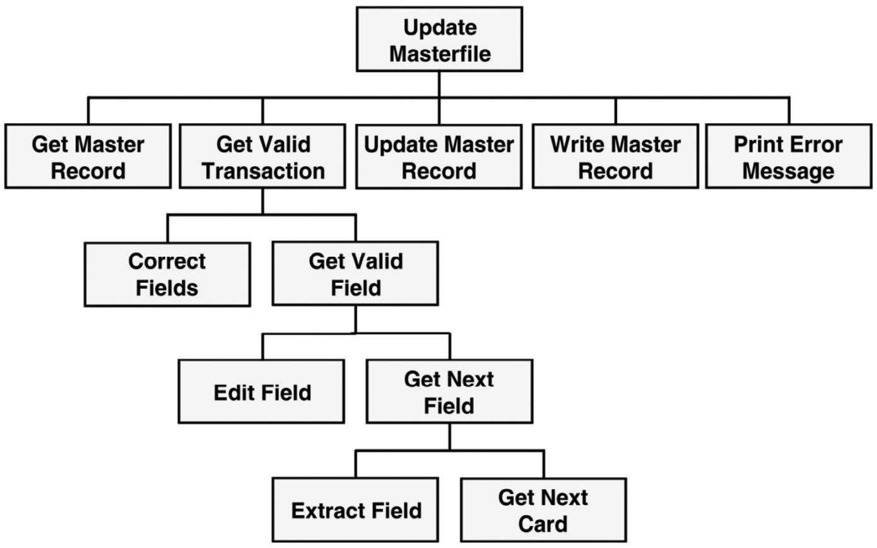


Fig. 5.31 *HIPO Diagram*

5.5.3 Decision Table

Decision table may be defined as a table that contains all the possible conditions for a specific problem and the corresponding results using condition rules that connect conditions with results. These tables are used to specify the complicated decision logic. A decision table is composed of rows and columns, which contain four separate quadrants, as shown in Figure 5.32.

Conditions	**Condition Alternatives**
Actions	**Action Entries**

Fig. 5.32 *Four Quadrants of a Decision Table*

- **Conditions** (upper left quadrant): Contains a list of all the possible conditions pertaining to a problem.
- **Condition alternatives** (upper right quadrant): Contains the condition rules (set of possible values for each condition) of alternatives.
- **Actions** (lower left quadrant): Contains actions, which can be a procedure or operation to be performed.
- **Action entries** (lower right quadrant): Contains the action rules, which specify the actions to be performed on the basis of the set of condition alternatives corresponding to that action entry.

Each column in the quadrant can be interpreted as a processing rule. The basic four-quadrant structure is common for all the decision tables, however, they may differ in the way of representation of the condition alternatives and action entries. For example, in some decision tables, condition alternative are represented using true/false values while in other tables, Yes/No, numbers (0 or 1), fuzzy logic or probabilistic representations may be used. Similarly, in some decision tables, action entries are represented by placing check marks, cross marks or numbering the actions to be performed.

Developing a Decision Table

In order to build decision tables, first the maximum size of the table is determined. Any impossible situations, inconsistencies, or redundancies are then eliminated from the table. To develop a decision table, follow these steps:

1. Determine all actions that can be associated with a specific module or procedure.
2. Determine the number of conditions that may affect the decision.
3. Associate the specified conditions with specific actions. In the simplest form of decision table, there are two alternatives (Y or N) for each condition.
4. Define rules by indicating actions that occur for the specified conditions.

Let us consider an example to understand the concept of decision table by designing a program for a phone card company that sends monthly bills to its customers. If the customers pay their bills within two weeks, they are offered discount as per the scheme given below:

- If the amount > 350, then the discount is 5%.
- If the amount >= 200 and amount <= 350, then the discount is 4%.
- If the amount < 200, there is no discount.

Refer to the decision table (Table 5.4).

Table 5.4 *Decision Table*

Conditions	**Rules**			
	1	**2**	**3**	**4**
Paid within 2 weeks	Y	Y	Y	N
Order over Rs 350	Y	N	N	-
Rs 200 < = order < = Rs 350	N	Y	N	-
Order < 200	N	N	Y	-
Actions				
5% discount	X			
4% discount		X		
No discount			X	X

5.5.4 Concurrency

A computer has limited resources which must be utilized efficiently and effectively. To utilize these resources efficiently, multiple tasks must be executed concurrently. This requirement makes concurrency one of the major concepts of software design. Every system must be designed to allow multiple processes to execute concurrently, whenever possible. When, for example, the current process is waiting for some event to occur, the system must execute some other process in the mean time.

However, concurrent execution of multiple processes sometimes may result in undesirable situations, such as inconsistent state and deadlock. As for example, consider two processes A and B, and a data item Q1 with the value 200. Further, suppose A and B are executing concurrently, and firstly A reads the value of Q1 (which is 200) to add 100 to it. However, before A updates the value of Q1, B reads the value of Q1 (which is still 200) to add 50 to it. In this situation, whether A or B first updates the value of Q1, the value of Q1 would definitely be wrong resulting in inconsistent state of the system. This is because the actions of A and B are not synchronized with each other. Thus, the system must control the concurrent execution and synchronize the actions of the concurrent processes.

One way to achieve synchronization is mutual exclusion which ensures that two concurrent processes do not interfere with the actions of each other. To ensure this, mutual exclusion may use locking technique. In this technique, the processes need to lock the data item to be read or updated. The data item locked by some process cannot be accessed by other processes until unlocked. It implies that the process that needs to access the data item locked by some other process has to wait.

5.6 OBJECT-ORIENTED DESIGN

Object-oriented design (OOD) transforms the analysis model created using the object-oriented analysis (OOA) into a design model of a system that serves as a basis for object-oriented implementation. It gives a detailed description of *how* a system (solution domain) can be built using objects. The objects in an object-oriented design are related to the solution to the problem.

To understand object-oriented design, it is important to understand various concepts used in object-oriented environment such as *objects*, *classes*, *encapsulation*, *inheritance, polymorphism and so on*. These concepts are listed in Table 5.5.

Table 5.5 *Object-Oriented Concepts*

Object-Oriented Concept	***Description***
Object	An instance of a class used to describes the entity
Class	A collection of similar objects which encapsulates data and procedural abstractions in order to describe their states and operations to be performed by them
Attribute	A collection of data values that describe the state of a class
Operation	Also known as method or service, it provides a means to modify the state of a class
Superclass	Also known as base class, it is a generalization of a collection of classes related to it
Subclass	A specialization of superclass that inherits the attributes and operations from the superclass
Inheritance	A process in which an object inherits some or all the features of a superclass
Polymorphism	An ability of objects to be used in more than one form in one or more classes

NOTES

Abstract class	A class that provides only the interface of one or more functions and not their implementations
Concrete class	The class that provides an implementation for all the functions of the abstract class
Message passing	A process of interacting between different objects in a program. The messages are exchanged by calling the member functions of the classes
Containership (containment or aggregation or composition)	Nesting in which a class has an object of another class as its member

Object-Oriented Design Paradigm

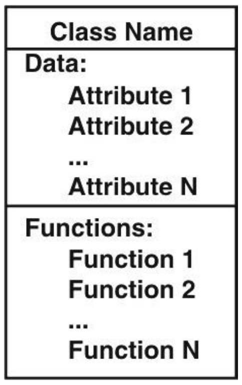
The basic aim of the object-oriented design phase is to identify the classes in the system and their relationships. The classes and their relationships are frequently represented by the class diagrams. A *class diagram* represents a collection of classes, along with the association and relationships among them. It is a pictorial representation of the detailed systems design. As we know, the use-case diagrams talk about 'what are the requirements' of a system. However, class diagrams are designed to convert this 'what' to a 'how' for each requirement. Class diagrams give a static view of a system and are referenced by the developers while implementing the system.

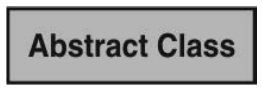
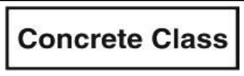
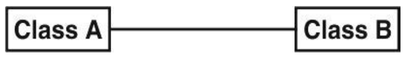
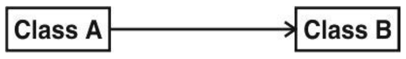
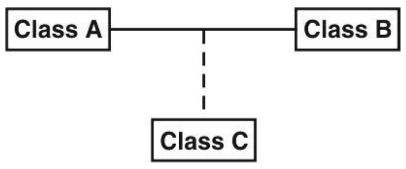
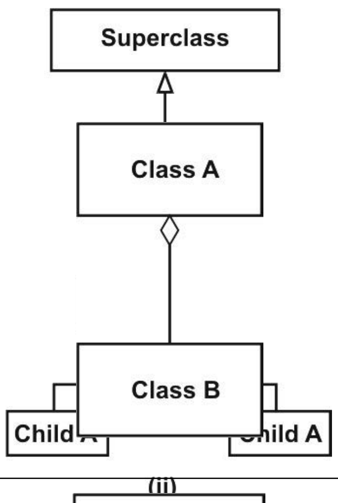
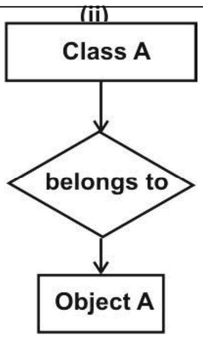
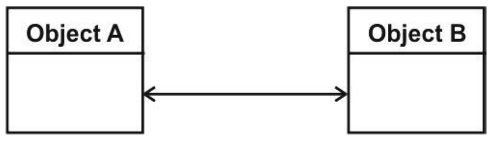
Graphical Notations

In object-oriented design, all the basic elements of an object-oriented software, such as classes, objects, user-interfaces, subclasses and relationships among various classes, are represented graphically. The graphical notations of these elements not only give a clear picture of the whole system but also increase the productivity of software developers. Note that there are no standard diagrams or notations for representing these elements; however, object-oriented design often uses unified modelling language (UML). UML is a standard language used in object-oriented design to graphically represent all the system elements.

Some of the commonly used elements such as classes, objects, message passing among various objects, inheritance and containment can be represented using UML. UML diagrams for these elements along with their descriptions are described in Table 5.6.

Table 5.6 *Various UML Diagrams*

UML Diagram	***Description***
 Class Name Data: Attribute 1 Attribute 2 ... Attribute N Functions: Function 1 Function 2 ... Function N	In UML notation, a class is represented by a rectangular box having three segments—class name, a set of attributes and a set of operations that can be performed on the objects of that class

	An abstract class is represented by a shaded rectangle containing the class name
	A concrete class is represented by a single rectangle containing the class name
	A direct line between two classes denotes the association between them
	A direct line with an 'open arrow' is used to denote a one-way or directed association
	A dashed line connected to the association line is used to denote an association or a link class, which indicates that the association between the two classes has its own set of attributes
	A direct line with a hollow triangle denotes the generalization (or inheritance) relationship. It denotes is 'a-kind-of' relationship A direct line with a diamond denotes the containership relationship between two classes. For example, Class A is an aggregation of (or contains) objects of Class B. In other words, Class A "has-a" Class B
	A diamond-shaped box connecting the class with its corresponding object(s) denotes the 'belongs-to' or 'is-a' relationship between a class and its objects. For example, a car (object) is-a vehicle (class). In other words, a car (object) belongs-to vehicle (class)
	A direct line with open arrows at both the ends denotes the interaction between two objects through message passing

5.7 DESIGN QUALITY ASSURANCE

NOTES

Software quality assurance is concerned with process quality and refers to planned and systematic set of activities that ensure that software life cycle processes and products conform to requirements, standards, and procedures. In addition, SQA assures that the standards, processes and procedures are appropriate for the software project and are correctly implemented. A high-quality product can be achieved if there is an appropriate process quality. The objectives of SQA are listed as follows:

- To determine the objectives of the software which are to be accomplished.
- To establish software plans, when the objectives are determined.
- To monitor and adjust software plans to satisfy user requirements.

Quality Costs

Costs incurred in performing quality-related activities are referred to as quality costs. These activities include preventing, finding and correcting errors in the software. The quality costs are divided broadly into three categories, namely:

1. **Prevention costs:** These costs are incurred while examining the activities that affect software quality during the software's development. These costs are particularly incurred to prevent poor quality in software. For example, preventive costs are used to avoid errors in coding and designing. In addition, it includes the cost incurred on correcting mistakes in user manuals, staff training, requirements analysis and fault tolerant design of software.
2. **Appraisal costs:** These costs are incurred in verifying or evaluating the product before the software is delivered to the user if the software quality is found inadequate. For example, appraisal costs include cost on conducting code inspection, glass box testing, black-box testing, training testers and beta testing.
3. **Failure costs:** These costs are incurred when the product does not meet user requirements. For example, failure costs are incurred in fixing errors while dealing with user complaints. Failure costs are further divided into:
   (i) **Internal failure costs:** These costs are incurred before the software is delivered to the user. When software is unable to perform properly due to errors in it, internal failure costs are incurred for missed milestones and the over-time done to complete these milestones within the specified schedule. For example, internal failure costs are incurred in fixing errors that are detected in regression testing.
   (ii) **External failure costs:** These costs are incurred after the software is delivered to the user. When errors are detected after software is delivered, a lot of time and money is consumed to fix the errors. For example, external failure costs are incurred on the modification of software, technical support calls, warranty costs and other costs imposed by law.

NOTES

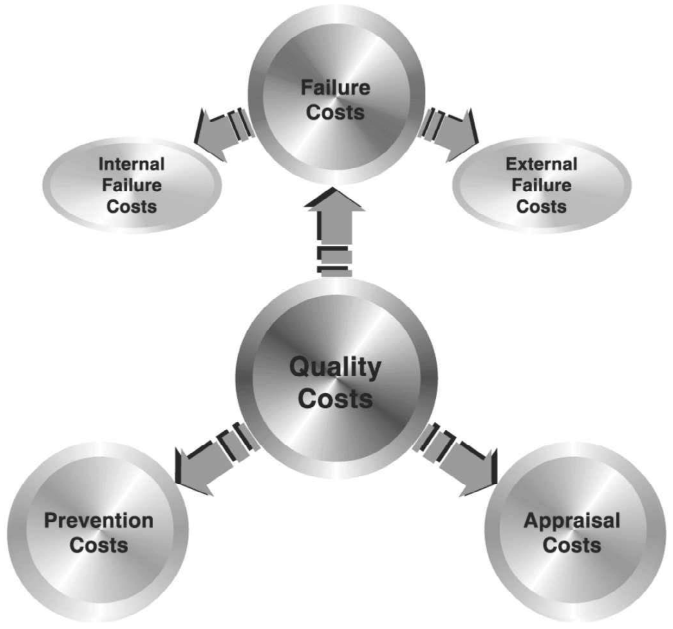


Fig. 5.33 *Types of Quality Costs*

Note: The total quality cost incurred on the software is the sum of the costs shown in Figure 5.33.

ISO 9126 Quality Factors

Software is said to be of high quality if it is free from errors and is according to user requirements. In order to obtain the desired quality, factors commonly defined in international organizations for standardization (ISO) 9126 are followed. Figure 5.34 shows the factors responsible for software quality.

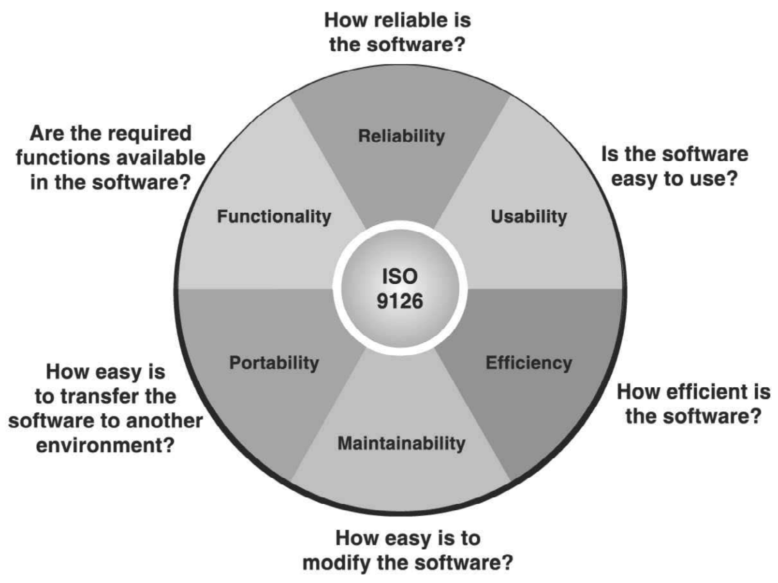

How reliable is the software?
Reliability
Are the required functions available in the software?
Functionality
Is the software easy to use?
Usability
ISO 9126
Portability
Efficiency
How easy is to transfer the software to another environment?
How efficient is the software?
Maintainability
How easy is to modify the software?

Fig. 5.34 *ISO 9126 Quality Factors*

The factors, which act as a checklist to assess quality of software (see Figure 5.35), have been discussed as follows:

1. **Functionality:** The capability of the software to achieve the intended purpose using functions and their specified properties is referred to as functionality. The attributes of functionality are:
   - (i) **Suitability:** Ability of the software to perform a set of functions that are used for specified tasks.
   - (ii) **Accuracy:** Appropriateness of the results in accordance with user requirements.
   - (iii) **Interoperability:** Ability of the software to interact with other systems.
   - (iv) **Security:** Prevention of unauthorized access to programs in the software.
2. **Reliability:** The capability of software to maintain its performance level under stated conditions for a period of time is referred to as reliability. The attributes of reliability are:
   - (i) **Fault tolerance:** Ability of software to perform its functions when unexpected faults occur.
   - (ii) **Maturity:** Ability of software to handle frequency of failures.
   - (iii) **Recoverability:** Ability of software to recover data that is lost due to software failure.
3. **Usability:** The capability of software to be easily operated and understood is referred to as usability. The attributes of usability are:
   - (i) **Understandability:** Ability of user to acknowledge the application of software.
   - (ii) **Learnability:** Ability of user to learn the functioning of software.
   - (iii) **Operability:** Ability of user to operate software with ease.
4. **Efficiency:** The capability of software to perform with optimal resources under specified conditions is referred to as efficiency. The attributes of efficiency are:
   - (i) **Time behaviour:** Response time required by software to perform a function.
   - (ii) **Resource behaviour:** Resources required to perform functions of software.
5. **Maintainability:** The capability of software to make specific modifications, which include correcting, improving, or adapting to the software is referred to as maintainability. The attributes of maintainability are:
   - (i) **Testability:** Ability of software to be easily tested and validated.
   - (ii) **Changeability:** Ability of software to be modified for fault removal and environmental change.
   - (iii) **Analysability:** Ability of software to allow detection of the cause of faults and failures.

(iv) **Stability:** Ability of software to withstand the risks occurring due to modifications.

6. **Portability:** The capability of software to be transferred from one hardware or software environment to another. The attributes of portability are:
   (i) **Installability:** Effort with which software is installed in the specified environment (both hardware and software environment).
   (ii) **Conformance:** Effort with which software complies with the standards that are related to portability.
   (iii) **Replaceability:** Effort with which software is used in place of other software for a particular environment.
   (iv) **Adaptability:** Ability of software to adapt to new environment or new technology.

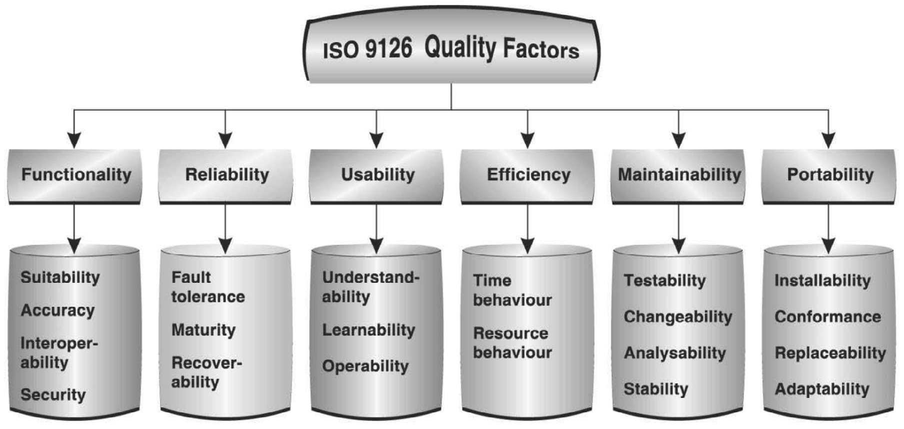


Fig. 5.35 *ISO 9126 Quality Sub-Attributes*

Mc Call's Quality Factors

There are several factors, which are used as a measure to enhance quality in software. These factors are referred to as Mc Call's quality factors. As shown in Figure 5.36, these factors are characterized according to the following three important aspects:

1. **Product operation:** It checks whether the software is able to perform the desired function efficiently or not. It comprises the following factors:
   (i) **Correctness:** The effort with which software programs perform according to user requirements.
   (ii) **Reliability:** The effort with which the software is expected to perform its specified functions accurately.
   (iii) **Usability:** The effort with which software programs are understood and operated.
   (iv) **Integrity:** The degree to which unauthorized access to the software can be controlled.
   (v) **Efficiency:** The degree to which software code and computing resources required by software programs are carried out using specified functions.

2. **Product revision:** It checks whether the software requires effort to modify it or not. It comprises the following factors:
   (i) **Maintainability:** The effort with which software can be easily maintained so that less time is taken to fix errors.
   (ii) **Flexibility:** The effort with which a software program is modified.
   (iii) **Testability:** The effort required to test a software program to ensure that it executes the required functions.
3. **Product transition:** It checks whether the software is adaptable to new environment or not. It comprises the following factors:
   (i) **Portability:** The effort required to transfer a software program from one hardware or software environment to another.
   (ii) **Reusability:** The degree to which a software program can be reused in other applications depending on the scope and functions performed by other software programs.
   (iii) **Interoperability:** The effort with which a system integrates with another system.

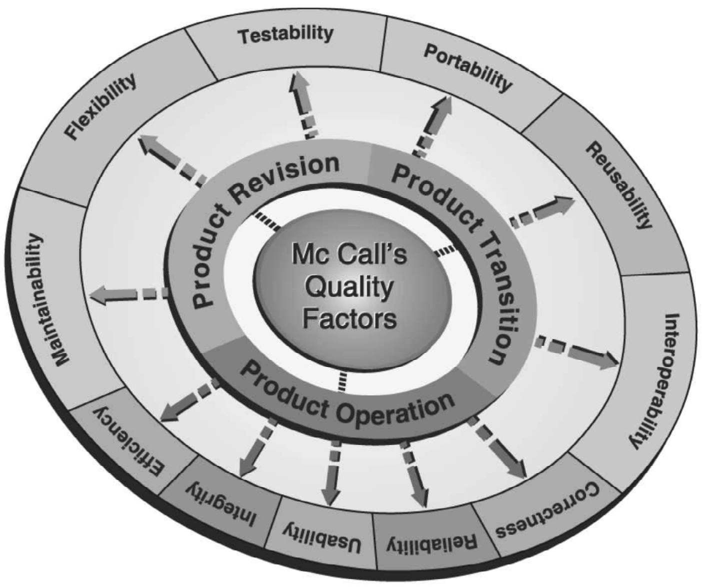


Fig. 5.36 *Mc Call's Quality Factors*

5.8 DESIGN REVIEWS

Software design reviews are well-documented, comprehensive, and systematic examinations of a design used to evaluate the adequacy of the design requirements, to evaluate the capability of the design to meet these requirements, and to identify problems. IEEE defines software design review as '*a formal meeting at which a system's preliminary or detailed design is presented to the user, customer, or other interested parties for comment and approval'*. These reviews are held at the end of the design phase to resolve issues (if any) related to software-related design decisions, architectural design, and detailed design (component-level and interface design) of the entire software or a part of it (such as a database).

Software design reviews should include examination of development plans, requirements specifications, design specifications, testing plans and procedures, all other documents and activities associated with the project, verification results from each stage of the defined life cycle, and validation results for the overall computer-based system. Note that design reviews are considered as the best mechanism to ensure product quality and to reduce the potential risk of avoiding the problems of not meeting the schedules and requirements.

Types of Software Design Reviews

Generally, the review process is carried out in three steps, which corresponds to the steps involved in the software design process (see Figure 5.37). First, **a preliminary design review** is conducted with the customers and users to ensure that the conceptual design (which gives an idea to the user of what the system will look like) satisfies their requirements. Next, **a critical design review** is conducted with analysts and other developers to check the technical design (which is used by the developers to specify *how* the system will work) in order to critically evaluate technical merits of the design. Next, **a program design review** is conducted with the programmers in order to get feedback before the design is implemented.

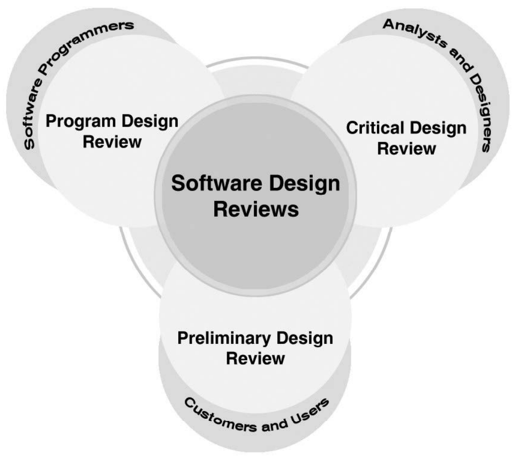


Fig. 5.37 *Types of Design Reviews*

Preliminary design review

The preliminary design review is a formal inspection of the high-level architectural design of the software, which is conducted to verify that the design satisfies the functional and non-functional requirements and conforms to the requirements specified by the users. The purpose is to:

- Ensure that software requirements are reflected in the software architecture
- Specify whether effective modularity is achieved or not
- Define interfaces for modules and external system elements
- Ensure that the data structure is consistent with information domain
- Ensure that maintainability has been considered
- Assess the quality factors

In this review, it is verified that the proposed design includes the required hardware and interfaces with the other parts of the computer-based system. To conduct a preliminary design review, a review team is formed where each review team member acts as an independent person authorized to make necessary comments and decisions. This review team comprises the following individuals:

- **Customers**: Responsible for defining the software's requirements.
- **Moderator**: Presides over the review. The moderator encourages discussions, maintains the main objective throughout the review, settles disputes and gives unbiased observations. In short, he is responsible for the smooth functioning of the review.
- **Secretary**: A silent observer who does not take part in the review process but records the main points of the review.
- **System designers**: These include persons involved in designing of not only the software but also the entire computer-based system.
- **Other stakeholders (developers) not involved in the project**: These people provide an outsider's idea on the proposed design. This is beneficial as they can infuse 'fresh' ideas, address issues of correctness, consistency, and good design practice.

If discrepancies are noted in the review process then the faults are assessed on the basis of their severity. That is, if there is a minor fault it is resolved by the review team. However, if there is a major fault, the review team may agree to revise the proposed conceptual design. Note that preliminary design review is again conducted to assess the effectiveness of the revised (new) design.

Critical design review

Once the preliminary design review is successfully completed and the customer(s) is satisfied with the proposed design, a critical design review is conducted. The purpose of this review is to:

- Ensure that the conceptual and technical designs are free of defects
- Determine that the design under review satisfies the design requirements established in the architectural design specifications
- Critically assess the functionality and maturity of the design
- Justify the design to outsiders so that the technical design is more clear, effective and easy to understand

In this review, diagrams and data (sometimes both) are used to evaluate alternative design strategies and how and why the major design decisions have been taken. Just like for the preliminary design review, a review team is formed to carry out a critical design review. In addition to the team members involved in the preliminary design review, this team comprises the following individuals:

- **System tester**: Who understands the technical issues of design and compares them with the design created for similar projects.
- **Analyst**: Who is responsible for writing system documentation.
- **Program designers for this project**: Who understands the design in order to derive detailed program designs.

Note: This review team does not involve customers.

Similar to a preliminary design review, if discrepancies are noted in the critical design review process, the faults are assessed on the basis of their severity. A minor fault is resolved by the review team. If there is a major fault, the review team may agree to revise the proposed technical design. Note that a critical design review is conducted again to assess the effectiveness of the revised (new) design.

Program design review

On successful completion of the critical design review, a program design review is conducted to get feedback on the designs before implementation (coding) begins. This review is conducted between the designers and developers in order to:

- Ensure that the detailed design is feasible
- Ensure that the interface is consistent with the architectural design
- Specify whether the design is amenable to implementation language
- Ensure that structured programming constructs are used throughout
- Ensures that the implementation team is able to understand the proposed design

To conduct a program design review, a review team is formed, comprising system designers, a system tester, moderator, secretary, and analyst. The review team also includes program designers and developers. The program designers, after completing the program designs, present their plans to a team of other designers, analysts and programmers for comments and suggestions. Note that a successful program design review presents considerations relating to coding plans before coding begins.

Process of Software Design Review

Design reviews are considered important as in these reviews the product is logically viewed as the collection of various entities/components and use cases. These reviews occur at all levels of the software design, and encompass through all parts of the software units. Generally, the review process comprises three criteria, as listed below:

- **Entry criteria**: Software design is ready for review.
- **Activities**: This criteria involves the following steps:
  1. Select software design review team participants, assign roles and prepare schedule for the review.
  2. Distribute the software design review package to all the reviewing participants.
  3. Participants assess compliance with requirements, completeness, efficiency and adherence to design methodology. Participants identify and discuss defects. The recorder documents defects, action items, and recommendations.
  4. The software design team corrects any defects in design and updates design review material as required.
  5. The software development manager obtains the approval of the software design from the software project manager.
- **Exit criteria**: The software design is approved.

Evaluation of Software Design Reviews

The software design review process is beneficial for everyone as the faults can be detected at an earlier stage thereby reducing the cost of detecting errors and reducing the likelihood of missing a critical issue. Every review team member examines the integrity of the design and not the persons involved in it (that is, designers), which in turn emphasizes that the common objective of developing a highly rated design is achieved. To check the effectiveness of the design, the review team members should address the following questions:

- Is the solution achieved with the developed design?
- Is the design reusable?
- Is the design well structured, and easy to understand?
- Is the design compatible with other platforms?
- Is it easy to modify or enlarge the design?
- Is the design well documented?
- Does the design use suitable techniques in order to handle faults and prevent failures?
- Does the design reuse component from other projects, wherever necessary?

In addition to these questions, if the proposed system is developed using a phased development (like waterfall and incremental model), then the phases should be interfaced sufficiently so that an easy transition can take place from one phase to the other.

5.8.1 Design Quality Metrics

The aim of a software developer is to develop high-quality software within a specified time and budget. To achieve this, software should be developed according to functional and performance requirements, document development standards and characteristics expected from professionally developed software. Note that private metrics are collected by software engineers and then assimilated to achieve project-level measures. The main aim at the project-level is to measure both errors and defects. These measures are used to derive metrics, which provide an insight into the efficacy of both individual and group software quality assurance and the software control activities.

Many measures have been proposed for assessing software quality, such as interoperability, functionality, and so on. However, it has been observed that reliability, correctness, maintainability, integrity and usability are most useful as they provide valuable indicators to the project team.

- **Reliability:** The system or software should be able to maintain its performance level under given conditions. Reliability can be defined as the ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations. Reliability can be measured using *Mean Time Between Failure* (*MTBF*), which is the average of time between successive failures. A similar measure to MTBF is *Mean Time To Repair* (*MTTR*), which is the average time taken to repair a machine after a failure occurs.

MTBF can be combined with *Mean Time To Failure* (*MTTF*), which describes how long a software can be used to calculate MTBF, i.e.,

MTBF = MTTF + MTTR

- **Correctness:** A system or software must function correctly. Correctness can be defined as the degree to which any software performs its specified function. It can be measured in terms of defects per KDLOC (thousand lines of code). For quality assessment, defects are counted over a specified period of time.
- **Maintainability:** In software engineering, software maintenance is one of the most expensive and time-consuming activities. Maintainability can be defined as the ease with which a software product can be modified to correct errors, to meet new requirements, to make future maintenance easier or adapt to the changed environment. Note that software maintainability is assessed by using indirect measures such as *Mean Time To Change* (*MTTC*), which can be defined as the time taken to analyse change requests, design modifications, implement changes, testing and distribute changes to all users. Generally, it has been observed that programs having lower MTTC are easier to maintain.
- **Integrity:** In the age of cyber-terrorism and hacking, software integrity has become an important factor in software development. Software integrity can be defined as degree to which unauthorized access to the components of software (program, data, and documents) can be controlled.

  For measuring the integrity of a software, attributes such as threat and security are used. *Threat* can be defined as the probability of a particular attack at a given point of time. *Security* is the probability of repelling an attack, if it occurs. Using these two attributes, integrity can be calculated by using the following equation:

  ```
  Integrity  = S(1 – [threat*{1 – security}])
  ```
- **Usability:** Software which is easy to understand and easy to use is always preferred by the user. Usability can be defined as the capability of a software to be understood, learned and used under specified conditions. Note that a software that accomplishes all user requirements but is not easy to use is often destined to fail.

In addition to the aforementioned measures, lack of conformance to software requirements should be avoided as this forms the basis of measuring software quality. Also, in order to achieve high quality, both explicit and implicit requirements should be considered.

Defect Removal Efficiency (DRE)

Defect removal efficiency (DRE) can be defined as a quality metrics that is beneficial at both the project level and the process level. Quality assurance and control activities that are applied throughout the software development process are responsible for detecting errors introduced at various phases of SDLC. The ability to detect errors (filtering abilities) is measured with the help of DRE, which can be calculated by using the following equation:

$$\text{DRE} = \text{E}/(\text{E} + \text{D})$$

where:

E = number of errors found before software is delivered to the user

D = number of defects found after software is delivered to the user

The value of DRE approaches 1, if there are no defects in the software. As the value of E increases for a given value of D, the overall value of DRE starts to approach 1. With an increase in the value of E, the value of D decreases as more errors are discovered before the software is delivered to the user. DRE improves the quality of a software by establishing methods which detect maximum number of errors before the software is delivered to the user.

DRE can also be used at different phases of the software development process. It is used to assess the software team's ability to find errors at one phase before they are passed on to the next development phase. When DRE is defined in the context of SDLC phases, it can be calculated as:

$$\text{DRE}_i = \text{E}_i/(\text{E}_i + \text{E}_{i+1})$$

where:

E_i = Number of errors found in phase *i*.

E_{i+1} = Number of errors that were ignored in phase *i*, but found in phase *i* + 1.

The objective of the software team is to achieve the value of DRE_i as 1. In other words, errors should be removed before they are passed on to the next phase.

5.8.2 User Interface Evaluation and Issues

While designing a user interface, there are certain issues that must be considered. Generally, the software designers refer to these issues lately in the design process, which often results in project delays or customers' dissatisfaction. Therefore, it is recommended to consider these issues when the design process starts. Various design issues of user interface are listed below.

Response Time

The response time of a system is the time from when the user issues some request to the time the system responds to that request. The response time should be as low as possible, since longer response time results in user frustration. In addition, the response time for any specific request should not vary significantly. If this happens, the user might think that something different has occurred. For example, it is not acceptable that a request sometimes responds in 0.5 seconds and sometimes in 2.5 seconds. Instead, an average response time of 1 second is acceptable in this case.

Online Help System

Modern software applications provide an online help system that helps the users to learn about the operations of any software at any time while using the software. In case the users forget a command or are unaware of some features of the software, they may use the online help system. The online help system must keep track of what the user is doing while allowing the user to generate help request so that it can provide help in the context of the user's actions. In addition, various other features that an online help system may incorporate in its design are:

- It may take advantage of graphics and animation characteristics while providing help to the users rather than representing just instructions.
- It may include help options for all or a subset of functions performed by the software.
- It may be used by the user through a help menu or a help command, or a special function key.
- It may use either a separate window or a reference to printed document or a fixed screen location in the same window for representing help messages.

Error Management

Sometimes, during processing, errors occur due to some exceptional conditions such as memory constraints and failure of communication link. The system represents these errors to the user using error messages. These error messages are either unambiguous or contain only general information, such as invalid input and system error, which hardly provides any guidance to the user to correct errors. Thus, any error message or warning produced by the system should have the following characteristics:

- It should contain proper suggestions to recover from the errors. The user may be asked to use the online help system to find out more about the error situation.
- It should be represented in a language that is easily understandable by the user and should use polite words.
- It should mention the negative effects (if any) caused by the error encountered. For example, a file that is corrupted because of that error.
- It should be associated with any audio or visual clue. For example, whenever an error occurs, the system may generate a beep or change the text colour in order to indicate the occurrence of an error.

In general, users do not like errors irrespective of the fact how well designed it is. Thus, a good user interface should minimize the scope of committing errors. To manage the errors or to reduce their possibilities, various procedures such as consistency of names, issue procedures and behaviour of similar commands can be used.

5.9 VERIFICATION AND VALIDATION

Verification refers to the process of ensuring that the software is developed according to its specifications. For verification, techniques such as reviews, analysis, inspections and walkthroughs are commonly used. IEEE defines verification as 'a process for determining whether the software products of an activity fulfill the requirements or conditions imposed on them in the previous activities.' Thus, verification confirms that the product is transformed from one form to another as intended and with sufficient accuracy. Validation refers to the process of checking that the developed software meets the requirements specified by the user. IEEE defines validation as 'a process for determining whether the requirements and the final, as-built system or software product fulfils its specific intended use.' Thus, validation substantiates the software functions with sufficient accuracy with respect to its requirements specification. Verification and validation can be thus summarized as:

Verification: Is the software being developed in the right way?

Validation: Is the right software being developed?

5.9.1 Output Interface

Before graphic-user interfaces, the command language-based interfaces or menu-based interfaces were used for every type of application. The command-language-based interface provides a set of primitive commands and the user can type an appropriate command to perform a particular function. Thus, the user is required to memorize a number of commands and type them in order to perform operations. On the other hand, in a menu-based interface, the user can select from a list of options (called *menu*) using some pointing device such as mouse. Thus, the user is relieved from the burden of remembering and typing commands. Several issues must be taken into consideration while designing a command-language-based or a menu-based user interface:

- Which menu options should be associated with a shortcut key?
- Which method is required for the user to execute any command? Several methods include pressing function keys, pressing a pair of control keys (such as Alt+D or Ctrl+S) or typing a word.
- How easy is it for the user to learn the commands and shortcut keys?
- Can a user assign a new shortcut key to any command or menu option?
- Are the menu labels self-explanatory?

Check Your Progress

7. Define user interface design.
8. What is decision table?
9. State about the object oriented design.
10. Elaborate on the term software quality assurance.
11. Give the definition of software design review according to IEEE.
12. State about the menu-based interfaces.

5.10 ANSWER TO 'CHECK YOUR PROGRESS'

1. Software design is a software engineering phase in which a blueprint is developed which serves as the base for constructing the software system. IEEE defines software design as 'both a process of defining the architecture,

components, interfaces, and other characteristics of a system or component and the result of that process.'

2. Software architecture refers to the structure of the system, which is composed of the various components of a program/system, the attributes (properties) of those components and the relationship amongst them.
3. IEEE defines architectural design as 'the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system'.
4. Procedural design is created by transforming the structural elements defined by the software architecture into procedural descriptions of software components. These components are derived from the analysis model where the data-flow-oriented element (present in the analysis model) serves as the base for the derivation.
5. Stepwise refinement decomposes the system design from a high level of abstraction into a more detailed level (lower level) of abstraction. At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data. As we proceed towards the lower levels of abstraction, more and more details are available.
6. IEEE defines program design language as, 'A specification language with special constructs and sometimes verification protocols used to develop, analyse and document a program design.'
7. User interfaces determine the way in which users interact with the software. The user interface design creates an effective communication medium between a human and a computing machine. It provides easy and intuitive access to information as well as efficient interaction and control of software functionality.
8. A decision table can be defined as a table that contains the values for all logical expressions pertaining to a certain problem (the conditions) and the corresponding logical outcome using logical rules that connect conditions with results.
9. Object-oriented design (OOD) transforms the analysis model created using the object-oriented analysis (OOA) into a design model of a system that serves as a basis for object-oriented implementation.
10. Software quality assurance is concerned with process quality and refers to planned and systematic set of activities that ensure that software life cycle processes and products conform to requirements, standards, and procedures.
11. IEEE defines software design review as 'a formal meeting at which a system's preliminary or detailed design is presented to the user, customer, or other interested parties for comment and approval'.
12. Graphic-user interfaces, the command language-based interfaces or menu-based interfaces were used for every type of application. The command-language-based interface provides a set of primitive commands and the user can type an appropriate command to perform a particular function.

NOTES

Thus, the user is required to memorize a number of commands and type them in order to perform operations. On the other hand, in a menu-based interface, the user can select from a list of options (called menu) using some pointing device such as mouse.

5.11 SUMMARY

- Software design is defined as both a process of defining the architecture, components, interfaces and other characteristics of a system or component and the result of that process.
- Software design principles act as a framework for the designers to follow a good design practice.
- There are various software design concepts, which lay a foundation for the software design process.
- Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components.
- Software architecture refers to the structure of the components of a program/ system, their interrelationships and guidelines governing their design and evolution over time.
- Architectural design specifies the relationship between the structural elements of a software, design patterns, architectural styles and the factors affecting the way in which architecture can be implemented.
- A procedural design converts the structural elements of software architecture into a procedural description of software components.
- Function-oriented design is an approach that is based on the decomposition of design into a set of interacting modules with each module having a specific function. For decomposing the design, a top-down design strategy called stepwise refinement is used.
- IEEE defines program design language as, 'A specification language with special constructs and sometimes verification protocols used to develop, analyse and document a program design.'
- Software quality refers to the extent to which a software is able to perform correctly as well as meet the required features and functions.
- IEEE defines program design language as, 'A specification language with special constructs and sometimes verification protocols used to develop, analyse and document a program design.'
- PDL, also known as structured English or pseudocode is a language that is used to specify the system design using natural language (English) along with the constructs of a programming language.
- The benefit of using PDL as a design notation is that it requires the programmer to focus only on the algorithms without taking care of the syntax of the programming language.

- User interfaces determine the way in which users interact with the software. The user interface design creates an effective communication medium between a human and a computing machine. It provides easy and intuitive access to information as well as efficient interaction and control of software functionality.
- To represent software design, design notations are used. These notations are important as they help the designers to represent modules, interfaces, hidden information, concurrency, message passing, invocation of operations and the overall program structure in a comprehensive manner.
- The various notations that are commonly used are flowcharts, data-flow diagrams, structure charts, HIPO diagram, and decision table and program design language.
- A flowchart is a graphical representation of an algorithm where a sequence of steps are drawn in the form of different shapes of boxes and the logical flow is indicated by inter-connecting arrows.
- Structured flowcharts differ from traditional flowcharts in that the former are limited to compositions of only certain basic forms.
- Structure charts are used to specify the high-level design, or architecture, of a computer program. As a design notation, structure charts show the control and data flow between modules.
- HIPO (Hierarchy-Input-Process-Output) diagrams were developed by IBM in the 1970s as a design notation for representing system design. These diagrams further became an excellent input for detailed program design.
- Object-oriented design (OOD) transforms the analysis model created using the object-oriented analysis (OOA) into a design model of a system that serves as a basis for object-oriented implementation.
- Software quality assurance is concerned with process quality and refers to planned and systematic set of activities that ensure that software life cycle processes and products conform to requirements, standards, and procedures.
- Software design reviews are well-documented, comprehensive, and systematic examinations of a design used to evaluate the adequacy of the design requirements, to evaluate the capability of the design to meet these requirements, and to identify problems.
- Graphic-user interfaces, the command language-based interfaces or menu-based interfaces were used for every type of application. The command-language-based interface provides a set of primitive commands and the user can type an appropriate command to perform a particular function.

NOTES

5.12 KEY TERMS

- **Software design:** It is a software engineering phase in which a blueprint is developed which serves as the base for constructing the software system.
- **Functional independence:** The refined form of the design concepts of modularity, abstraction and information hiding.

NOTES

- **Coupling:** A measure of relative interconnection among modules.
- **Cohesion:** Measure of strength of the association of elements within a module.
- **Temporal cohesion:** Elements within the modules contain unrelated activities that can be carried out at the same time.
- **Software quality:** The extent to which a software is able to perform correctly as well as meet the required features and functions.
- **Stepwise refinement:** A top-down design strategy used for decomposing a system from a high level of abstraction into a more detailed level of abstraction.
- **Programming design language**: Also known as structured English or pseudocode, it is a language that is used to specify the system design using natural language (English) along with the constructs of a programming language.
- **Structure chart:** Graphical representation of the high-level design or architecture of computer program.
- **Prototyping:** An approach that helps others to better evaluate the final product and understand the weaknesses, restrictions and strengths of the product design.
- **Flowchart:** A graphical representation of an algorithm where a sequence of steps is drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows.
- **Decision table:** A table that contains all the possible conditions for a specific problem and the corresponding results using condition rules that connect conditions with results.
- **Software quality assurance:** It is usually called SQA and comprises various tasks that are responsible for ensuring quality.

5.13 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is abstraction?
2. Define the term patterns.
3. List the advantages of the object-oriented architecture style.
4. State about the data-flow architecture.
5. Elaborate on the term design decomposition.
6. Why is programming design language (PDL) necessary?
7. Mention some techniques for evaluating user interface.
8. State about the structure charts.
9. What is concurrency system design?

10. Write the objectives of software quality assurance.
11. State about the user interface evaluation and issues.
12. Differentiate between verification and validation.

Long-Answer Question

1. Discuss the various principles of a software design.
2. What are the various models used to represent architectural design?
3. Briefly explain the architectural design giving appropriate examples.
4. Explain module coupling and cohesion with the help of suitable diagrams.
5. Describe the software design quality giving appropriate examples.
6. Explain the use of PDL in design specifications.
7. How can one test the user interface? Discuss the merits and demerits of user interface rules.
8. Explain different software design notations used to represent software design.
9. What is the difference between structure charts and structured flow charts? Explain by giving suitable examples and supporting diagrams.
10. Briefly explain the object-oriented design with the help of appropriate examples.
11. Why are software design reviews used? Explain the process of carrying out software design reviews.
12. Discuss about the design quality metrics giving appropriate examples.

5.14 FURTHER READING

Mali, Rajib. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hall of India.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Education.

Jalote, Pankaj. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Sommerville, Ian. *Software Engineering*. New Delhi: Addison Wesley.

Fenton, N.E. and F.L. Pfleeger. *Software Metrics*. Mumbai: Thomson Learning.