

SOFTWARE ENGINEERING

BCA - 303



Dr. N.P. Singh
Dean

Directorate of Distance Education

ਪੰਜਾਬ ਟੈਕਨੀਕਲ ਯੂਨੀਵਰਸਿਟੀ ਜਲੰਧਰ

PTU Punjab
Technical
University

Estd. Under Punjab Technical University Act, 1996
(Punjab Act No. 1 of 1997)

Preface

The Punjab Government established Punjab Technical University (PTU) in 1997 by an act of State Legislative. The University was entrusted with the responsibility of developing the new generation of technical manpower that can spearhead the industrial development of the State. Punjab Technical University has been envisaged to be the grooming ground for the future Engineers, Managers and Researchers.

As of today, PTU affiliates more than 300 Engineering, Management, Pharmacy, Hotel Management and Architecture institutions in the State that are approved by All India Council of Technical Education (AICTE).

PTU understands that restricting technical education to its campuses will not serve its objective of effective spreading of knowledge in the society. It is firmly understood that latest technical education has to be spread to the masses in every corner of the nation. This is how the Distance Education Programme (DEP) of the Punjab Technical University was conceived.

The objectives of the programme are to impart affordable, relevant, skill-based & remunerative technical education to the masses in the different corner of the country.

Today, the University has more than 2000 Learning Centres spread across the country offering quality technical education in the fields of Information Technology and Management, Paramedical Technology, Fashion Technology, Hotel Management and Tourism, Media and Mass Communication and Journalism etc.

The main purpose of this book is to impart the student an insight into the subject, explaining the complexities involved, in a simplified manner and helping them to achieve their academic goals.

For an easier navigation and understanding, this book contains the complete PTU curriculum of this subject and the topics. The various topics are dividing into Chapters, Units & Sub-Units and sufficient space is provided for students to make their brief notes.

This book encompasses a global approach for providing the simplified study material to both working as well as non-working students and is certain to get benefitted from the efforts of the authors of this book.

Dr.N.P. Singh
Dean (Distance Education Programme)

eINDIA 2010
AWARDS
Recognising excellence in ICT

Award of the Year

ICT Enabled
University of the Year

Open & Distance Learning
Initiative of the Year

“Propelling Punjab to a Prosperous Knowledge Society”

Punjab Technical University Jalandhar

Jalandhar-Kapurthala Highway, Near Science City, Kapurthala- 144601

Phones : 01822-662502 Fax : 01822-255532

Website : www.ptu.ac.in email : deandep@ptu.ac.in, singhnp59@gmail.com

SOFTWARE ENGINEERING

BCA - 303

This SIM has been prepared exclusively under the guidance of Punjab Technical University (PTU) and reviewed by experts and approved by the concerned statutory Board of Studies (BOS). It conforms to the syllabi and contents as approved by the BOS of PTU.

Reviewer

Dr. N. Ch. S.N. Iyengar	Senior Professor, School of Computing Sciences, VIT University, Vellore
--------------------------------	--

Author: Rohit Khurana

Copyright © ITL Education Solutions Ltd., 2006

Reprint 2010

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

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



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: 576, Masjid Road, Jangpura, New Delhi 110 014

• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

CAREER OPPORTUNITIES

Software engineers who work in applications or systems development are engaged in analyzing user needs and designing, constructing, testing, and maintaining computer applications software or systems. These engineers are also geared to tackle technical problems and hitches.

Computer software engineers, who develop a finished product ready to release, are usually a part of the mega team that designs and works on advanced hardware, software, and systems and that includes workers from various fields like engineering, marketing, production and design. They work for companies that need configuration, implementation, and installation of complete computer systems. These engineers may also be part of the marketing or sales staff, and serve as the chief technical resource for these sales officers, staff, as well as customers. They may even engage in product sales and provide continued technical support to the buyers and consumers.

PTU DEP SYLLABI-BOOK MAPPING TABLE

BCA - 303 Software Engineering

Syllabi

Mapping in Book

Section-I

Software: Characteristics, Components Applications, Software Process Models: Waterfall, Spiral, Prototyping, Fourth Generation Techniques, Concepts of Project Management, Role of Metrics And Measurement.

Unit 1: Software Process and Process Models
(Pages 3-31)

Section-II

S/W Project Planning: Objectives, Decomposition Techniques: S/W Sizing, Problem Based Estimation, Process Based Estimation, Cost Estimation Models: COCOMO Model, The S/W Equation, System Analysis: Principles of Structured Analysis, Requirement Analysis, DFD, Entity Relationship Diagram, Data Dictionary.

Unit 2: Software Project Planning & Cost Estimation
(Pages 33-63);
Unit 3: Systems Analysis (Pages 65-111);

Section-III

S/W Design: Objectives, Principles, Concepts, Design Mythologies: Data Design, Architecture Design, Procedural Design, Object – Oriented Concepts, Testing Fundamentals: Objectives, Principles, Testability, Test Cases: White Box & Black Box Testing, Testing Strategies: Verification & Validation, Unit Test, Integration Testing, Validation Testing, System Testing.

Unit 4: Software Design
(Pages 113-155)
Unit 5: Software Testing
(Pages 157-206)

CONTENTS

INTRODUCTION	1
UNIT 1: SOFTWARE PROCESS AND PROCESS MODELS	3–31
1.0 Introduction; 1.1 Unit Objectives	
1.2 Software; 1.2.1 History of Software Development; 1.2.2 Software Characteristics; 1.2.3 Software Components; 1.2.4 Evolution of Software Engineering; 1.2.5 Evolution of Software Engineering	
1.3 Software Engineering: Definition 1.3.1 Software Engineer	
1.4 Phases in Software Engineering 1.4.1 Preliminary Investigation; 1.4.2 Case Study: Bridge and Software Development	
1.5 Software Project Management 1.5.1 Process Management and Product Engineering Process; 1.5.2 Process Framework	
1.6 Process Models 1.6.1 Waterfall Model; 1.6.2 Prototyping Model; 1.6.3 Spiral Model; 1.6.4 Fourth Generation Techniques (4GT)	
1.7 Role of Software Metrics and Measurement 1.7.1 Software Measurement; 1.7.2 Software Metrics	
1.8 Let us Summarize; 1.9 Answers to ‘Check Your Progress’	
1.10 Questions and Exercises; 1.11 Further Reading	
UNIT 2: SOFTWARE PROJECT PLANNING AND COST ESTIMATION	33–63
2.0 Introduction; 2.1 Unit Objectives	
2.2 Project Planning 2.2.1 Project Purpose; 2.2.2 Project Scope; 2.2.3 Project Planning Process; 2.2.4 Project Plan	
2.3 Project Scheduling	
2.4 Basics of Cost Estimation 2.4.1 Resources for Software Cost Estimation; 2.4.2 Software Product Cost Factors	
2.5 Software Cost Estimation Process	
2.6 Decomposition Techniques 2.6.1 Problem-based Estimation; 2.6.2 Process-based Estimation	
2.7 Cost Estimation Models 2.7.1 Constructive Cost Model; 2.7.2 Software Equation	
2.8 Let us Summarize; 2.9 Answers to ‘Check Your Progress’	
2.10 Questions and Exercises; 2.11 Further Reading	
UNIT 3: SYSTEM ANALYSIS	65–111
3.0 Introduction; 3.1 Unit Objectives	
3.2 What is Software Requirement? 3.2.1 Guidelines for Expressing Requirements; 3.2.2 Types of Requirements; 3.2.3 Requirements Engineering Process	
3.3 Feasibility Study 3.3.1 Types of Feasibility; 3.3.2 Feasibility Study Process	
3.4 Requirements Elicitation 3.4.1 Elicitation Techniques	
3.5 Requirements Analysis 3.5.1 Structured Analysis; 3.5.2 Object-oriented Modelling; 3.5.3 Other Approaches	

- 3.6 Requirements Specification
 - 3.6.1 Structure of SRS
- 3.7 Requirements Validation
 - 3.7.1 Requirement Review; 3.7.2 Other Requirement Validation Techniques
- 3.8 Requirements Management
 - 3.8.1 Requirements Management Process; 3.8.2 Requirements Change Management
- 3.9 Case Study: Student Admission and Examination System
 - 3.9.1 Problem Statement; 3.9.2 Data Flow Diagrams;
 - 3.9.3 Entity Relationship Diagram; 3.9.4 Software Requirements Specification Document
- 3.10 Data Dictionary; 3.11 Let us Summarize
- 3.12 Answers to 'Check Your Progress'; 3.13 Questions and Exercises
- 3.14 Further Reading

UNIT 4: SOFTWARE DESIGN

113–155

- 4.0 Introduction; 4.1 Unit Objectives
- 4.2 Basics of Software Design
 - 4.2.1 Principles of Software Design; 4.2.2 Software Design Concepts;
 - 4.2.3 Developing a Design Model
- 4.3 Data Design
- 4.4 Architectural Design
 - 4.4.1 Architectural Design Representation; 4.4.2 Architectural Styles
- 4.5 Procedural Design
 - 4.5.1 Functional Independence
- 4.6 User Interface Design
 - 4.6.1 User Interface Rules; 4.6.2 User Interface Design Process
 - 4.6.3 Evaluating User Interface Design
- 4.7 Software Design Notation
- 4.8 Software Design Reviews
 - 4.8.1 Types of Software Design Reviews; 4.8.2 Software Design Review Process;
 - 4.8.3 Evaluating Software Design Reviews
- 4.9 Software Design Documentation (SDD)
- 4.10 Case Study: Higher Education Online Library System
 - 4.10.1 Data Design; 4.10.2 Architectural Design; 4.10.3 Procedural Design;
 - 4.10.4 User Interface Design
- 4.11 Object-oriented Concepts; 4.12 Let us Summarize
- 4.13 Answers to 'Check Your Progress'; 4.14 Questions and Exercises
- 4.15 Further Reading

UNIT 5: SOFTWARE TESTING

157–206

- 5.0 Introduction; 5.1 Unit Objectives
- 5.2 Software Testing Basics
 - 5.2.1 Principles of Software Testing; 5.2.2 Testability; 5.2.3 Characteristics of Software Test
- 5.3 Test Plan; 5.4 Test Case Design;
- 5.5 Software Testing Strategies
 - 5.5.1 Unit Testing; 5.5.2 Integration Testing; 5.5.3 System Testing; 5.5.4 Validation Testing
- 5.6 Testing Techniques
 - 5.6.1 White Box Testing; 5.6.2 Black Box Testing;
 - 5.6.3 Difference between White Box and Black Box Testing; 5.6.4 Gray Box Testing
- 5.7 Object-oriented Testing
 - 5.7.1 Testing of Classes; 5.7.2 Developing Test Cases in Object-oriented Testing
 - 5.7.3 Object-oriented Testing Methods
- 5.9 Let us Summarize; 5.10 Answers to 'Check Your Progress'
- 5.11 Questions and Exercises; 5.12 Further Reading

INTRODUCTION

Software as a technology is changing the face of the world and is the driving force behind many aspects of business, science and engineering. As computing systems become more powerful and complex the need of systematic approaches in software development has become inevitable. *Software engineering* provides methods and tools to deal with the complexities involved in software development and enables us to develop a high-quality, reliable, maintainable, and error free software that satisfies customers' requirements.

Since the coining of the word, software engineering has evolved from an obscure idea practised by relatively small number of people to a full-fledged engineering discipline. Today, it is accepted as a subject that involves intensive research and diligent study. Universities the world over have incorporated software engineering as an integral part of Computer Science, Computer Application, and Information Technology curricula.

This book provides an in-depth coverage of fundamental principles, methods and applications of software engineering and meets the requirements of software engineering students enrolled in MCA course. The text is presented in simple, concise, and easy-to-understand language. Also, the subject matter is well supported by examples, tables, diagrams, and case studies to make topics clear and understandable.

The text in this book has been organized into five units:

Unit 1 introduces software and software engineering concepts. Also, the unit deals with the various software process models used to develop software systems.

Unit 2 provides the foundation to learn project planning process and helps to understand the factors influencing the cost of developing a software product.

Unit 3 helps to understand the requirement process and how software requirement specification lays foundation for other software engineering activities.

Unit 4 helps to understand the various design elements in the design model and the various design notations used to represent software design.

Unit 5 discusses the basics of software testing, software testing strategies and various testing techniques.

For any suggestions and comments about this book, please feel free to send your feedback at feedback@itlesl.com

Hope you enjoy reading this book as much as we have enjoyed writing it.

UNIT 1 SOFTWARE PROCESS AND PROCESS MODELS

NOTES

Structure

- 1.0 Introduction
- 1.1 Unit Objectives
- 1.2 Software
 - 1.2.1 History of Software Development; 1.2.2 Software Characteristics
 - 1.2.3 Software Components; 1.2.4 Evolution of Software Engineering;
 - 1.2.5 Evolution of Software Engineering
- 1.3 Software Engineering: Definition
 - 1.3.1 Software Engineer
- 1.4 Phases in Software Engineering
 - 1.4.1 Preliminary Investigation; 1.4.2 Case Study: Bridge and Software Development
- 1.5 Software Project Management
 - 1.5.1 Process Management and Product Engineering Process
 - 1.5.2 Process Framework
- 1.6 Process Models
 - 1.6.1 Waterfall Model; 1.6.2 Prototyping Model
 - 1.6.3 Spiral Model; 1.6.4 Fourth Generation Techniques (4GT)
- 1.7 Role of Software Metrics and Measurement
 - 1.7.1 Software Measurement; 1.7.2 Software Metrics
- 1.8 Let us Summarize
- 1.9 Answers to 'Check Your Progress'
- 1.10 Questions and Exercises
- 1.11 Further Reading

1.0 INTRODUCTION

Software systems have become ubiquitous. These systems are now virtually present in all electronic and electric equipments. Be it electronic gizmos and gadgets, traffic lights, medical equipments—almost all electrical equipments are run by software. Software is an intangible entity that embodies instructions and programs, which drives the actual functioning of a computer system.

In the early days of computers, the computer memory was small, its language consisted of binary and machine code, and programmers used to develop code that could be used in developing more than one software system. Thus, the developed software was simple in nature and did not involve much creativity from the developers' end. However, as technology improved, there was a need to build bigger and complex software systems in order to meet the users' changing and growing requirements. This led to emergence of software engineering which included development of software processes and various process models. Software process helps in developing a timely, high-quality, and highly efficient product or system. It consists of 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 many different people.

In this chapter we focus on what is software, how software engineering evolved, why process models are used, and why software metrics and measurement are used.

NOTES

1.1 UNIT OBJECTIVES

After reading this unit, the reader will understand:

- History of software development.
- Software characteristics and classification of software.
- Various software myths, such as management myths, user myths, and developer myths.
- Software crisis, which has been used since the early days of software engineering to describe the impact of the rapid increases in computer power and its complexity.
- What is software engineering?
- The role of software engineer.
- Phased development of software, which is often referred to as software development life cycle.
- What is software process, project, and product?
- The major components of software process, which helps in developing a product that accomplishes user requirements.
- How process framework determines the processes that are essential for completing a complex software project.
- The need for process assessment to ensure that it meets a set of basic process criteria.
- Various process models, which comprises of processes, methods, steps for developing software.
- The role of software metrics and measurement.

1.2 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 of IEEE, software is not just programs, but 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 to do and how to do it. 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, such as C, C++, and so on. 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.

1.2.1 History of Software Development

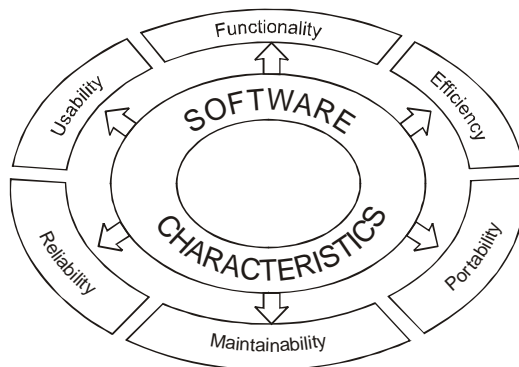
Software development came into existence in later half of 1940s, when the first stored-program computer was created at Cambridge EDSAC. Earlier the programs were created as binary machine instructions. This approach of programming was considered slow and cumbersome, as it was difficult for humans to memorise long and complex 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 late 1950s, managing program even with the aid of reusable subroutines was becoming uneconomical. Hence, research in the area of automatic programming began. Automatic programming allowed programmer to write program 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 for specific areas and was being marketed and sold separately from hardware. This marked a deviation from earlier practices of giving software free as a part of the hardware platform. In addition, hiding of internal details of an operating system using abstract programming interfaces improved the productivity of the programmer.
1970s	With the development of structured design, software development models were introduced. These were based on a more organic, evolutionary approach, deviating from the waterfall-based methodologies of hardware engineering. Research was done on quantitative techniques for software design. During this time, researchers began to focus on software design to address the problems of developing complex software systems.
1980s	Software engineering research shifted focus toward integrating designs and design processes into the larger context of software development process and management. In the latter half of the 1980s a new design paradigm known as object-oriented modelling was introduced. Software engineers using the OOPs technique were able to model both the problems domain and solution domain within the programming languages.
1990s	Object orientation was augmented with design techniques, such as class/responsibilities/collaborators (CRC) cards and use case analysis. Methods and modelling notations from the structured design made their way into the object-oriented modelling methods. This included diagramming techniques, such as state transition diagrams and processing models.
Presently	A multi viewed approach to design is used to manage the complexity of designing and developing large-scale software systems. This multi view approach has resulted in the development of the unified modelling language (UML), which integrates modelling concepts and notations from many methodologies.

NOTES**1.2.2 Software Characteristics**

Different individuals judge software on different basis. This is because they are involved with the software in different ways. For example, 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. These components are listed below:

**Figure 1.1** Characteristics of Software

NOTES

- **Functionality:** Refers to the degree of performance of the software against its intended purpose.
- **Reliability:** Refers to the ability of software to perform a required function under given conditions for a specified period.
- **Usability:** Refers the degree to which software is easy to use.
- **Efficiency:** Refers to the ability of software to use system resource in the most effective and efficient manner.
- **Maintainability:** Refers to the ease with which a software system can be modified to add capabilities, improve system performance, or correct errors.
- **Portability:** 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 characteristics, *robustness* and *integrity* are also considered to be important. Robustness refers to the extent to which software can continue to operate correctly despite the introduction of invalid input, while integrity refers to the extent to which unauthorised access or modification of software or data can be controlled in the computer system.

1.2.3 Software Components

A software component is defined as an independent executable software element with a well-defined set of inputs, outputs and interface. All the services provided by a component are made available through the interface and all the interactions with the component are done through that interface. Council and Heinmann define the term software component as follows.

“A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition.”

Szyperski describes the term as follows.

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

The software engineering that emphasizes the design and development of computer-based systems using software components is called component-based software engineering (CBSE). The main objective of CBSE is to standardize the interfaces between software components so that they can be assembled easily to develop new software. The basic idea behind CBSE is to reuse the existing components. The components developed for a specific application have to be generalized to make them reusable. In other words, the more generalized interface, the greater the reusability. Apart from the advantage of reusability, the components have the following advantages.

- The components are independent and hence, they do not interfere with each other and are easy to modify.
- The inner workings of the components are hidden from the user.
- The components do not have to be compiled prior to their use with other components.
- The communication and interaction with the components is done through well-defined interfaces.
- The component platforms are shared and hence, the development costs are reduced.

There are some characteristics that a software program must possess before it qualifies as a component. These are given below.

- It should be independent and deployable, that is, it has to be a self-contained and stand-alone entity and it should not depend on other software components for its use.
- It should provide some pre-defined interfaces and all interactions must take through these interfaces.
- It should have a complete documentation so that the users of the component can decide whether or not the component is meeting their needs.
- It has to conform to some specified standards.
- It should be language-independent.

NOTES

1.2.3.1 Components Applications

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 below:

- **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 organisations 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 software comprises of payroll, inventory, accounting, and software that permits user to access relevant data from the database. These activities are usually performed with the help of specialised 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,

NOTES

linked with hyperlinks. Web browser is a 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, such as, software testing tools and object modelling tools.
- **Customised 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.
- **Customised COTS:** In this classification, user can enter into a contract with the software vendor to develop a COTS product for a special purpose, that is, software can be customised according to the needs of the user. Another growing trend is the development of COTS software components—components that 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. These types are listed below:

- **Stand-alone Software:** Resides on a single computer and does not interact with any other software installed in a different computer.
- **Embedded Software:** Part of unique application involving hardware like automobile controller.
- **Real-time Software:** Operations execute within very short time limits, often microseconds. For example, radar software in air traffic control system.
- **Network Software:** Software and its components interact across a network.

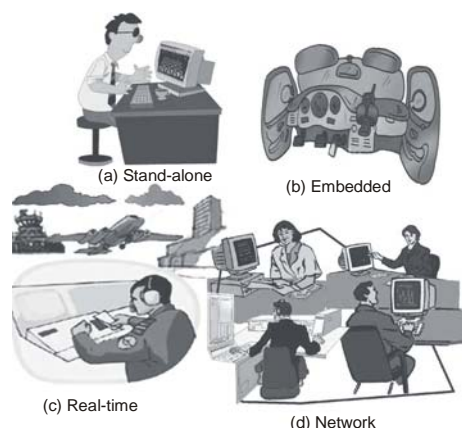


Figure 1.2 Types of Customised COTS

1.2.5 Evolution of Software Engineering

In the late 1960s, it was clear that software development was unlike the construction of physical structures. This was because in software development, more programmers could not be added simply to speed up a lagging development project. Software had become a critical component of many systems, yet it was too complex to develop with any certainty of schedule or quality. This problem imposed financial and public safety concerns.

Software errors have caused large-scale financial losses as well as inconvenience to many. Disasters, such as Y2K problem have affected economic, political, and administrative system 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 importantly, dissatisfaction among users of the software. Other problems associated with software are listed below:

- 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 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.

Software market today has a turnover of more than millions of rupees. Out of this, approximately 30% 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 seen many such failures. Some of these are listed below:

- During the gulf war in 1991, United States of America used Patriot missile as a defence against Iraqi Scud missile. However, this Patriot failed to hit 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 resulted in the miscalculation of missile path.
- 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 a software bug existed 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.
- Year 2000 (Y2K) problem refers to the widespread snags computers had 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 realise this major shortcoming in the computer application and millions were spent to handle this problem.

NOTES

NOTES

1.3 SOFTWARE ENGINEERING: DEFINITION

As discussed earlier, over the last 50 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 efficient manner.

IEEE defines software engineering 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 the technological and managerial discipline concerned with systematic production and maintenance of software that is developed and modified on time and within cost estimates.

Software engineering is a discipline, which can be described as the combination of techniques of engineering and all aspects of software development. This includes design, implementation, and maintenance of software. It includes standardised approach to program development, both in its managerial and technical aspects.

The foundation for software engineering lies in the good working knowledge of computer science theory and practice. The theoretical background involves knowing how and when to use data structures, algorithms, and understanding what problems can be solved and what cannot. The practical knowledge includes thorough understanding of the workings of the hardware as well as thorough knowledge of the available programming languages and tools.

One of the main objectives of software engineering is to help developers obtain high-quality software. This quality is achieved through use of *Total Quality Management*, which enables continuous process improvement custom that leads to the development of more established approaches to software engineering.

Software Engineering Layers: Software engineering can be viewed as a layered technology. The various layers are listed below:

- The process layer is an adhesive that enables rational and timely development of computer software. Process 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 a broad array of tasks that include requirements, analysis, design, program construction, testing, and support phases of the software development.
- The tools layer provides computerised or semi-computerised 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

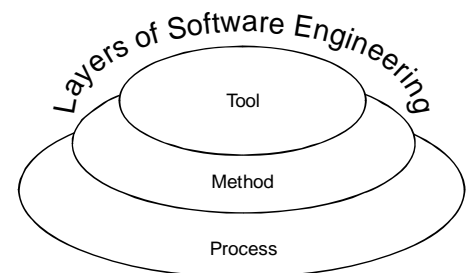


Figure 1.3 Layers of Software Engineering

Check Your Progress

1. What is software?
2. Describe how software evolved in 1960s and 1970s.
3. Explain artificial intelligence software.
4. Define software portability.

NOTES

engineering database to create software engineering analogous to computer-aided design (CAD) for hardware. CASE aids 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. For example, the two prominent delivered technologies using CASE tools are application generators and PC-based workstations that provide graphics-oriented automation of the development process.

1.3.1 Software Engineer

A software engineer is an individual responsible for analysis, design, testing, implementation, and maintenance of effective and efficient software system. In addition, software engineer is also responsible for maintaining subsystems and external interfaces, subject to time and budgetary constraints.

Apart from management of analysis, specification, design and development of software applications, software engineers oversee the certification, maintenance, and testing of software applications. Software engineer also integrates the components of a complex software system. Generally, software engineer should possess the following qualities:

- **Problem solving skills:** Software engineer should develop algorithms and solve programming problems.
- **Programming skills:** Software engineer should be well versed in data structures and algorithms, and must be expert in one or more programming languages and possess strong programming capabilities.
- **Design approaches:** Software engineer should be familiar with numerous design approaches required during the development of software, at the same time, he should be able to translate ambiguous requirements and needs into precise specifications, and be able to converse with the use of a system in terms of applications.
- **Software technologies:** Software engineer should have good understanding of software technologies. Ability to move among several levels of abstractions at different stages of the software project, from specific application procedures and requirements to the detailed coding level is also required.
- **Project management:** Software engineer should know how to make a project work, on time and on budget, in order to produce quality applications and systems.
- **Model of the application:** Software engineer should be able to create and use a model of the application to guide choices of the many tradeoffs that will be faced by him. The model is used to find answers to questions about the behaviour of the system.

In addition to the above-mentioned qualities, software engineer should have good communication and interpersonal skills. Moreover knowledge of object-orientation, quality concept, International Organization of Standardization (ISO standards), and Capability Maturity Model (CMM) are also required. The tasks performed by software engineers have evolved rapidly, which has resulted in new areas of specialization and changing technology. Software engineers often work as part of a team that designs new hardware and software. This team comprises of engineering, marketing, manufacturing, and designing people who work together until the software is released.

NOTES

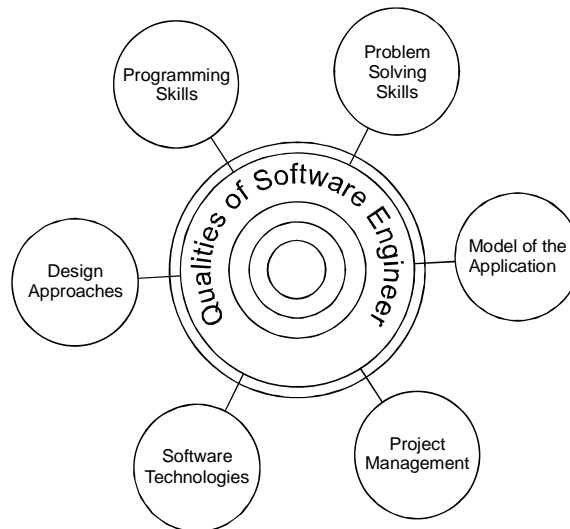


Figure 1.4 Skills of Software Engineer

1.4 PHASES IN SOFTWARE ENGINEERING

Software engineering shares common interest with other engineering disciplines. In the engineering domain, developing a solution to a given problem, whether building a bridge or making an electronic component, involves a sequence of interconnected steps. These steps or phases occur in software development as well. Also, since the prime objective of software engineering is to develop methods for large systems, which produce high quality software at low cost and in reasonable time, it is essential to perform software development in phases. This phased development of software is often referred to as **software development life cycle (SDLC)** or **software life cycle**.

A software development process comprises of different phases. These phases work in top to bottom approach, implying that the phases take inputs from the previous phases, add features, and then produce outputs. The outputs from different phases are referred to as **intermediate product**, **work product**, or derivable. The various phases involved in the systematic development of software are shown in Figure 1.5.

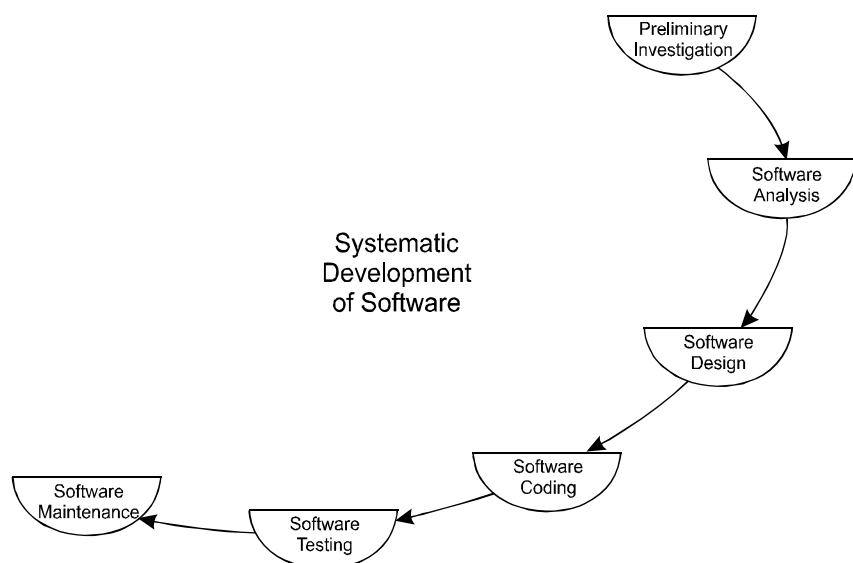


Figure 1.5 Software Development Process

Check Your Progress

5. Define software engineering.
6. What are the responsibilities of software engineer?
7. Explain the process layer.

1.4.1 Preliminary Investigation

This phase commences with discussion on the requests made by the user. The requests can be for a new system or modifying the existing system. An estimate is made of whether the identified user needs can be satisfied with the current hardware and software technologies or not. Preliminary investigation verifies the problem and understands the need for required system. It considers whether the proposed system will be cost effective from business point of view and whether it can be developed within existing budgetary constraints. In addition, time factor, which determines the duration of the project, is also considered.

Preliminary investigation should be quick and cost effective. The output of preliminary investigation decides whether the new system should be developed or not. There are three constraints, which decides the go or no-go decision.

- **Technical:** This evaluation determines whether technology needed for proposed system is available or not and if it is available then how can it be integrated within the organization. Technical evaluation also determines whether the existing system can be upgraded to use new technology and whether the organization has the expertise to use it or not.
- **Time:** This evaluation determines the time needed to complete a project. Time is an important issue in software development as cost increases with an increase in the time period of a project.
- **Budgetary:** This evaluation looks at the financial aspect of the project. Budgetary evaluation determines whether the investment needed to implement the system will be recovered at later stages or not.

(a) **Software Analysis:** This phase studies the problem or requirements of software in detail. These requirements define the processes to be managed during the software development. After analysing the requirements of the user, a requirement statement known as **software requirement specification (SRS)** is developed. After analyses, planning for the project begins. It includes developing plans that describes the activities to be performed during the project, such as software configuration management plans, project and scheduling, and the quality assurance plans. In addition, the resources required during the project are also determined.

Table 1.2 Building Bridge and Corresponding SDLC Phase

Phase	Building Bridge	SDLC Phase
Formulate the problem by understanding the nature and general requirements of the problem.	Understand the load of the bridge it must carry, the approximate locations where it can be built, the height requirements, and so on.	Preliminary investigation.
Defining the problem precisely.	Specify the site for the bridge, its size, and a general outline of the type of bridge to be built.	Software requirement analysis and specifications.
Detailing the solution to the problem.	Determine exact configuration, size of the cables and beams, and developing blueprints for the bridge.	Software design.
Implementing.	Correspond to actual building of the bridge.	Software coding.
Checking.	Specify load, pressure, endurance, and robustness of the bridge.	Software testing.
Maintaining.	Specify repainting, repaving, and making any other repairs, which are necessary.	Software maintenance.

NOTES

NOTES

(b) Software Design: In this phase the requirements are given a 'defined' form. Design can be defined as a process of deciding information structures, in terms of efficiency, flexibility, and reusability. During this phase, strategic and tactical decisions are made to meet the required functional and quality requirements of a system. Software design serves as the blueprint for the implementation of requirement in the software system. Each element of the analysis model in the analysis phase provides information that is required to create design models. The requirement specification of software, together with data, functional, and behavioural models provides a platform to feed the design task to meet required functional and quality requirements of a system.

(c) Software Coding: This phase can be defined as a process of translating the software requirements into a programming language using tools that are available. Writing a software code requires a thorough knowledge of programming language and its tools. Therefore, it is important to choose the appropriate programming language according to the user requirements. A program code is efficient if it makes optimal use of resources and contains minimum errors.

Writing an efficient software code requires thorough knowledge of programming. However, to implement programming, *coding style* is followed. This style is used for writing software code in a programming language. Coding style also helps in writing the software code efficiently and with minimum errors. To ensure that all developers work in a harmonised manner (the source code should reflect a harmonised style, as if a single developer has written the entire code in one session), the developers should be aware of the coding guidelines before the inception of a software project.

(d) Software Testing: This testing is performed to assure that software is free from errors. Efficient testing improves the quality of software. To achieve this, software testing requires a thorough analysis of the software in a systematic manner. Test plan is created to test software in a planned and systematic manner. In addition, software testing is performed to ensure that software produces the correct outputs. This implies that outputs produced should be according to user requirements.

(e) Software Maintenance: This phase comprises of a set of software engineering activities that occur after software is delivered to the user. The objective of software maintenance is to make software operational according to user requirements. The need of software maintenance is to provide continuity of service. This implies that software maintenance focuses on fixing errors, recovering from failures, such as hardware failures, or incompatibility of hardware with software. In addition, it facilitates future maintenance work by modifying the software code and databases used in the software.

After the software is developed and delivered, it may require changes. Sometimes, changes are made in software system when user requirements are not completely met. To make changes in software system, software maintenance process evaluates, controls, and implements changes. Note that changes can also be forced on the software system because of changes in government regulations or changes in policies of the organization.

1.4.2 Case Study: Bridge and Software Development

Requirements analysis, design, and implementation are concerned with, what to do, how to do, and the way to do respectively. For example, Table 1.5 lists the phases involved in building a bridge and also lists the corresponding software development phase.

1.5 SOFTWARE PROJECT MANAGEMENT

Project management is concerned with the overall planning and coordination of a project from its commencement to its completion. This involves application of knowledge, skills,

tools and techniques to meet the user's requirements within the specified time and cost. Effective software project management primarily concentrates on process, project, and product.

A process is defined as a series of steps involving activities and resources, which produce the desired output. Process can also be defined as a collection of procedures to develop a software product according to certain goals or standards. Generally, following points are noted about software processes:

- Process uses resources subject to given constraints, and produces intermediate and final products.
- Processes are composed of sub-processes that are organised in such a manner that each sub-process has its own process model.
- Each process is carried out with 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 more than procedure, 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 software.

The characteristics of software processes are listed in Table 1.6.

Table 1.3 Software Process Characteristics

Characteristics	Description
Understandability	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 defined process is acceptable and usable by the engineers responsible for producing 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 the process can continue inspite of unexpected problems or not.
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 project is developed when software processes or activities are executed for certain specific requirements of the user. Thus, using software process, software project can be easily developed. The activities in a software project comprises of various tasks for managing resources and developing product. Figure 1.6 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 they

NOTES

Check Your Progress

8. Define software development life cycle.
9. Explain different types of constraints that are analyzed during preliminary investigation.

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.

NOTES

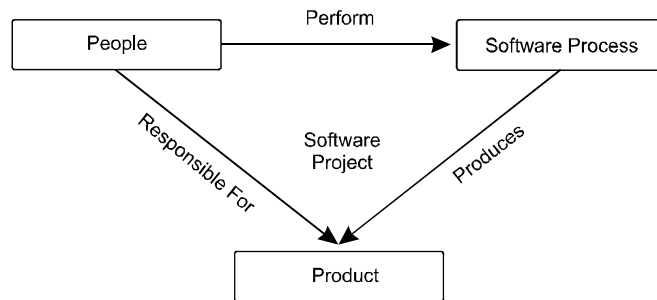


Figure 1.6 Software Project

Software process can consist of many software projects and each of them can produce one or more software products. The interrelationship between these three entities (process, project, and product) is shown in Figure 1.7. A software project begins with requirements and ends with the accomplishment of requirements. Thus, software process should be performed to develop final software by accomplishing the user requirements. Note that software processes are not specific to the software project.

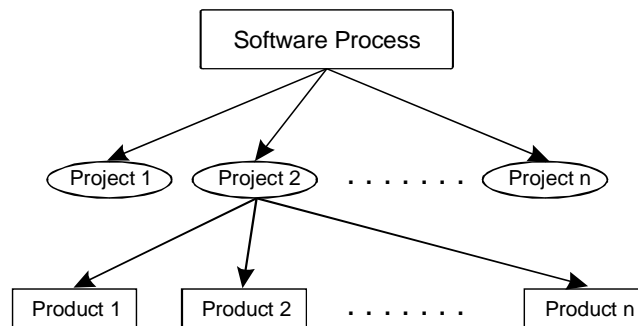


Figure 1.7 Processes, Projects, and Products

1.5.1 Process Management and Product Engineering Process

As stated above, the objective of software process is to develop a product, which accomplishes user requirements. For this, software processes requires components, which are shown in Figure 1.8. The major components of software process include process management process and product engineering process. The **process management processes** (PMP) aims at improving software processes so that a cost effective and a 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 affecting 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 analysis stated above, the **product engineering processes** are improved, thereby improving the software process. The aim of product engineering processes is to develop the product according to user requirements. The product engineering process comprises of three major components, which are listed below:

- **Development process:** Implies the process, which is used during the development of software. It specifies the development and quality assurance activities that are to be

NOTES

- performed. Programmers, designers, testing personnel, and others perform these processes.
- **Project management process:** Provides means to plan, organise 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. Also, project management process is concerned with the set of activities or tasks, which are used to successfully accomplish a set of goals.
 - **Configuration control process:** Manages 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 process are performed by a group called configuration control board (CCB).

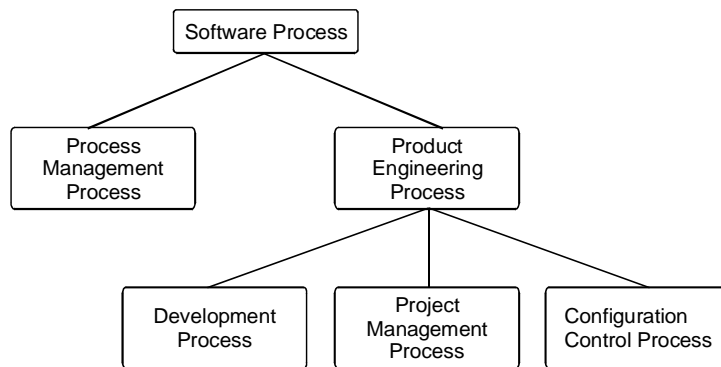


Figure 1.8 Software Processes

Note that 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.

1.5.2 Process Framework

Process framework determines the processes, which 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 1.9. Some of the framework activities are listed below:

- **Communication:** Involves communication with the user so that the requirements are easy to understand.
- **Planning:** Establishes a project plan for the project. In addition, it describes the schedule for the project, technical tasks involved, expected risks and the required resources.
- **Modelling:** 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:** Combines generation of code with testing to uncover errors in the code.
- **Deployment:** 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.

NOTES

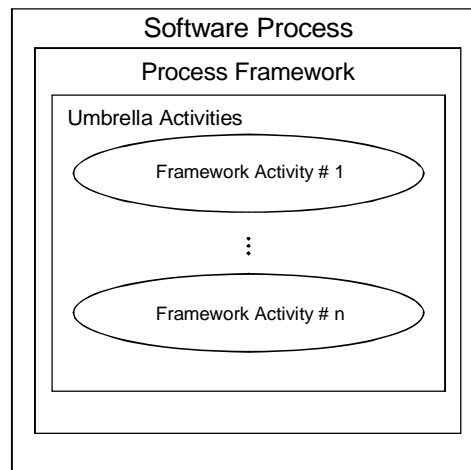


Figure 1.9 Process Framework

In addition to these activities, process framework also comprises of a set of activities known as **umbrella activities**. The umbrella activities are used throughout the software process and are listed below:

- **Software project tracking and control:** Monitors the actual process so that management can take necessary steps if software project deviates from the laid plans. 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:** Assess the code, products and documents of software engineering practises to detect errors.
- **Software quality assurance:** Assures that 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:** Determines the criteria for products' reuse and establishes mechanisms to achieve reusable components.
- **Software configuration management:** Manages the changes made in the software processes of the products throughout the software project life cycle. It controls changes made to the configuration and maintains the integrity in the software development process.
- **Risk management:** Identifies, analyses, evaluates and eliminates the possibility of unfavourable deviations from expected results, by a systematic activity and then develops strategies to manage them.

1.6 PROCESS MODELS

A process model also known as **software engineering paradigm** can be defined as a strategy which comprises of processes, methods, tools or steps for developing software. These models provide a basis for controlling various activities required to develop and maintain software. In addition, it helps 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 software project. Thus, it is essential to define process models for each software project. IEEE defines 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.” 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 listed below:

- **Enables effective communication:** Enhances understanding and provides a specific basis for process execution.
- **Facilitates process reuse:** Process development is a time-consuming and expensive activity, thus, software development team utilise the existing processes for different projects.
- **Effective:** Since process models can be used again and again; reusable processes provide an effective means for implementing processes for software development.
- **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 product 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 software development lifecycle.

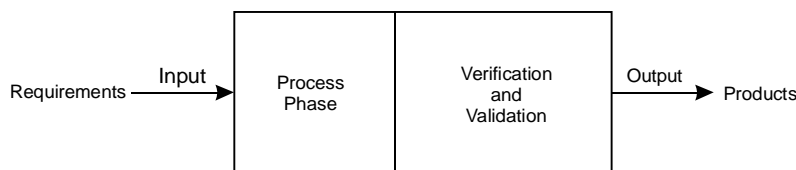


Figure 1.10 Phases in Development Process

Verification is the process of evaluating a system or its component 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 ensure compliance with the requirements. In addition, it is the process of establishing a procedure and a 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.

Various kinds of process models used are *waterfall model*, *prototyping model*, *spiral model*, and *fourth generation techniques*.

1.6.1 Waterfall Model

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

This model is simple to understand, and represents processes, which are easy to manage and measure. Waterfall model comprises of different phases and each phase has its distinct goal. Figure 1.11 shows that once a phase is completed, the development of software

NOTES

Check Your Progress

10. What is software project management?
11. What is the aim of process management processes (PMP)?
12. Define process framework.

proceeds to the next phase. 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 1.4.

NOTES

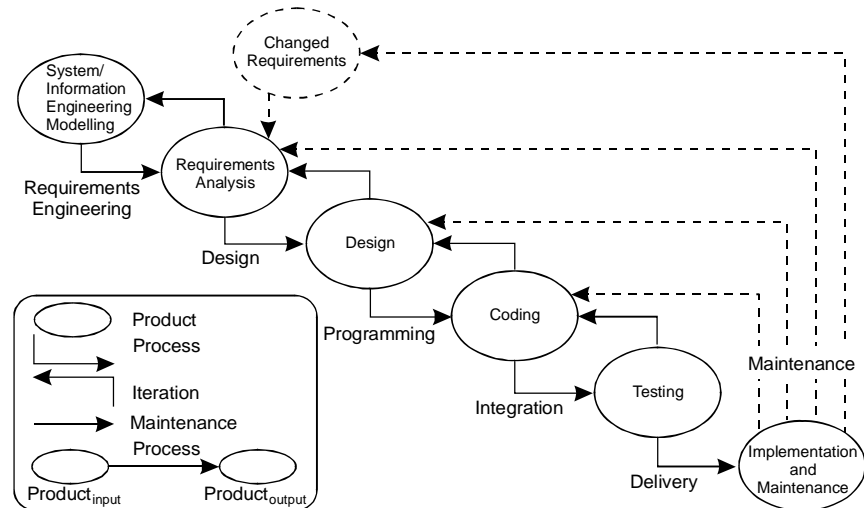


Figure 1.11 Waterfall Model

Table 1.4 Processes and Products of Waterfall Model

Input to the 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, waterfall model comprises of several phases. These phases are listed below:

- System/information engineering modelling:** 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. The information gathering is necessary when the requirements are collected at a level where all decisions regarding business strategies are taken.
- Requirement analysis:** Focuses on the requirements of the software which is to be developed. It determines the processes that are incorporated during the development of software. To specify the requirements' users specification should be clearly understood and the requirements should be analysed. This phase involves interaction between user and software engineer, and produces a document known as **software requirement specification (SRS)**.
- Design:** Determines the detailed process of developing software after the requirements are analysed. It utilises 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, in this phase is mainly concerned with the data structure, algorithmic detail, and interface representations.

- **Coding:** Emphasises on 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:** Ensures that the product is developed according to the requirements of the user. Testing is performed to verify that the product is functioning efficiently with minimum errors. It focuses on the internal logics and external functions of the software and ensures that all the statements have been exercised (tested). Note that testing is a multi-stage activity, which emphasises verification and validation of the product.
- **Implementation and maintenance:** 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 external environment (these include upgrading new operating system or addition of a new peripheral device). The changes also occur due to changing requirements of the user and the changes occurring in the field of technology. This phase focuses on modifying software, correcting errors, and improving the performance of the software.

The various advantages and disadvantages associated with waterfall model are listed in Table 1.5.

Table 1.5 Advantages and Disadvantages of Waterfall Model

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Relatively simple to understand. ▪ Each phase of development proceeds sequentially. ▪ Allows managerial control where a schedule with deadlines is set for each stage of development. ▪ Helps in controlling schedules, budgets, and documentation. 	<ul style="list-style-type: none"> ▪ Requirements need to be specified before the development proceeds. ▪ The changes of requirements in later phases of the waterfall model cannot be done. This implies that once an application is in the testing phase, it is difficult to incorporate changes at such a late phase. ▪ No user involvement and working version of the software is available when the software is developed. ▪ Does not involve risk management. ▪ Assumes that requirements are stable and are frozen across the project span.

1.6.2 Prototyping Model

The prototyping model is applied when there is an absence of detailed information regarding input and output requirements in the software. Prototyping model is developed on the assumption that it is often difficult to know all the requirements at the beginning of a project. It is usually used when there does not exist a system or in case of large and complex system where there is no manual process to determine the requirements.

Prototyping model increases flexibility of the development process by allowing the user to interact and experiment with a working representation of the product 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 thrown away and an entirely new system is developed. Generally, prototyping can be prepared by following the approaches listed below:

NOTES

NOTES

- By creating major user interfaces without any substantive coding in the background in order to give the users a feel of what the system will look like.
- By abbreviating a version of the system that will perform limited subsets of functions.
- Using system components to demonstrate functions that will be included in the developed system.

Figure 1.12 illustrates the steps carried out in prototyping model. All these steps are listed below:

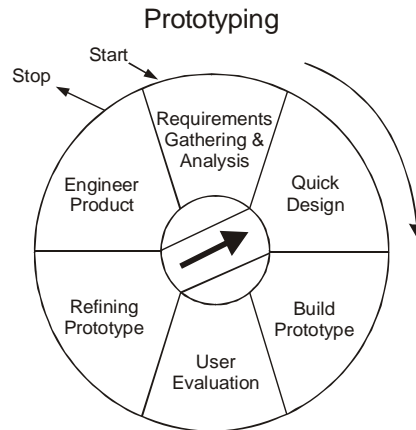


Figure 1.12 Prototyping Model

1. **Requirements gathering and analysis:** Prototyping model begins with requirements analysis, and the requirements of the system are defined in detail. The user is interviewed to know the requirements of the system.
2. **Quick design:** When requirements are known, a preliminary design or a quick design for the system is created. It is not a detailed design, however, it includes the important aspects of the system, which gives an idea of the system to the user. Quick design helps in developing the prototype.
3. **Build prototype:** Information gathered from quick design is modified to form a prototype. The first prototype of the required system is developed from quick design. It represents a 'rough' design of the required system.
4. **Assessment or user evaluation:** Next, the proposed system is presented to the user for consideration as a part of development process. The users thoroughly evaluate the prototype and recognise 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, it is refined according to the requirements. The developer revises the prototype to make it more effective and efficient according to the user requirement. If the user is not satisfied with the developed prototype, then a new prototype is developed with the additional information provided by the user. The new prototype is evaluated in the same manner, as the previous 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 based on the final prototype.
6. **Engineer product:** Once the requirements are completely known, user accepts the final prototype. The final system is thoroughly evaluated and tested followed by routine maintenance on continuing basis to prevent large-scale failures and to minimise downtime.

The various advantages and disadvantages associated with prototyping model are listed in Table 1.6.

Table 1.6 Advantages and Disadvantages of Prototyping Model

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Provides a working model to the user early in the process, enabling early assessment and increasing user confidence. ▪ Developer gains experience and insight by developing a prototype, thereby resulting in better implementation of requirements. ▪ Prototyping model serves to clarify requirements, which are not clear, hence reducing ambiguity and improving communication between developer and user. ▪ There is a great involvement of users in software development. Hence, the requirements of the users are met to the greatest extent. ▪ Helps in reducing risks associated with the project. 	<ul style="list-style-type: none"> ▪ 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. ▪ Developer loses focus of the real purpose of prototype and compromise with the quality of the product. For example, they apply some of the inefficient algorithms or inappropriate programming languages used in developing the prototype. ▪ Prototyping can lead to false expectations. It often creates a situation where user believes that the development of the system is finished when it is not. ▪ The primary goal of prototyping is rapid development, thus, the design of system can suffer as it is built in a series of layers without considering integration of all the other components.

NOTES**1.6.3 Spiral Model**

In 1980's Boehm introduced a process model known as **spiral model**. The spiral model comprises of activities organised in a spiral, which has many cycles. This model combines the features of 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, spiral model 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 spiral model is to emphasise 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 obsolete the project. Figure 1.13 shows the spiral model and the steps involved in the model are listed below:

1. 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.
2. 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 1.13 shows the strategy, which includes prototyping, simulation, benchmarking administering user questionnaires or risk resolution technique.
3. 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.

4. 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.

NOTES

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.

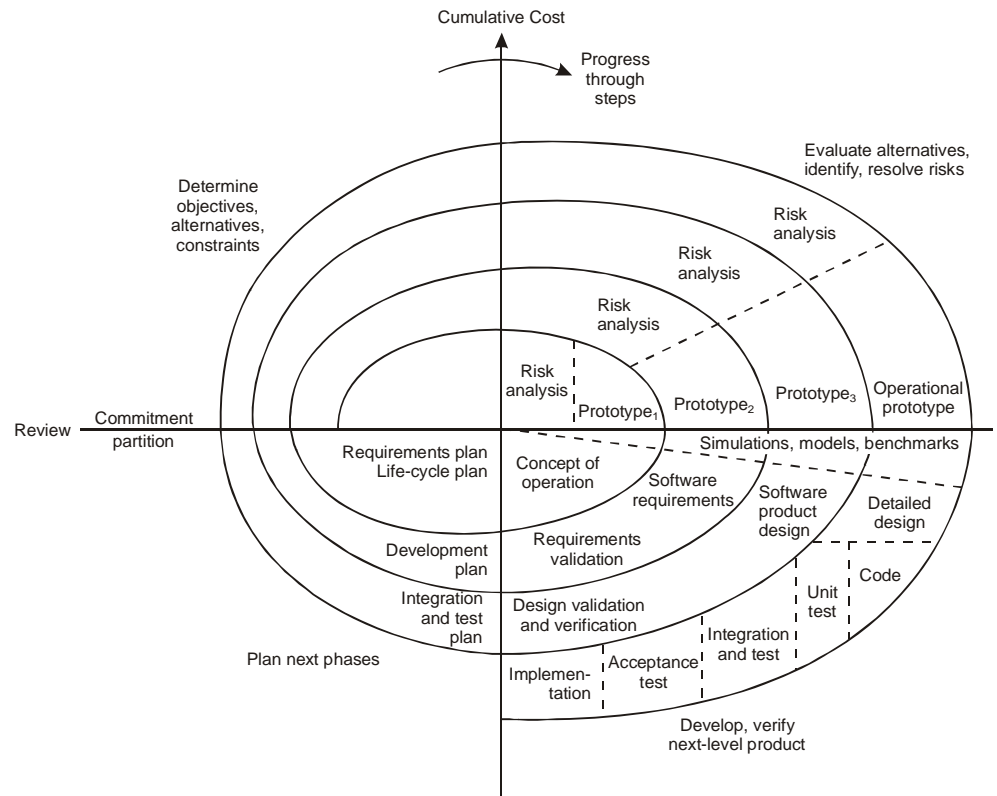


Figure 1.13 Spiral Model

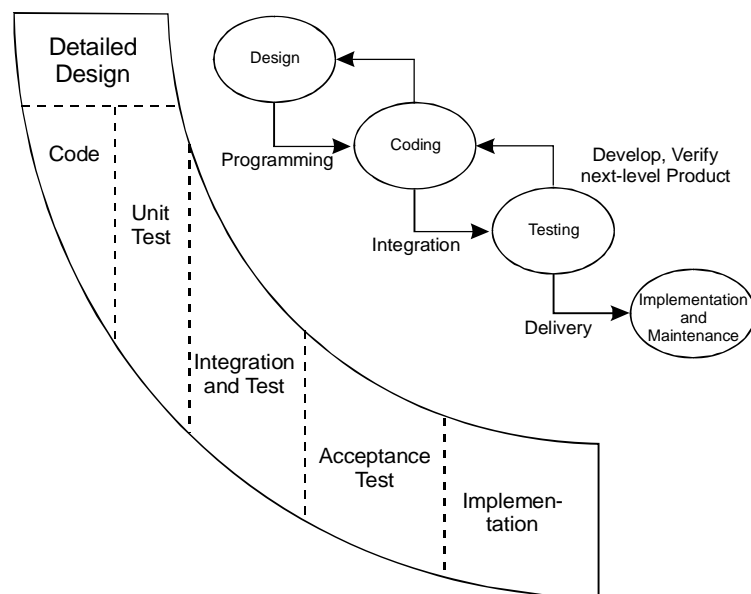


Figure 1.14 Spiral and Waterfall Model

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 spiral model. When these issues are resolved, a detail design of the software is developed. Note that processes in the waterfall model are followed by different cycles in the spiral model as shown in Figure 1.14.

The various advantages and disadvantages associated with spiral model are listed in Table 1.7.

Spiral model is also similar to 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.

Table 1.7 Advantages and Disadvantages of Spiral Model

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Avoids the problems resulting in risk-driven approach in the software. ▪ Specifies a mechanism for software quality assurance activities. ▪ Spiral model is utilised by complex and dynamic projects. ▪ Re-evaluation after each step allows changes in user perspectives, technology advances or financial perspectives. ▪ Estimation of budget and schedule gets realistic as the work progresses. 	<ul style="list-style-type: none"> ▪ Assessment of project risks and its resolution is not an easy task. ▪ 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.

1.6.4 Fourth Generation Techniques (4GT)

Fourth generation techniques enable software engineers to specify characteristics of software at a high level and then automatically generate the source code. In addition to being a process model, fourth generation techniques are a collection of software tools used by software engineers to solve a problem by using a specialised language or a graphic notation so that users easily understand the problem. Hence, fourth generation techniques use instructions similar to spoken languages to allow the programmers to define *what* they want the computer to do rather than *how* to do it. For this, fourth generation techniques use certain tools, which are listed below:

- Non-procedural languages for database query.
- Report generation.
- Data manipulation.
- Screen interaction and definition.
- Code generation.
- High-level graphics capability.
- Spreadsheet capability.
- Automated generation of hypertext markup language and similar languages used for web-site creation using advanced software tools.

The various advantages and disadvantages associated with fourth generation techniques are listed in Table 1.8.

NOTES

Table 1.8 Advantages and disadvantages of Fourth generation Techniques**NOTES**

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Development time is reduced, when used for small and intermediate applications. ▪ The interaction between user and developer helps in detection of errors. ▪ When integrated with CASE tools and code generators, fourth generation techniques provide a solution to most of the software engineering problems. 	<ul style="list-style-type: none"> ▪ Difficult to use. ▪ Limited only to small business information systems.

1.7 ROLE OF SOFTWARE METRICS AND MEASUREMENT

To achieve accurate schedule and cost estimate, better quality products, and higher productivity an effective software management is required, which in-turn can be attained through use of software metrics. A metric is a derived unit of measurement that cannot be directly observed, but is created by combining or relating two or more measures.

1.7.1 Software Measurement

To assess the quality of the engineered product or system and to better understand the models that are created, some measures are used. These measures are collected throughout the software development life cycle with an intention to improve the software **process** on a continuous basis. Measurement helps in estimation, quality control, productivity assessment, and project control throughout a software **project**. Also, measurement is used by software engineers to gain insight into the design and development of the work **products**. In addition, measurement assists in strategic decision-making as a project proceeds.

Software measurements are of two categories namely, direct measures and indirect measures. **Direct measures** include software processes like cost and effort applied and product like lines of code produced, execution speed, and other defects that have been reported. **Indirect measures** include products like functionality, quality, complexity, reliability, maintainability, and much more.

Generally, software measurement is considered as a management tool, which if conducted in an effective manner helps project manager and the entire software team to take decisions that lead to successful completion of the project. Measurement process is characterised by a set of five activities, which are listed below:

- **Formulation:** Performs measurement and develops appropriate metrics for software under consideration.
- **Collection:** Collects data to derive the formulated metrics.
- **Analysis:** Calculates metrics and use mathematical tools.
- **Interpretation:** Analyses the metrics to attain insight into the quality of representation.
- **Feedback:** Communicates recommendation derived from product metrics to the software team.

Note that collection and analysis activities drive the measurement process. In order to perform these activities effectively, it is recommended to automate data collection and analysis, establish guidelines and recommendations for each metric, and use statistical techniques to interrelate external quality features and internal product attributes.

Check Your Progress

13. Define process model.
14. Why is waterfall model also referred to as linear sequential model?
15. Explain the scenario in which prototyping model is best suited.
16. List the advantages associated with fourth generation techniques.

1.7.2 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 listed below:

- Measure the size of the software quantitatively.
- Assess the level of complexity involved.
- Assess the strength of the module by measuring coupling.
- Assess the testing techniques.
- Specify when to stop testing.
- Determine the date of release of the software.
- Estimate 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.

1.8 LET US SUMMARIZE

1. 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.
2. Software characteristics are classified into six major components, namely functionality, reliability, usability, efficiency, maintainability, and portability.
3. 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. Based on 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.
4. Based on how closely software users or software purchasers are associated with the software development, software is classified as commercial of the shelf software, customised or bespoke software, and customised COTS software.
5. Disasters, such as Y2K problem have affected economic, political, and administrative system of various countries around the world. This situation where catastrophic failures have occurred is known as software crisis.
6. The major cause of software crisis is the problems associated with the poor quality of software, such as malfunctioning of software systems, inefficient development of software, and dissatisfaction among the users who are using the software.
7. Software engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
8. There are three layers of software engineering, namely process layer, method layer, and tools layer.

NOTES

Check Your Progress

17. Mention various direct and indirect measures associated with software measurement.
18. Define metric.

NOTES

9. Software engineer is an individual responsible for analysis, design, testing, implementation, and maintenance of effective and efficient software system.
10. Various phases involved in systematic development of software are preliminary investigation, software analysis, software design, software coding, software testing, and software maintenance.
11. Software process comprises of activities, which are used to develop a software project. The major characteristics of software processes include understandability, reliability, and maintainability.
12. To accomplish the objectives of software, the major processes required are development process, management process, and configuration control process.
13. Various processes are used, as a process framework to develop a software project in the organization. The framework activities comprise of umbrella activities, which are used across the software process.
14. Some of the umbrella activities include software project tracking, software quality assurance, reusability management, software configuration management and risk management.
15. Software product engineering aims at consistently performing a well-defined engineering process, which integrates all the software engineering activities to develop correct, effective, efficient, and consistent software.
16. To develop quality software it is essential to use effective software processes. Thus, processes are assessed to evaluate methods, tools, practices, organizational structure, and environment, which are used to develop software.
17. A software process model is an abstraction of the process. The purpose of software process model is to determine the order of stages, which are involved in the development of the software. It is essential to define process models for each software project.
18. Some of the process models are waterfall model, prototyping model, spiral model, and fourth generation techniques.
19. The requirements of the software in waterfall model are defined early in software development life cycle. To develop software, waterfall model uses different stages. This model is simple to understand and easy to use.
20. In 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.
21. 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.
22. Fourth generation techniques are a collection of software tools used by software engineers to solve a problem by using a specialised language or a graphic notation so that users easily understand the problem.
23. To assess the quality of the engineered product or system and to better understand the models that are created, some measures are used. These measures are collected throughout the software development life cycle with an intention to improve the software process on a continuous basis.
24. Once measures are collected they are converted into metrics for use.

1.9 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. 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”.

NOTES

2. **1960s:** Software was developed for specific areas and was being marketed and sold separately from hardware. This marked a deviation from earlier practices of giving software free as a part of the hardware platform. In addition, hiding of internal details of an operating system using abstract programming interfaces improved the productivity of the programmer.
1970s: With the development of structured design, software development models were introduced. These were based on a more organic, evolutionary approach, deviating from the waterfall-based methodologies of hardware engineering. Research was done on quantitative techniques for software design. During this time, researchers began to focus on software design to address the problems of developing complex software systems.
3. 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.
4. Software portability 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.
5. IEEE defines software engineering 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 the technological and managerial discipline concerned with systematic production and maintenance of software that is developed and modified on time and within cost estimates.
6. A software engineer is responsible for analysis, design, testing, implementation, and maintenance of effective and efficient software system. In addition, software engineer is responsible for maintaining subsystems and external interfaces, subject to time and budgetary constraints.
7. The process layer is an adhesive that enables rational and timely development of computer software. Process defines an outline for a set of key process areas that must be acclaimed for effective delivery of software engineering technology.
8. Software engineering shares common interest with other engineering disciplines. In the engineering domain, developing a solution to a given problem, whether building a bridge or making an electronic component, involves a sequence of interconnected steps. These steps or phases occur in software development as well. Also, since the prime objective of software engineering is to develop methods for large systems, which produce high quality software at low cost and in reasonable time, it is essential to perform software development in phases. This phased development of software is often referred to as **software development life cycle (SDLC)** or **software life cycle**.
9. There are three constraints that are analyzed during preliminary investigation.
 - **Technical:** This evaluation primarily determines whether technology needed for proposed system is available or not and if it is available then how can it be integrated within the organization.
 - **Time:** This evaluation determines the time needed to complete a project.

NOTES

- **Budgetary:** This determines whether the investment needed to implement the system will be recovered at later stages or not.
10. Software project management is concerned with the overall planning and coordination of a software project from its commencement to its completion. This involves application of knowledge, skills, tools and techniques to meet the user's requirements within the specified time and cost.
 11. The process management processes aims at improving software processes so that a cost effective and a high quality product is developed. The process of comprehending the existing process, analyzing its properties, determining how to improve it, and then affecting the improvement is also carried out by PMP.
 12. Process framework determines the processes, which are essential for completing a complex software project. This framework also identifies certain activities, which are applicable to all software projects, regardless of their type and complexity.
 13. A process model also known as software engineering paradigm can be defined as a strategy, which comprises of processes, methods, tools or steps for developing software. These models provide a basis for controlling various activities required to develop and maintain software. In addition, it helps the software development team in facilitating and understanding the activities involved in the project.
 14. In waterfall model the development of software proceeds linearly and sequentially from requirement analysis to design, coding, testing, integration, implementation, and maintenance. Therefore, this model is also known as linear sequential model.
 15. The prototyping model is best suited when there is an absence of detailed information regarding input and output requirements in the software. Prototyping model is developed on the assumption that it is often difficult to know all the requirements at the beginning of a project. It is usually used when there does not exist a system or in case of large and complex system where there is no manual process to determine the requirements.
 16. Fourth generation techniques enable software engineers to specify characteristics of software at a high level and then automatically generate the source code. In addition to being a process model, fourth generation techniques are a collection of software tools used by software engineers to solve a problem by using a specialised language or a graphic notation so that users easily understand the problem.
 17. Software measurements can be categorized into direct measures and indirect measures. **Direct measures** include software processes like cost and effort applied and product like lines of code produced, execution speed, and other defects that have been reported. **Indirect measures** include products like functionality, quality, complexity, reliability, maintainability, and much more.
 18. A metric is a derived unit of measurement that cannot be directly observed, but is created by combining or relating two or more measures. IEEE defines metric as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

1.10 QUESTIONS AND EXERCISES

I. Fill in the Blanks

1. Based on its applications, software is classified as system software, _____, business software, _____, artificial intelligence software, _____, and personal computer software.

2. There are three layers of software engineering, namely _____, _____, and _____.
3. _____ confirms that software is translated according to the sufficient accuracy.

NOTES

II. Multiple Choice Questions

1. Which of the following does not compose software?
(a) Programs (b) Hardware (c) Data (d) Documentation
2. Which of the following is not the characteristic of software?
(a) Availability (b) Maintainability (c) Usability (d) Efficiency
3. Which of the following is not the type of software?
(a) Web-based software (b) System software
(c) Management software (d) Customised COTS

III. State Whether True or False

1. Software is a degradable entity.
2. Functionality refers to the ability of software to perform a required function under given conditions for a specified period.
3. Tools layer provides computerised and semi-computerised support for process and method layer.

IV. Descriptive Questions

1. Define software along with its characteristics?
2. Write a short note on the following
Evolution of Software Engineering.
3. The main tasks of software engineer are to analyse, design, test, implement, and maintain the software. So as to perform all these tasks efficiently software engineer should possess certain qualities. Explain them.
4. What is a process model? Outline the major steps involved in a spiral model.

1.11 FURTHER READING

1. Software Engineering: A Practitioner's Approach—*Pressman*
2. Software Engineering – *Ian Sommerville*
3. Software Engineering – *K. K. Aggarwal and Yogesh Singh*

UNIT 2 SOFTWARE PROJECT PLANNING AND COST ESTIMATION

NOTES

Structure

- 2.0 Introduction
- 2.1 Unit Objectives
- 2.2 Project Planning
 - 2.2.1 Project Purpose; 2.2.2 Project Scope;
 - 2.2.3 Project Planning Process; 2.2.4 Project Plan
- 2.3 Project Scheduling
- 2.4 Basics of Cost Estimation
 - 2.4.1 Resources for Software Cost Estimation; 2.4.2 Software Product Cost Factors
- 2.5 Software Cost Estimation Process
- 2.6 Decomposition Techniques
 - 2.6.1 Problem-based Estimation; 2.6.2 Process-based Estimation
- 2.7 Cost Estimation Models
 - 2.7.1 Constructive Cost Model; 2.7.2 Software Equation
- 2.8 Let us Summarize
- 2.9 Answers to 'Check Your Progress'
- 2.10 Questions and Exercises
- 2.11 Further Reading

2.0 INTRODUCTION

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

Cost estimation should be done before software development is initiated since it helps the project manager to know about resources required and the feasibility of the project. Also, the initial estimate may be used to establish budget for the project or to set a price for the software to the potential customer. However, estimate must be done repeatedly throughout the development process, as more information about the project is available in the later stages of development. This helps in effective usage of resources and time. For example, if actual expenditure is greater than the estimate, then the project manager may apply additional resources for the project or modify the work to be carried out.

2.1 UNIT OBJECTIVES

After reading this unit, the reader will understand:

- The need, scope, and purpose of project planning.
- Project planning process.

NOTES

- Project plan.
- Project scheduling.
- Resources required for accurate software cost estimation.
- Factors that influence the cost of developing a software product.
- Software cost estimation process.
- Decomposition techniques, such as problem-based and process-based estimation.
- Cost estimation models like COCOMO and software equation.
- How estimation is carried out in object-oriented projects.

2.2 PROJECT PLANNING

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, which focuses on activities required for successful completion of the 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 utilisation of resources and optimal usage of the allotted time for a project. The other objectives of project planning are listed below:

- Define roles and responsibilities of the project management team members.
- Ensure that project management team works according to business objectives.
- Check feasibility of schedule and user requirements.
- Determine 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. **Project management team**, which generally includes project managers and developers, is responsible for planning, determining, and tracking the activities of the project. Table 2.1 lists the tasks performed by individuals involved in the software project.

Table 2.1 Tasks of Individuals involved in Software Project

Senior Management	Project Management Team
<ul style="list-style-type: none"> ▪ Approves the project, employ 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. 	<ul style="list-style-type: none"> ▪ Reviews the project plan 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.

Note: In software project, tasks and activities represent the tasks performed during software development. Hence, both the terms are used interchangeably throughout this chapter.

Project planning comprises of *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.

2.2.1 Project Purpose

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 listed below:

- **Meet user requirements:** Develop the project according to user requirements after understanding them.
- **Be according to schedule:** Complete the project milestones as described in the project plan on time in order to complete the project according to schedule.
- **Be within budget:** Manage the overall project cost so that the project is within budget.
- **Produce quality deliverables:** 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 listed below:

- **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.

2.2.2 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 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 software according to user requirements. After this, 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. Project scope describes the information listed below:

- 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 functionality.

NOTES

NOTES

2.2.3 Project Planning Process

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 of 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 of several activities, which are essential for carrying out a project systematically. These activities refer to the series of tasks, which 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.

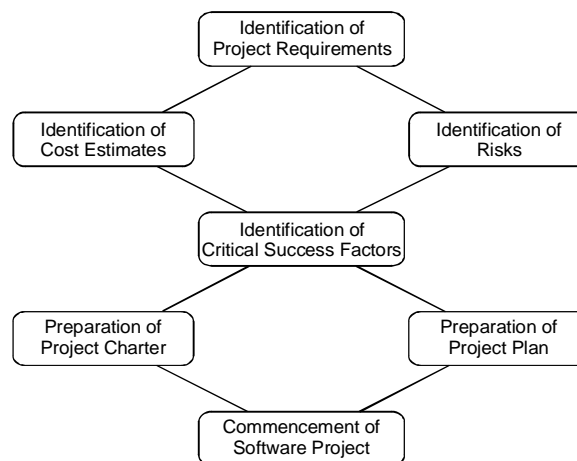


Figure 2.1 Project Planning Activities

Figure 2.1 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 listed below:

- **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 of 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 adverse effect on the project. Software project involves several risks (like technical risks and business risks) that 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.

NOTES

- **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 project planning. It is prepared by the management 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.

Figure 2.1 shows the process of project planning. Once the project objectives and business objectives are determined, the project end date is fixed. 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, there is delay in completing the project, the project plan is re-analyzed and corrective actions are taken to complete the project. The project is tracked regularly and when the project plan is modified, the senior management is informed.

2.2.4 Project Plan

As stated earlier, project plan stores the outcome of project planning. 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, 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. Various advantages associated with project plan are listed below:

- Ensures that software is developed according to user requirements, objectives, and scope of the project.
- Identifies the role of each project management team member involved in the project.
- Monitors the progress of the project according to the project plan.
- Determines the available resources and the activities to be performed during software development.
- Provides an overview to management about the costs of the software project, which are estimated during project planning.

Note that there are differences in the contents of two or more project plans as they differ depending on the kind of project and user requirements. Project plan is divided into several sections, which are listed below:

- **Introduction:** Describes the objectives of the project and provides information about the constraints that affect the software project.
- **Project organization:** Describes the responsibilities, which are assigned to the project management team members for completing the project.

NOTES

- **Risk analysis:** 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:** Specifies the hardware and software that are required to carry out the software project. According to these resource requirements, cost estimation is done.
- **Work breakdown:** Describes the activities into which the project is divided. It also describes the milestones and deliverables of the project activities.
- **Project schedule:** 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*.

(a) **Quality Assurance:** The quality assurance plan describes the strategies and methods that are to be followed to accomplish the objectives listed below:

- 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.

(b) **Verification and Validation:** Verification and validation plan describes the approach, resources, and schedule used for system validation.

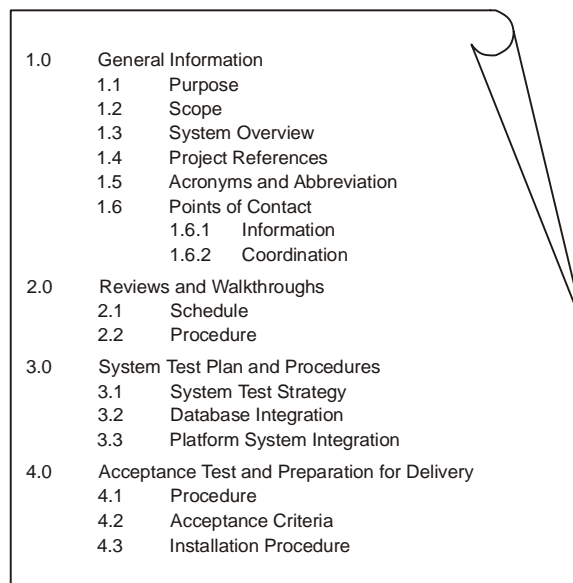


Figure 2.2 Verification and Validation Plan

Figure 2.2 shows the verification and validation plan, which comprises of various sections listed below:

- **General information:** Provides description of the purpose, scope, 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

NOTES

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:** 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:** 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:** 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.

(c) **Configuration Management:** The configuration management plan defines the process, which is used for making changes to the project scope. Generally, 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).

(d) **Maintenance:** 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.

Figure 2.3 shows the maintenance plan, which comprises of various sections listed below:

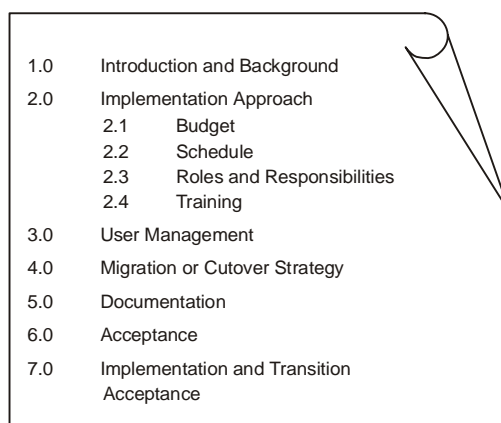


Figure 2.3 Maintenance Plan

NOTES

- **Introduction and background:** Provide description of 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:** Specifies the budget required for carrying out software maintenance and operations activities.
- **Roles and responsibilities:** Specify 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 of user support, user training and support staff.
- **Performance measures and reporting:** Identify 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:** 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. Management approach also describes how users are involved in software maintenance and operations activities. In addition, it describes how users and project management team communicate with each other.
- **Documentation strategies:** Provide 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:** Provides information about training activities.
- **Acceptance:** Defines a point of agreement between the project management team and software maintenance team after the completion of implementation and transition activities. After this, software maintenance begins.

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

Staff planning is performed by a staff planner, who is responsible for determining the individuals available for the project. The other responsibilities of a staff planner are listed below:

- Determines individuals, who can be 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.
- 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.
- 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.

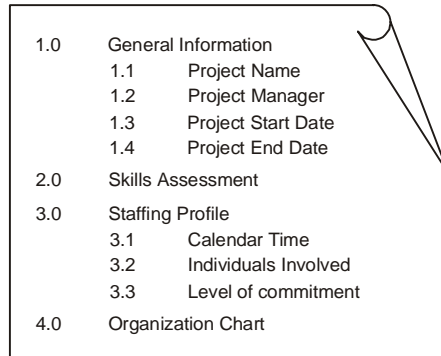


Figure 2.4 Staffing Plan

- Defines 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 sub-team depending on the size and complexity of the project.

Figure 2.4 shows staffing plan which comprises of various sections listed below:

- **General information:** 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:** Provides information, which 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:** 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** that are involved in the project have specific designations, such as project manager and the developer. **Level of commitment** is the utilisation rate of individuals, such as work performed on full time and part time basis.
- **Organization chart:** Describes the organization of project management team members. In addition, it includes information, such as name, designation, and role of each team member.

2.3 PROJECT SCHEDULING

It is essential to perform project scheduling to effectively manage the tasks of the 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 another. An appropriate project schedule prepared according to 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 listed below:

- **Unrealistic deadline:** 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 deadline is established by inexperienced individual(s) or without

NOTES

Check Your Progress

1. Define project planning.
2. Mention different considerations described in project scope.
3. Mention different project planning process activities.
4. What information is provided under skills assessment?

NOTES

the help of project management team. Here, the project management team is constrained to work according to that deadline. The project is delayed if the deadline is not achieved.

- **Changing 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 both in revision of project plan and implementation of new user requirements.
- **Under-estimation of resources:** If the estimation of the resources for the project is not done according to its requirement, the schedule is affected. This under-estimation of resources leads to delay in performing tasks of the project.
- **Lack of consideration of risks:** Risks should be considered during project planning and scheduling, otherwise it becomes difficult for 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 project management team to understand and develop the software according to user requirements and schedule.
- **Difficulties of team members:** Software project can also be delayed due to unforeseen difficulties of the team members. For example, some of the team members may require leave for personal reasons.
- **Lack of action by project management team:** Sometimes, project management team does not recognize that the project is getting delayed. Thus they do not take necessary action to speed up the software development process and complete it on time.

Generally, the task of assigning the end date is done by the project sponsor or the user. While preparing the project schedule, project manager assists the project sponsor by providing information about the project scope, deliverables, and resources. In addition, project manager provides an estimate of the time to be consumed to complete project tasks. Preparing an accurate project schedule is a difficult task and requires thorough knowledge about the processes and time consumed to perform them. 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.

2.4 BASICS OF COST ESTIMATION

Cost estimation is the process of approximating the costs involved in the software project. Cost estimation should be done before software development is initiated since it helps the project manager to know about resources required and the feasibility of the project.

Accurate software cost estimation is important for the successful completion of a software project. However, the need and importance of software cost estimation is underestimated due to the reasons listed below:

- Analysis of the software development process is not considered while estimating cost.
- It is difficult to estimate software cost accurately, as software is intangible and intractable.

There are many parameters (also called factors), such as complexity, time availability, and reliability, which are considered during cost estimation process. However, software size is considered as one of the most important parameters for cost estimation.

Cost estimation can be performed during any phase of software development. The accuracy of cost estimation depends on the availability of software information (requirements, design, and source code). It is easier to estimate the cost in the later stages, as more information is available during these stages as compared to the information available in the initial stages of

Check Your Progress

5. Why is project scheduling essential?
6. Explain unrealistic deadline.

software development. Figure 2.5 shows accuracy of cost estimation in each phase of development life cycle.

NOTES

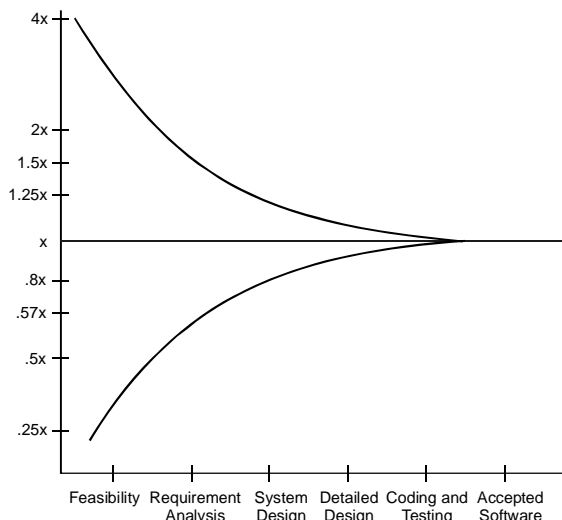


Figure 2.5 Accuracy of Cost Estimation

In Figure 2.5, the funnel shaped lines narrowing at the right hand side show how cost estimates get more accurate as additional software information is available. For example, cost estimated during system design phase is more accurate than cost estimated during requirement phase. Similarly, cost estimated during coding and testing phase is more accurate than it is at design phase. Note that when all the information about project is not known, the initial estimate may differ from the final estimate by factor of four.

Note: Cost estimation should be done more diligently throughout the life cycle of the project so that unforeseen delays and risks can be avoided in the future.

2.4.1 Resources for Software Cost Estimation

Various inputs are required to develop software as per user the specifications. These inputs are in the form of human resources, environmental resources, and reusable software resources as shown in Figure 2.6. Each resource is specified with four characteristics namely, description of resources, statement of availability, time when resources are required, and duration for which they are used.

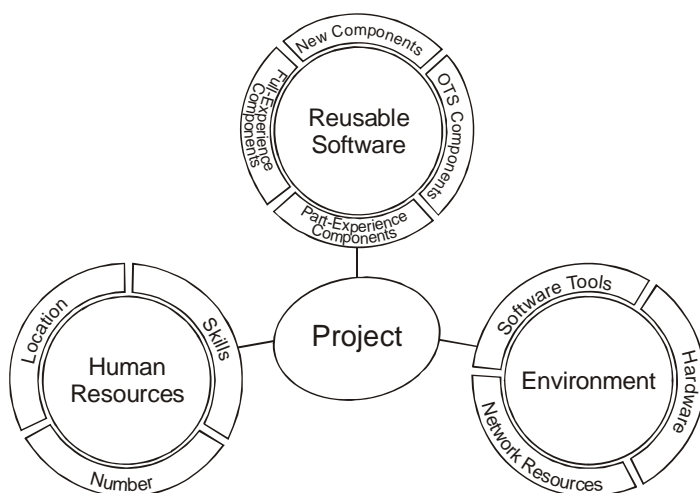


Figure 2.6 Project Resources

NOTES

(a) Human Resources: Human resources are one of the most important resources required for the successful completion of a project. For small projects, individuals are capable of performing many tasks. However, for large projects, individuals perform only a single task depending on their specialisation. These individuals can act as analysts, programmers, or testers. Also, software project team involved in development process can be geographically spread out across different locations. Hence, it is necessary to specify the location of human resources.

(b) Environmental Resources: In order to accomplish user requirements in a software project, different types of hardware and software resources are used. These resources are incorporated in an environment known as **Software Engineering Environment (SEE)**. Hardware is needed to support software, which is required to produce desired outputs. While developing software, developer involved in different phases of software development may require access to SEE. Hence, it is the responsibility of the project planner to prescribe the time in which these resources can be used and ensure that these resources are available. Also, the project planner must specify each hardware element to be used.

(c) Reusable Software Resources: While developing software components, the developers should emphasise on the concept of reusability. This is because reusing components, ideas, and process helps the developers to save time and effort when developing a project. Also, since software projects have rigid time constraints, the reuse of software components can lead to early completion of the project. Note that software components to be reused are indexed for easy reference, standardised for ease of use, and validated for system integration.

Generally, four types of reusable software resources, which are commonly used are listed below:

- **Off-the-shelf components:** The components to be used in existing projects are acquired from a third party or from the previously developed software. Similarly, commercial off the shelf (COTS) components can be purchased from a third party. These components can be readily used in the current project as they are completely validated.
- **Full-experience components:** The components (specifications, design, or code) that have been used in a previous project can be easily used in the current project if the components of both the systems are similar to each other.
- **Partial-experience components:** The components that have been used in a previous project are similar to the components of the current project but require significant modification. Thus, components in the current project involve risks as they can be reused only after certain modification.
- **New components:** The components, which are specially developed for existing software, are known as new components. These components are developed to cater to the need of the current project.

2.4.2 Software Product Cost Factors

To achieve reliable cost estimates, several factors that influence the cost of developing a software product are taken into consideration. These factors are listed below:

- **Experience in application domain:** In a software project, developer works in an application domain, which comprises of software and hardware technologies that are used to develop the project. If the developer is familiar with the programming language, operating system and hardware used in a project, then the cost of developing the project will be less. This is because they 'know' the application domain and do not have to undergo training. Note that experience of software developer plays a vital role in determining the cost of the project.
- **Product complexity:** Generally, software is categorised into three parts, namely application programs, utility programs, and system programs. **Application programs**

NOTES

(like Microsoft word, Microsoft excel, and so on) can be defined as a program that performs specific functions directly for the end user. **Utility programs** (like compiler, linker, and loader) can be defined as programs, which perform functions, such as file copying, sorting, merging, memory dump analysis, and so on. **System program** can be defined as a program that implements high-level functionality of an operating system. The cost of software project increases with the level of complexity. There are three levels of product complexity namely, organic, semi-detached, and embedded programs, which correspond to the application programs, utility programs, and system programs respectively.

- **Project size:** Size of the project is an important criterion for estimating the cost of a software project. A large sized project consumes more resources than smaller projects, hence are more costly. According to an equation given by Boehm, the rate of increase in required effort grows with the number of lines of code (LOC) at an exponential rate slightly greater than 1. As effort increases, the cost of software also increases.
- **Available time:** Time available to develop a software project according to the user requirements is an important factor to determine the project cost. In some cases, software projects require more resources and effort if development time is decreased from the allocated time thus leading to an increase in the cost of the project.
- **Programmer ability:** Software project cost is also dependent on the ability of the programmers who are involved in the software development. Efficient software programmers bring down the cost, whereas inefficient programmers need to be trained, which in turn increases the cost of the project. Note that, on a large project, the differences between individual programmers tend to 'average out'. However, in a small project, difference between the programmers' abilities can affect the software project cost considerably. Also, programmers' productivity is influenced by the ease of use and access to the hardware devices, which in-turn influences the cost of a software project. It is observed that the number of LOC written per day by a programmer is largely dependent on the programming language used.
- **Level of technology:** In a software development project, level of technology also helps in determining the cost of a project. Technology involves programming practices, hardware and software tools, and other supporting infrastructure that are used during the software project. Modern practices, such as system analysis and design techniques, design notations, and technical reviews substantially affect the project cost. Also, software tools like compiler, debugger, and automated verification tools have a considerable influence while estimating the cost.
- **Required level of reliability:** Reliability is generally expressed in terms of accuracy, robustness, consistency, and completeness of the source code. The cost estimates of the software product depend on the level of analysis, design, implementation, and verification effort that must be used to ensure high reliability. A considerable level of reliability should be established during the planning phase by considering the cost of software failure. In some cases, these failures may lead to financial losses and inconvenience to the user. It is observed that there exist *five* categories of reliability (see Table 2.2), which are rated against a numeric value called **effort multiplier**.

Table 2.2 Development Effort Multiplier for Software Reliability

Category	Effect of Failure	Effort Multiplier
Very Low	Slight inconvenience	0.75
Low	Losses easily recovered	0.88
Nominal	Moderately difficult to recover losses	1.00
High	High financial loss	1.15
Very High	Risk to human life	1.40

Check Your Progress

7. Define cost estimation.
8. Explain the importance of human resources for successful completion of a software project.
9. How does project size affect cost of a software project?

NOTES

2.5 SOFTWARE COST ESTIMATION PROCESS

To lower the cost of conducting business, identify and monitor cost and schedule risk factors, and to increase the skills of key staff members, software cost estimation process is followed. This process is responsible for tracking and refining cost estimate throughout the project life cycle. This process also helps in developing a clear understanding of the factors which influence software development costs.

Cost of estimating software varies according to the nature and type of the product to be developed. For example, the cost of estimating an operating system will be more than the cost estimated for an application program. Thus, in the software cost estimation process, it is important to define and understand the software, which is to be estimated.

In order to develop a software project successfully, cost estimation should be well planned, review should be done at regular intervals, and process should be continually improved and updated. The basic steps required to estimate cost are shown in Figure 2.7.

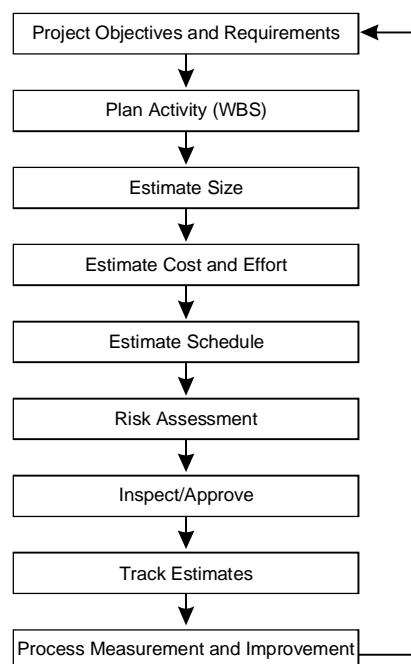


Figure 2.7 Software Cost Estimation Process

(a) Project Objectives and Requirements: In this phase, the objectives and requirements for the project are identified, which is necessary to estimate cost accurately and accomplish user requirements. The project objective defines the end product, intermediate steps involved in delivering the end product, end date of the project, and individuals involved in the project.

This phase also defines the constraints/limitations that affect the project in meeting its objectives. Constraints may arise due to the factors listed below:

- Start date and completion date of the project.
- Availability and use of appropriate resources.
- Policies and procedures that require explanations regarding their implementation.

Project cost can be accurately estimated once all the requirements are known. However, if all requirements are not known, then the cost estimate is based only on the known requirements. For example, if software is developed according to the incremental development model, then the cost estimation is based on the requirements that have been defined for that increment.

(b) Plan Activities: Software development project involves different set of activities, which helps in developing software according to the user requirements. These activities are performed in fields of software maintenance, software project management, software quality assurance, and software configuration management. These activities are arranged in the work breakdown structure according to their importance.

Work breakdown structure (WBS) is the process of dividing the project into tasks and ordering them according to the specified sequence. WBS specifies only the tasks that are performed and not the process by which these tasks are to be completed. This is because WBS is based on requirements and not the manner in which these tasks are carried out.

(c) Estimating Size: Once the WBS is established, product size is calculated by estimating the size of its components. Estimating product size is an important step in cost estimation as most of the cost estimation models usually consider size as the major input factor. Also, project managers consider product size as a major technical performance indicator or productivity indicator, which allows them to track a project during software development.

(d) Estimating Cost and Effort: Once the size of the project is known, cost is calculated by estimating effort, which is expressed in terms of person-month (PM). Various models (like COCOMO, COCOMO II, expert judgement, top-down, bottom-up, estimation by analogy, Parkinson's principal, and price to win) are used to estimate effort. Note that for cost estimation, more than one model is used, so that cost estimated by one model can be verified by another model.

(e) Estimating Schedule: Schedule determines the start date and end date of the project. Schedule estimate is developed either manually or with the help of automated tools. To develop a schedule estimate manually, a number of steps are followed, which are listed below:

1. The work breakdown structure is expanded, so that the order in which functional elements are developed can be determined. This order helps in defining the functions, which can be developed simultaneously.
2. A schedule for development is derived for each set of functions that can be developed independently.
3. The schedule for each set of independent functions is derived as the average of the estimated time required for each phase of software development.
4. The total project schedule estimate is the average of the product development, which includes documentation and various reviews.

Manual methods are based on past experience of software engineers. One or more software engineers, who are experts in developing application, develop an estimate for schedule. However, automated tools (like COSTAR, COOLSOFT) allow the user to customise schedule in order to observe the impact on cost.

(f) Risk Assessment: Risks are involved in every phase of software development therefore, risks involved in a software project should be defined and analysed, and the impact of risks on the project costs should also be determined. Ignoring risks can lead to adverse effects, such as increased costs in the later stages of software development. In the cost estimation process, four risk areas are considered, which are listed in Table 2.3.

NOTES

Table 2.3 Risk Resulting from Poor Software Estimates**NOTES**

Risk Area	Factor Associated with Risk
Size of the software project	Software developers are always optimistic while estimating the size of the software. This often results in underestimation of software size, which in turn can lead to cost and schedule overruns.
Development environment and process stability	An inadequate or unstable development environment can result in poor estimate of cost and schedule.
Staff skills	Misalignment of skills to tasks can result in inaccurate cost and schedule estimates. This can also result in poor estimates of project staffing requirements.
Change in requirements	Requirements of a software project can change during any phase of software development. However, unconstrained change of requirements results in changing project goals that can result in customer dissatisfaction, and cost and schedule overruns.

(g) Inspect and Approve: The objective of this phase is to inspect and approve estimates in order to improve the quality of an estimate and get an approval from top-level management. The other objectives of this step are listed below:

- Confirm the software architecture and functional WBS.
- Verify the methods used for deriving the size, schedule, and cost estimates.
- Ensure that the assumptions and input data used to develop the estimates are correct.
- Ensure that the estimate is reasonable and accurate for the given input data.
- Confirm and record the official estimates for the project.

Once the inspection is complete and all defects have been removed, project manager, quality assurance group, and top-level management sign the estimate. Inspection and approval activities can be formal or informal as required but should be reviewed independently by the people involved in cost estimation.

(h) Track Estimates: Tracking estimate over a period of time is essential, as it helps in comparing the current estimate to previous estimates, resolving any discrepancies with previous estimates, comparing planned cost estimates and actual estimates. This helps in keeping track of the changes in a software project over a period of time. Tracking also allows the development of a historical database of estimates, which can be used to adjust various cost models or to compare past estimates to future estimates.

(i) Process Measurement and Improvement: Metrics should be collected (in each step) to improve the cost estimation process. For this, two types of process metrics are used namely, process effective metrics and process cost metrics. The benefit of collecting these metrics is to specify a reciprocal relation that exists between the accuracy of the estimates and the cost of developing the estimates.

Check Your Progress

10. Why is software cost estimation process needed?
11. Explain the significance of estimating size of a software project.
12. Explain the importance of tracking estimates.

- **Process effective metrics:** Keeps track of the effects of cost estimating process. The objective is to *identify* elements of the estimation process, which enhance the estimation process. These metrics also identify those elements which are of little or no use to the planning and tracking processes of a project. The elements that do not enhance the accuracy of estimates should be isolated and eliminated.
- **Process cost metrics:** Provides information about implementation and performance cost incurred in the estimation process. The objective is to *quantify* and identify different ways to increase the cost effectiveness of the process. In these metrics, activities that cost-effectively enhance the project planning and tracking process remain intact, while activities that have negligible effect on the project are eliminated.

2.6 DECOMPOSITION TECHNIQUES

Software cost estimation is a form of problem solving and in most cases, the problems to be solved are too complex to be considered in a single form. Therefore, the problem is decomposed into components in order to achieve an accurate cost estimate. Two approaches are mainly used for decomposition namely, **problem-based estimation** and **process-based estimation**. However, before estimating cost, project planner should establish an estimate of the software size, which is referred to as quantitative outcome of the software project.

Software Sizing: Before estimating cost, it is necessary to estimate the accurate size of software. This is a cumbersome task as many software are of large size. Therefore, software is divided into smaller components to estimate size. This is because it is easier to calculate size of smaller components, as the complexity involved in them is less than the larger components. These small components are then added to get an overall estimate of software size.

Various approaches can be followed for estimating size. These include direct and indirect approaches. In **direct approach**, size can be measured in terms of lines of code (LOC) and in an **indirect approach**, size can be measured in terms of functional point (FP). Note that the accuracy of size estimates depends on many parameters, which are listed below:

- The degree to which the size of the software has been properly estimated.
- The ability to convert size estimate into human effort, calendar time and money.
- The degree to which the ability of a software team is reflected by the software plan.
- The stability of product requirements and environment that supports the development process.

It has been observed that an estimate of the project's cost is as good as the estimate of its size. In estimating cost, size is considered as the first problem faced by the project planner. This problem is commonly known as **software-sizing problem**. In order to solve this problem, various approaches are followed, which are listed below:

- **Fuzzy logic sizing:** To implement this approach, the planner must identify the application type and its magnitude on a quantitative scale. The magnitude is then refined within the original range.
- **Function point sizing:** This approach is used for measuring functionality delivered by the software system. Function points are derived with the help of empirical relationship, which is based on countable measures of software information domain and assessment of software complexity.
- **Standard component sizing:** Generally, software comprises of a number of standard components, which are common to a particular application only. Standard components can be modules, screens, reports, lines of code, and so on. In cost estimation process, the number of occurrence of each component is estimated and then the historical data of the project is used to determine the delivered size of each standard component.
- **Change sizing:** When an already existing project is modified in order to use it in the new project, this approach is followed. The number and type of modifications that should be accomplished in the existing project are estimated.

***Note:** It is easier to perform size estimation than cost estimation because components costs cannot be added together (since other costs, such as integration costs are also involved while developing a system). Therefore, size is used as a key parameter by estimation models.*

2.6.1 Problem-based Estimation

Lines of code and functional point are described as a measure from which productivity metrics can be calculated. During software project estimation, lines of code (LOC) and function point (FP) are used in two different ways as given below:

NOTES

NOTES

- As an estimation variable to size each element of the software.
- As baseline metric gathered from the previous projects and used with estimation variables to develop cost and effort projections.

(a) Line of Code: One of the most commonly used software size metric is line of code, which is highly dependent on the programming language. LOC can be defined as the number of delivered lines of code in software excluding the comments and blank lines.

LOC depends on the programming language chosen for the project. For example, in assembly language, lines of code will be comparatively higher than the lines of code written in any high-level language (like C++, Java). However, the exact number of lines of code can only be determined after the project is complete since less information about the project is available at the early stages of development.

For determining LOC, certain guidelines are followed, which are listed below:

- One line of code is for one logical line of code.
- Lines of code that are delivered as a part of software are included and test drivers, test stubs, and other support software are excluded.
- Software code written by the software developer is included, while code created by the application generators is excluded.
- Declarations in the programs are counted as lines of code.
- Comments in the programs are not counted as lines of code.

Using historical data, project planner estimates three values for each size. These values are optimistic (S_{opt}), most likely (S_m), and pessimistic (S_{pess}). An expected value (S) can then be computed by the following equation:

$$S = (S_{opt} + 4S_m + S_{pess})/6 \quad \dots(1)$$

Example of LOC: In this example, software comprises of six functions namely, user interface, word processing, file storage and retrievals, database management, word processor, and peripheral control.

To estimate size of each and every function in terms of LOC, developer should determine size of each function in terms of optimistic, most likely, and pessimistic values using equation (1). For instance, in Table 2.4, user interface function's expected size (s) is calculated as follows:

$$S = (2200 + 4 \times 1800 + 1400)/6 = 1800$$

In Table 2.4, size of the software in terms of LOC is 12984.

Table 2.4 Estimating Size

Function	Pessimistic (S_{pess})	Most likely (S_m)	Optimistic (S_{opt})	Expected Size (S)
User Interface	1400	1800	2200	1800
Word processing	1800	2500	3100	2483
File storage and retrievals	1700	2200	2500	2167
Database management	2400	3100	4200	3167
Word processor	1200	1600	2300	1650
Peripheral control	1400	1700	2100	1717
Total estimated lines of code				12984

(b) Function Point Function point metric is used to measure the functionality delivered by the system. Function point estimates can help in estimating effort required to design, code and test software, predict the number of errors, and forecast the number of components used in the system.

Function point is derived using an empirical relationship, which is based on the measure of software information domain value and software complexity. Software complexity can be classified in terms of simple, average, and complex levels. Information domain value can be defined as a combination of all the points listed below:

- **Number of external inputs (EI):** Users and other applications act as a source of external inputs and provide distinct application oriented data or information.
- **Number of external outputs (EO):** Each external output provided by the application provides information to the user. External outputs refer to reports, screens, error message, and so on. Individual data items in reports or screens are not counted separately.
- **Number of external inquiries (EQ):** External inquiries are defined as online input that helps to generate immediate response in the form of online output. Here, each distinct inquiry is counted separately.
- **Number of internal logical files (ILF):** Logical grouping of data that resides within the application boundary, such as master file as a part of database, is known as internal logical files. These files are maintained through external inputs.
- **Number of external interface files (EIF):** Logical grouping of data that resides external to the application, such as data files on tape or disk, is known as external interface file. External interface files provide data, which can be used by the application.

Once all the information regarding information domain value is collected (as listed in Table 2.5), complexity value for each count is determined. Organization using FP method develops criteria, which helps in determining whether a particular entry is simple, average, or complex. A weighting factor in terms of numeric value is assigned for each level of complexity.

Table 2.5 Computing Function Points

Information Domain Value	Count		Simple	Weighting Factor Average	Complex		
EI	<input type="text"/>	×	3	4	6	=	<input type="text"/>
EO	<input type="text"/>	×	4	5	7	=	<input type="text"/>
EQ	<input type="text"/>	×	3	4	7	=	<input type="text"/>
ILF	<input type="text"/>	×	7	10	15	=	<input type="text"/>
EIF	<input type="text"/>	×	5	7	10	=	<input type="text"/>

Functional points in software are estimated by the following equation:

$$FP = \text{count total} \times [0.65 + 0.01 \times \Sigma (f_i)] \quad \dots(2)$$

where,

Count total is the sum of all the FP entries.

f_i ($i = 1$ to 14) are value adjustment factors.

The value adjustment factors are based on the response to 14 questions, which are listed below:

1. Is reliable backup and recovery required by the system?
2. Is data communication required to transfer the information?
3. Do distributed processing functions exist?

NOTES

NOTES

4. Is performance vital?
5. Does the system run under immensely utilised operational environment?
6. Is on-line data entry required by system?
7. Is it possible for the on-line data entry (that requires the input transaction) to be built over multiple screens or operations?
8. Is updation of internal logical files allowed on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code reusable?
12. Does design include conversion and installation?
13. Does system design allow multiple installations in different organizations?
14. Is the application easy to use and does it facilitate changes?

The above-mentioned questions are answered using a scale of 0 to 5 (see Figure 2.8) where 0 refers to not important and 5 considered as essential. The constant values in equation (2) and the weighting factors in Table 2.5 are determined based on the information domain.

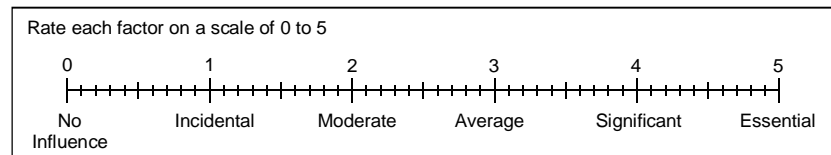


Figure 2.8 Rating of Value Adjustment Factors

Example of Function Point: To estimate size in terms of function point, first FP count should be determined, which is calculated by the following equation:

$$\text{FP count} = \text{Count} \times \text{Weighting factor (Average)} \quad \dots(3)$$

For instance, in Table 2.6 FP count for number of user inputs (measurement parameter) is calculated as follows:

$$\text{FP count} = 22 \times 4 = 88$$

Table 2.6 Computing Function Points

Measurement parameter	Count		Simple	Average	Complex		FP count
Number of user inputs	22	×	3	4	6	=	88
Number of user outputs	15	×	4	5	7	=	75
Number of user inquiries	26	×	3	4	6	=	104
Number files	6	×	7	10	15	=	60
Number of external interfaces	2	×	5	7	10	=	14
Count Total							341

After determining count for each parameter and calculating count total, 14 other parameters are considered, which are listed in Table 2.7.

Table 2.7 Value Adjustment Factors

Factors	Value
Backup and recovery	3
Data communications	1
Distributed processing	1
Performance critical	3
Existing operating environment	2
On-line data entry	3
Input transactions over multiple screens	4
Master files updated on-line	2
Information domain value complex	4
Internal processing complex	4
Code design for reuse	3
Conversions/installation in design	2
Multiple installations	4
Application design for change	4
Value adjustment factor	40

NOTES

To calculate the function point, use equation (1):

$$FP = 341 \times [0.65 + (0.01 \times 40)] = 358$$

Note: FP-based estimation is more complex than LOC.

2.6.2 Process-based Estimation

In software project development, a process is followed to accomplish objectives of the project. Commonly used technique for estimating the effort in a software project is to base the estimate on the process, which will be used. For this, the process is decomposed into smaller set of tasks, such as analysis, design, coding, and so on. Once these tasks are identified, effort required to accomplish each task is estimated.

In process-based estimation, software functions are derived from project scope and for each function, a series of activities are performed. These activities are in the form of customer communication, planning, risk analysis, engineering, and so on. The project planner then combines the functions and activities together to estimate the effort, which is required to accomplish each activity for each function. Next, with each process activity, average labour rates (cost per unit effort) are applied. The labour rate varies from task to task. For example, top-level management activities are more expensive than the activities performed by the junior staff in the initial stages of performing the framework activities.

Example for Process Based Estimation: The software considered for process-based estimation is divided into seven functions. For all the seven functions, a set tasks and activities are identified as listed in Table 2.8. Once all functions and activities are identified, effort required to accomplish each software activity for each software function is estimated. Lastly, effort for each function and activities is calculated.

Table 2.8 Process based Estimation

NOTES

Activity	Customer communication	Planning	Risk analysis	Engineering		Construction release		Total
Task				Analysis	Design	Code	Test	
Function 1				0.40	1.50	0.40	4.00	6.30
Function 2				0.65	2.00	0.60	1.00	4.25
Function 3				0.40	3.00	1.00	2.00	6.40
Function 4				0.40	2.00	1.00	2.50	5.90
Function 5				0.40	2.00	0.75	2.50	5.65
Function 6				0.15	1.00	0.50	2.50	4.15
Function 7				0.40	1.00	0.50	1.00	2.90
Total	0.25	0.25	0.25	2.80	12.50	4.75	15.50	36.00

The average labour rate available for this example is Rs 50000 per month, and based on Table 2.8 estimated effort is 36 person-month. Considering these two factors, the total estimated project cost is Rs 1800000. Note that if required, labour rate can be linked with each framework activity or software engineering activity and the labour rate can be computed independently.

2.7 COST ESTIMATION MODELS

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

- **Algorithmic models:** Estimation in these models is performed with the help of mathematical equations, which are based on historical data or theory. In order to estimate cost 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 cost of software project. The various non-algorithmic cost estimation models are expert *judgement*, *estimation by analogy*, and *price to win*.

Note: We will discuss algorithmic models only.

2.7.1 Constructive Cost Model

In the early 80's, Barry Boehm developed a model called COCOMO (**CO**nstructive **CO**st **MO**del) to estimate total effort required to develop a software project. 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, cost of development, management, and other support tasks are included. However, cost of secretarial and other staff are excluded. In this model, size is measured in terms of thousand of delivered lines of code (KDLOC).

In order to estimate effort accurately, COCOMO model divides projects into three categories listed below:

Check Your Progress

13. Why are decompositions techniques needed?
14. Mention the parameters on which the accuracy of size estimates depends.
15. Explain the term lines of code.

NOTES

- **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, which are less demanding. Most people involved in these projects have thorough understanding of how the software under development contributes in achieving the organization 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 2.9.

Table 2.9 Advantages and Disadvantages of COCOMO Model

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ 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 needs and environment. 	<ul style="list-style-type: none"> ▪ 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, which is not always available. ▪ Excludes overhead cost, travel cost and other incidental cost.

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

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

$$E = A \times (\text{size})^B \quad \dots(5)$$

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 2.10.

Table 2.10 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):

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

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

NOTES

(b) Intermediate Model: In 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 below:

1. Calculate an initial estimate of development effort by considering the size in terms of KDLOC.
2. Identify a set of 15 parameters, which are derived from attributes of the current project. All these parameters are rated against a numeric value, called **multiplying factor**. 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.

To understand the above-mentioned steps properly, let us consider an example. For simplicity reasons, an organic project whose size is 45 KDLOC is considered. In intermediate model, the values of constants (A and B) are listed in Table 2.11. To estimate total effort in this model, a number of steps are followed, which are listed below:

1. An initial estimate is calculated with the help of effort equation (5). This equation shows the relationship between size and the effort required to develop a software project. This relationship is given by the following equation:

$$E_i = A \times (\text{size})^B \quad \dots(6)$$

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 2.11.

Table 2.11 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 (6) and the value of constant for organic project, initial effort can be calculated as follows:

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

1. 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 2.12, 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 2.12 Effort Multipliers for Cost Drivers

Cost Drivers	Description	Very Low	Low	Rating 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	

NOTES

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 2.13, EAF is calculated as:

$$0.8895 (1.15 \times 0.85 \times 0.91 \times 1.00).$$

Table 2.13 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:

$$\text{Total effort} = \text{EAF} \times E_i$$

For this example, the total effort will be 155 PM.

(c) 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 2.14. Note that multiplying factors yield better estimates because the cost driver ratings differ during each phase.

NOTES

Table 2.14 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, 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 (6):

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

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,

$$E = 174 \times 1 = 174 \text{ PM}$$

2.7.2 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. 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 above given equation, there are two independent parameters namely, an estimate of size in KDLOC and project duration in calendar months or years.

Check Your Progress

16. Why are cost estimation models used?
17. Define non-algorithmic models.

2.8 LET US SUMMARIZE

1. Project planning is an organized and integrated management process, which focuses on activities required for successful completion of the software project.
2. 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.
3. 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.
4. Project plan stores the outcomes of a project planning. It includes various plans, such as quality assurance plan, verification and validation plan, configuration management plan, maintenance plan, and staffing plan.
5. Project scheduling is concerned with determining the time limit required to complete the project. An appropriate project schedule aims to complete the project on time, and also helps in avoiding additional cost that is incurred when software is not developed on time.
6. Cost estimation is the process of approximating the costs involved in the software project. Cost estimation should be done before software development is initiated since it helps the project manager to know about resources required and the feasibility of the project.
7. Various inputs are required to develop software as per user specifications. These inputs are in the form of human resources, environmental resources, and reusable software resources.
8. Software cost estimation process is followed to lower the cost of conducting business, identify and monitor cost and schedule risk factors, and increase the skills of key staff members. In order to develop software project successfully, cost estimation should be well planned, reviews should be done at regular intervals, and process should be continually improved and updated.
9. The steps required to accomplish software estimation are – project objectives and requirements, plan activities, estimate size, estimate cost and effort, estimate schedule, risk assessment, inspect and approve, tracking estimates, and process measurement and improvement.
10. Software cost estimation is a form of problem solving and in most cases the problems to be solved are too complex to be considered in a single form. Therefore, the problem is decomposed into components in order to achieve an accurate cost estimate. Two approaches are mainly used for decomposition namely, problem-based estimation and process based estimation.
11. One of the most commonly used software size metric is line of code, which is highly dependent on the programming language. LOC can be defined as the number of delivered lines of code in software excluding the comments and blank lines.
12. Function point metric is used to measure the functionality delivered by the system. Function point estimates can help in estimating effort required to design, code and test

NOTES

NOTES

software, predict the number of errors, and forecast the number of components used in the system.

13. In software project development, a process is followed to accomplish objectives of the project. Commonly used technique for estimating the effort in a software project is to base the estimate on the process, which will be used. For this, the process is decomposed into smaller set of tasks, such as analysis, design, coding, and so on.
14. Estimation models use derived formulas to predict effort as a function of LOC or FP. Various estimation models are used to estimate cost of a software project. In these models, cost of software project is expressed in terms of effort required to develop the software successfully.
15. In the early 80's, Barry Boehm developed a model called COCOMO to estimate total effort required to develop a software project. COCOMO model is commonly used as it is based on the study of already developed software projects. To estimate effort accurately, COCOMO model divide projects into three categories, namely organic projects, embedded projects, and semi-detached projects.
16. In order to estimate effort in a software project, software equation assumes specific distribution of efforts over the useful life of the project.

2.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. 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. Project planning is an organized and integrated management process, which focuses on activities required for successful completion of the project.
2. Project 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 software by hardware, memory, and so on. **Interfaces** describe the interaction of software components (like modules and functions) with each other.
3. Project planning process comprises of several activities, which are essential for carrying out a project systematically. These activities refer to the series of tasks, which 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.
4. Skills assessment provides information, which 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.
5. It is essential to perform project scheduling to effectively manage the tasks of the 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 another.
6. Unrealistic deadline refers to the scenario when the time allocated for completing a project is impractical and not according to the effort required for it. Generally, this situation arises when deadline is established by inexperienced individual(s) or without the help of project management team.

7. Cost estimation is the process of approximating the costs involved in the software project. Cost estimation should be done before software development is initiated since it helps the project manager to know about resources required and the feasibility of the project.
8. Human resources are one of the most important resources required for the successful completion of a project. For small projects, individuals are capable of performing many tasks. However, for large projects, individuals perform only a single task depending on their specialization.
9. Size of the project is an important criterion for estimating the cost of a software project. A large sized project consumes more resources than smaller projects, hence are more costly. According to an equation given by Boehm, the rate of increase in required effort grows with the number of lines of code (LOC) at an exponential rate slightly greater than 1. As effort increases, the cost of software also increases.
10. To lower the cost of conducting business, identify and monitor cost and schedule risk factors, and to increase the skills of key staff members, software cost estimation process is followed. This process is responsible for tracking and refining cost estimate throughout the project life cycle. This process also helps in developing a clear understanding of the factors which influence software development costs.
11. Product size is calculated by estimating the size of its components. Estimating product size is an important step in cost estimation as most of the cost estimation models usually consider size as the major input factor. Also, project managers consider product size as a major technical performance indicator or productivity indicator, which allows them to track a project during software development.
12. Tracking estimates over a period of time is essential, as it helps in comparing the current estimate to previous estimates, resolving any discrepancies with previous estimates, comparing planned cost estimates and actual estimates. This helps in keeping track of the changes in a software project over a period of time. Tracking also allows the development of a historical database of estimates, which can be used to adjust various cost models or to compare past estimates to future estimates.
13. Software cost estimation is a form of problem solving and in most cases, the problems to be solved are too complex to be considered in a single form. Therefore, there arises need for decompositions techniques so that the problem can be decomposed into components in order to achieve an accurate cost estimate.
14. The accuracy of size estimates depends on following parameters:
 - The degree to which the size of the software has been properly estimated.
 - The ability to convert size estimate into human effort, calendar time and money.
 - The degree to which the ability of a software team is reflected by the software plan.
 - The stability of product requirements and environment that supports the development process.
15. Lines of code (LOC) can be defined as the number of delivered lines of code in software excluding the comments and blank lines. LOC depends on the programming language chosen for the project. For example, in assembly language, lines of code will be comparatively higher than the lines of code written in any high-level language (like C++, Java).
16. Estimation models use derived formulas to predict effort as a function of LOC or FP. Various estimation models are used to estimate cost of a software project. In these

NOTES

NOTES

models, cost of software project is expressed in terms of effort required to develop the software successfully.

17. In non-algorithmic models, estimation depends on the prior experience and domain knowledge of project managers. Note that these models do not use mathematical equations to estimate cost of software project.

2.10 QUESTIONS AND EXERCISES

I. Fill in the Blanks

1. The individuals included in project planning are _____ and _____.
2. _____ is the process of searching, evaluating, and establishing work relationship with the personnel required for the software project.
3. _____ describe the steps, which are followed to estimate the initial software life cycle cost.
4. The various resources used in software development are _____, environmental resources, and _____.

II. Multiple Choice Questions

1. Which one of the following business objectives ensure that the organizational objectives and requirements are accomplished in the project?
 - (a) Renew policies and processes
 - (b) Be according to schedule
 - (c) Meet user requirements
 - (d) None of the above
2. Which of the following is not included in project plan?
 - (a) Quality assurance plan
 - (b) Verification and validation plan
 - (c) Configuration management plan
 - (d) Test plan
3. Software product cost factors, include:
 - (a) Product complexity
 - (b) Available time
 - (c) Level of technology
 - (d) All the above
4. Which one of the following is not the category of the COCOMO model?
 - (a) Organic projects
 - (b) Product complexity
 - (c) Semi-detached projects
 - (d) Embedded projects

III. State Whether True or False

1. The purpose of project scheduling is to make the schedule of project management team members.
2. Software project is carried out to accomplish a specific purpose.
3. Software cost estimation does not enhance the skill level of staff members in an organization.
4. Software equation is a single variable cost estimation model.

IV. Descriptive Questions

1. Write short notes on the following:
 - (a) Maintenance plan
 - (b) Verification and validation plan
 - (c) Software equation

2. COCOMO model is based on the hierarchy of three models, namely basic model, intermediate model, and advance model. Explain them.

*Software Project Planning
and Cost Estimation*

2.11 FURTHER READING

1. Software Engineering: A Practitioner's Approach – *Roger Pressman*
2. Software Engineering – *Ian Sommerville*

NOTES

UNIT 3 SYSTEM ANALYSIS

Structure

- 3.0 Introduction
- 3.1 Unit Objectives
- 3.2 What is Software Requirement?
 - 3.2.1 Guidelines for Expressing Requirements;
 - 3.2.2 Types of Requirements
 - 3.2.3 Requirements Engineering Process
- 3.3 Feasibility Study
 - 3.3.1 Types of Feasibility;
 - 3.3.2 Feasibility Study Process
- 3.4 Requirements Elicitation
 - 3.4.1 Elicitation Techniques
- 3.5 Requirements Analysis
 - 3.5.1 Structured Analysis;
 - 3.5.2 Object-oriented Modelling;
 - 3.5.3 Other Approaches
- 3.6 Requirements Specification
 - 3.6.1 Structure of SRS
- 3.7 Requirements Validation
 - 3.7.1 Requirement Review;
 - 3.7.2 Other Requirement Validation Techniques
- 3.8 Requirements Management
 - 3.8.1 Requirements Management Process;
 - 3.8.2 Requirements Change Management
- 3.9 Case Study: Student Admission and Examination System
 - 3.9.1 Problem Statement;
 - 3.9.2 Data Flow Diagrams
 - 3.9.3 Entity Relationship Diagram;
 - 3.9.4 Software Requirements Specification Document
- 3.10 Data Dictionary
- 3.11 Let us Summarize
- 3.12 Answers to 'Check Your Progress'
- 3.13 Questions and Exercises
- 3.14 Further Reading

NOTES

3.0 INTRODUCTION

In the software development process, requirements phase is the first software engineering activity, which translates the ideas or views into a requirements document. This phase is user-dominated phase. Defining and documenting the user requirements in a concise and unambiguous manner is the first major step to achieve a high quality product.

Requirements phase encompasses a set of tasks, which helps to specify the impact of the software on the organisation, customers' needs, and how users will interact with the developed software. The requirements are the basis of system design. If 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. This is because requirements are meant for (such as users, managers, system engineers, and so on). For example, managers may be interested in knowing how the system is implemented rather than knowing the detailed features of the system. Similarly, end-users are interested in knowing whether the specified requirements meet their desired needs or not.

3.1 UNIT OBJECTIVES

After reading this unit, the reader will understand:

- What is Software Requirement?
- Feasibility study, which includes technical, operational, and economic feasibility.

NOTES

- Requirements elicitation, which is a process of collecting information about software requirements from different individuals.
- How requirements analysis helps to understand, interpret, classify, and organize the software requirements.
- Requirements document that lays a foundation for software engineering activities and is created when entire requirements are elicited and analyzed.
- Requirements validation phase where work products are examined for consistency, omissions, and ambiguity.
- Requirements management phase which can be defined as a process of eliciting, documenting, organizing and controlling changes to the requirements.

3.2 WHAT IS SOFTWARE REQUIREMENT?

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)”.

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

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

3.2.1 Guidelines for Expressing Requirements

The purpose of the requirements document is to provide a basis for the mutual understanding between users and designers of the initial definition of the software development life cycle (SDLC), including the requirements, operating environment, and development plan.

The requirements document 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 requirements document 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 requirements document needs to describe the operating environment and provide, or make reference to, a development plan.

There is no standard method to express and document the 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 listed below:

- 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.

3.2.2 Types of Requirements

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 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, namely, functional requirements, non-functional requirements, and domain requirements.

(a) 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 function requirements as “*a function that a system or component must be able to perform.*”

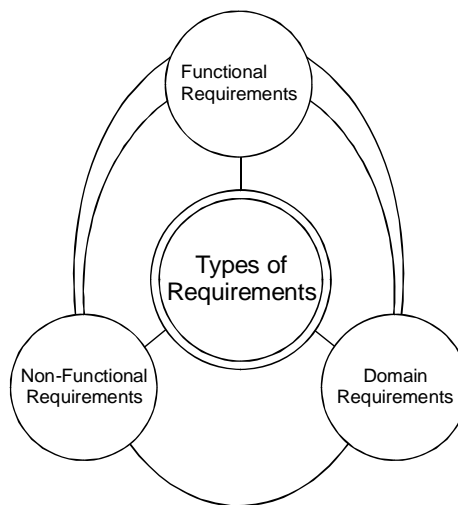


Figure 3.1 Types of Requirements

To understand functional requirements properly, let us consider an example of an online banking system, which is listed below:

- The user of the bank should be able to search the desired services from the available services.
- 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 the user can later be given an account number.

The above-mentioned 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, then they are misinterpreted by the software engineers and user requirements are not 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 in achieving 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.

NOTES

NOTES

(b) 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 related directly 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.

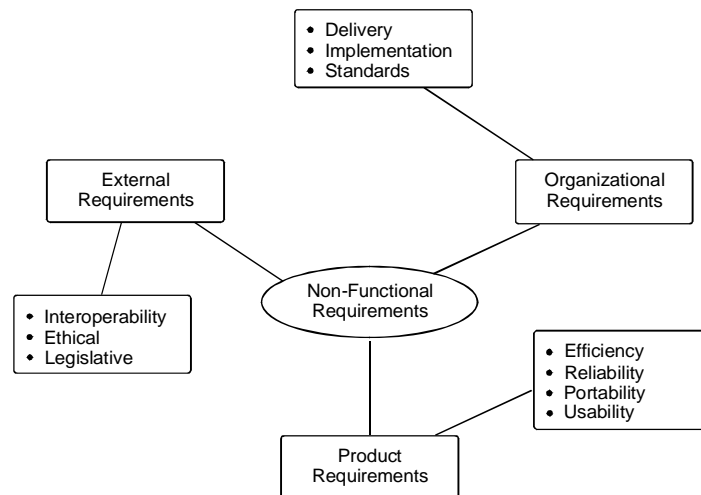


Figure 3.2 Types of Non-functional Requirements

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

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

- **Implementation requirements:** Describe requirements, such as programming language and design method.
- **Standards requirements:** Describe the process standards to be used during software development. For example, software should be developed using standards specified by ISO (International Organization for Standardization) and IEEE standards.
- **External requirements:** These requirements include all the requirements that affect the software or its development process externally. External requirements comprise of the following:
 - **Interoperability requirements:** Define the way in which different computer-based systems interact with each other in one or more organizations.
 - **Ethical requirements:** Specify the rules and regulations of the software so that they are acceptable to users.
 - **Legislative requirements:** Ensure that 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	<ul style="list-style-type: none"> ▪ Processed transaction/second ▪ User/event response time ▪ Screen refresh rate
Size	<ul style="list-style-type: none"> ▪ Amount of memory (KB) ▪ Number of RAM chips
Ease of use	<ul style="list-style-type: none"> ▪ Training time ▪ Number of help windows
Reliability	<ul style="list-style-type: none"> ▪ Mean time to failure (MTTF) ▪ Portability of unavailability ▪ Rate of failure occurrence
Robustness	<ul style="list-style-type: none"> ▪ Time to restart after failure
	<ul style="list-style-type: none"> ▪ Percentage of events causing failure ▪ Probability of data corruption on failure
Portability	<ul style="list-style-type: none"> ▪ Percentage of target-dependent statements ▪ Number of target systems

(c) **Domain Requirements:** Requirements 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 existing functional requirements. As domain requirements reflect 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, a system may comprise of 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 design as per established standards. Similarly, the requirements requested by the user, such as copyright restrictions and security mechanism for the files and documents used in the system are also domain requirements.

NOTES

NOTES

When domain requirements are not expressed clearly, it can result in various problems, 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 these requirements.
- **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.

3.2.3 Requirements Engineering Process

The requirements engineering (RE) process is a series of activities that are performed in requirements phase in order to express requirements in software requirements specification (SRS) document. This process focuses on understanding the requirement and its type so that an appropriate technique is determined to carry out the requirements engineering process. The new software developed after collecting requirements either replaces the existing software or enhances its features and functionality. For example, the payment mode of existing software can be changed from payment through hand-written cheques to electronic payment of bills.

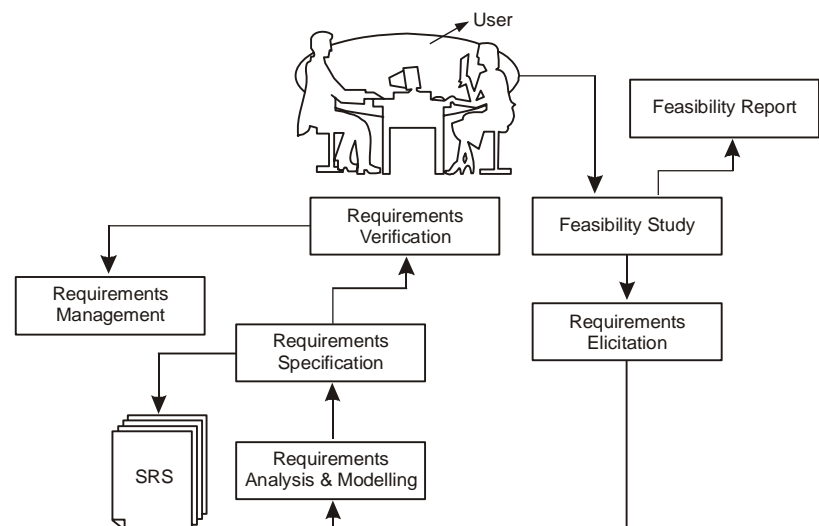


Figure 3.3 Requirements Engineering Process

In Figure 3.3, a requirements engineering process is shown, which comprises of various steps. These steps include *feasibility study*, *requirements elicitation*, *requirements analysis*, *requirements specification*, *requirements validation*, and *requirements management*.

Requirements engineering process begins with a feasibility study of the requirements. Then, requirements elicitation is performed, which focuses on gathering user requirements. After the requirements are gathered, analysis is performed, which further leads to requirements specification. The output of this is stored in the form of software requirements specification document. Next, the requirements are checked for their completeness and correctness in requirements validation. Lastly, to understand and control changes to system requirements, requirements management is performed.

Check Your Progress

1. Name the three main categories of requirements.
2. Define requirement engineering process.

3.3 FEASIBILITY STUDY

Feasibility is defined as the practical extent to which a project can be performed successfully. To evaluate feasibility, a **feasibility study** is performed, which determines whether the solution considered to accomplish the requirements is practically workable in the software or not. For this, it considers information, such as resource availability, cost estimates for software development, benefits of software to organization after it is developed, and cost to be incurred on its maintenance. The objective of feasibility study is to establish the reasons for developing software that is acceptable to users, adaptable to change, and conformable to established standards. Various other objectives of feasibility study are listed below:

- Analyze whether the software will meet organizational requirements or not.
- Determine whether the software can be implemented using current technology and within the specified budget and schedule or not.
- Determine whether the software can be integrated with other existing software or not.

3.3.1 Types of Feasibility

Various types of feasibility (see Figure 3.4) that are commonly considered include *technical feasibility*, *operational feasibility*, and *economic feasibility*.

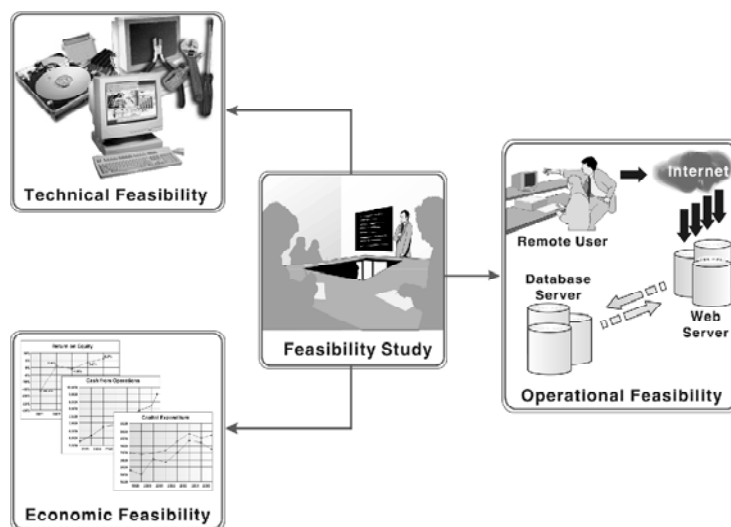


Figure 3.4 Types of Feasibility

(a) Technical Feasibility: Technical feasibility assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget. For this, software development team ascertains whether the current resources and technology can be upgraded or added in the software to accomplish specified user requirements. Technical feasibility performs the tasks listed below:

- Analyzes the technical skills and capabilities of software development team members.
- Determines whether the relevant technology is stable and established or not.
- Ascertains that the technology chosen for software development has large number of users so that they can be consulted when problems arise or improvements are required.

NOTES

NOTES

(b) Operational Feasibility: Operational feasibility assesses the extent to which the required software performs series of steps to solve business problems and user requirements. This feasibility is dependent on human resources (software development team) and involves visualizing whether or not the software will operate after it is developed and be operated once it is installed. In addition, operational feasibility performs the tasks listed below:

- Determines whether the problems proposed in user requirements are of high priority or not.
- Determines whether the solution suggested by software development team is acceptable or not.
- Analyzes whether users will adapt to new software or not.
- Determines whether the organization is satisfied by the alternative solutions proposed by software development team or not.

(c) Economic Feasibility: Economic feasibility determines whether the required software is capable of generating financial gains for an organization or not. It involves the cost incurred on software development team, estimated cost of hardware and software, cost of performing feasibility study, and so on. For this, it is essential to consider expenses made on purchases (such as hardware purchase) and activities required to carry out software development. In addition, it is necessary to consider the benefits that can be achieved by developing the software.

Software is said to be economically feasible if it focuses on the issues listed below:

- Cost incurred on software development produces long-term gains for an organization.
- Cost required to conduct full software investigation (such as requirements elicitation and requirements analysis).
- Cost of hardware, software, development team, and training.

Note: As economic feasibility assesses cost and benefits of the software, cost-benefit analysis is performed for it. Economic feasibility uses several methods to perform cost-benefit analysis, such as payback analysis, return on investment (ROI), and present value analysis.

3.3.2 Feasibility Study Process

Feasibility study comprises of various steps, which are listed below:

- 1. Information assessment:** Identifies information about whether the system helps in achieving the objectives of the organisation. In addition it verifies that the system can be implemented using new technology and within the budget. It also verifies whether the system can be integrated with the existing system
- 2. Information collection:** Specifies the sources from where information about software can be obtained. Generally, these sources include users (who will operate the software), organization (where the software will be used), and software development team (who understands user requirements and knows how to fulfil them in software).
- 3. Report writing:** Uses a *feasibility report*, which is the conclusion of the feasibility by the software development team. It includes the recommendation of whether the software development should continue or not. This report may also include information about changes in software scope, budget, schedule, and suggestion of any requirements in the system.

Figure 3.5 shows the feasibility study plan, which comprises of the various sections listed below:

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 Contacts
	1.6.1 Information
	1.6.2 Co-ordination
2.0	Management Summary
2.1	Environment
	2.1.1 Organizations Involved
	2.1.2 Input/Output
	2.1.3 Processing
	2.1.4 Security
	2.1.5 System Interaction
	2.1.6 Physical Environment
2.2	Current Functional Procedures
2.3	Functional Objectives
2.4	Performance Objectives
2.5	Assumptions and Constraints
2.6	Methodology
2.7	Evaluation Criteria
2.8	Recommendations
3.0	Proposed System
3.1	Description of Proposed System
3.2	Improvements
3.3	Time and Resource Costs
3.4	Impacts
	3.4.1 Equipment Impacts
	3.4.2 Software Impacts
	3.4.3 Organizational Impacts
	3.4.4 Operational Impacts
	3.4.5 Developmental Impacts
	3.4.6 Site or Facility Impacts
	3.4.7 Security and Privacy Impacts
3.5	Rationale for Recommendations
4.0	Alternative System
4.1	Description

NOTES

Figure 3.5 Feasibility Study Plan

- **General information:** Describes the purpose and scope of feasibility study. It also describes system overview, acronyms and abbreviations, and points of contact to be used. **System overview** provides description about the name of organization responsible for software development, system name or title, system category, operational status, and so on. **Project references** provide a list for the references used to prepare this document, such as documents relating to the project or previously developed documents that are related to the project. **Acronyms and abbreviations** provide a list of the terms that are used in this document along with their meanings. **Points of contact** provide a list of points of organizational contact with users for information and coordination. For example, users require assistance to solve problems (such as troubleshooting) and collect information, such as contact number, E-mail address, and so on.
- **Management summary:** Provides the information listed below:
 - **Environment:** Identifies the individuals responsible for software development. It provides information about input and output requirements, processing requirements of software, and the interaction of software with other software. In addition, it also identifies system security requirements and system's processing requirements.

NOTES

Check Your Progress

3. What are the objectives of feasibility study?
4. List the task are performed by operational feasibility.
5. What is the role of information assessment in feasibility study process?
6. Explain briefly functional objective and performance objective of feasibility study plan.

- **Current functional procedures:** Describes the current functional procedures of an existing system, whether automated or manual. It also includes the data flow of current system and the number of team members required to operate and maintain the software.
- **Functional objective:** Provides information about functions of the system, such as new services, increased capacity, and so on.
- **Performance objective:** Provides information about performance objectives, such as reduced staff and equipment cost, increased processing speed of software, and improved controls.
- **Assumptions and constraint:** Provides information about assumptions and constraints, such as operational life of the proposed software, financial constraints, changing hardware, software and operating environment, and availability of information and sources.
- **Methodology:** Describes the methods that are applied to evaluate the proposed software in order to reach a feasible alternative. These methods include survey, modelling, benchmarking, and so on.
- **Evaluation criteria:** Identifies the criteria, such as cost, priority, development time, and ease of system use. The criteria are applicable for the development process to determine the most suitable system option.
- **Recommendation:** Describes a recommendation for the proposed system. This includes the delays and acceptable risks.
- **Proposed software:** Describes the overall concept of the system as well as the procedure to be used to meet user requirements. In addition, it provides information about improvements, time and resource costs, and impacts. **Improvements** are performed to enhance functionality and performance of existing software. **Time and resource costs** include the costs associated with software development from its requirement to its maintenance and staff training. **Impacts** describe the possibility of future happenings and include various types of impacts, which are listed below:
 - **Equipment impacts:** Determine new equipment requirements and changes to be made in the currently available equipment requirements.
 - **Software impacts:** Specify any additions or modifications required in the existing software and supporting software to adapt to the proposed software.
 - **Organizational impacts:** Describe any changes in organization, staff, and skills requirement.
 - **Operational impacts:** Describe effects on operations, such as user operating procedures, data processing, data entry procedures, and so on.
 - **Developmental impacts:** Specify developmental impacts, such as resources required to develop databases, resources required to develop and test the software, and specific activities to be performed by user during software development.
 - **Security impacts:** Describe security factors that may influence the development, design, and continued operation of the proposed software.
- **Alternative systems:** Provide description of alternative systems, which are considered in feasibility study. It also describes the reasons for choosing a particular alternative system to develop the proposed software and the reason for rejecting other alternative systems.

3.4 REQUIREMENTS ELICITATION

Requirements elicitation (also known as **requirements capture** or **requirements acquisition**) is a process of collecting information about software requirements from different

individuals, such as users and other stakeholders. Stakeholders are individuals who are affected by the system, directly or indirectly. These include project managers, marketing people, consultants, software engineers, maintenance engineers, and user.

Various issues may arise during requirements elicitation and may cause difficulty in understanding the software requirements. Some of the problems are listed below:

- **Problems of scope:** This problem arises when the boundary of software (that is, scope) is not defined properly. Due to this, it becomes difficult to identify objectives as well as functions and features to be accomplished in software.
- **Problems of understanding:** This problem arises when users are not certain about their requirements and thus are unable to express what they require in software and which requirements are feasible. This problem also arises when users have no or little knowledge of the problem domain and are unable to understand the limitations of computing environment of software.
- **Problems of volatility:** This problem arises when requirements change over time.

Requirements elicitation uses *elicitation techniques*, which facilitate a software engineer to understand user requirements and software requirements needed to develop the proposed software.

3.4.1 Elicitation Techniques

Various elicitation techniques are used to identify the problem, determine its solution, and identify different approaches for the solution. These techniques also help the stakeholders to clearly express their requirements by stating all the important information. The commonly followed elicitation techniques are listed below:

- **Interviews:** These are conventional ways for eliciting requirements, which help software engineer, users, and software development team to understand the problem and suggest solution for the problem. For this, the software engineer interviews the users with a series of questions. When an interview is conducted, rules are established for users and other stakeholders. In addition, an agenda is prepared before conducting interviews, which includes the important points (related to software) to be discussed among users and other stakeholders. An effective interview should have the characteristics listed below:
 - Individuals involved in interviews should be able to accept new ideas. Also, they should focus on listening to the views of stakeholders related to requirements and avoid biased views.
 - Interviews should be conducted in defined context to requirements rather than in general terms. For this, users should start with a question or a *requirements proposal*.
- **Scenarios:** These are descriptions of a sequence of events, which help to determine possible future outcome before the software is developed or implemented. Scenarios are used to *test* whether the software will accomplish user requirements or not. Also, scenarios help to provide a framework for questions to software engineer about users' tasks. These questions are asked from users about future conditions (what-if) and procedure (how) in which they think tasks can be completed. Generally, a scenario comprises of the information listed below:
 - Description of what users expect when scenario starts.
 - Description of how to handle the situation when software is not operating correctly.
 - Description of the state of software when scenario ends.
- **Prototypes:** Prototypes help to clarify unclear requirements. Like scenarios, prototypes also help users to understand the information they need to provide to software development team.

NOTES

NOTES

- **Quality function deployment (QFD):** This deployment translates user requirements into technical requirements for the software. For this, QFD facilitates development team to understand what is valuable to users so that quality can be maintained throughout the software development. QFD identifies some of the common user requirements, which are listed below:

- **General requirements:** These requirements describe the system objectives, which are determined by various requirements elicitation techniques. Examples of general requirements are graphical displays requested by users, specific software functions, and so on.
- **Expected requirements:** These requirements are implicit to software, as users consider them to be fundamental requirements, which will be accomplished in the software and hence do not express them. Examples of expected requirements are ease of software installation, ease of software and user interaction, and so on.
- **Unexpected requirements:** These requirements specify the requirements that are beyond user expectations. These requirements are not requested by the user but if development team adds them to the software, users are satisfied to have the software with the additional features. An example of unexpected requirements is to have word processing software with additional capabilities, such as page layout capabilities along with the earlier features.

3.5 REQUIREMENTS ANALYSIS

IEEE defines requirements 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. Various other tasks performed using requirements analysis are listed below:

- Detect and resolve conflicts that arise due to unclear and unstated requirements.
- Determine operational characteristics of software and how it interacts with its environment.
- Understand the problem for which software is to be developed.
- Develop analysis model to analyze the requirements in the software.

Software engineers perform analysis modelling and create analysis model to provide information of ‘what’ software should do instead of ‘how’ to fulfil the requirements in software. This model emphasizes on information, such as the functions that 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 other components. The clear and complete requirements specified in analysis model help software development team to develop software according to those requirements. An analysis model is created to help the development team to assess the quality of software when it is developed. An 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 user knows the dimensions of the room, the location of doors and windows, and the available wall space. Before constructing the study room, user provides information about flooring, wallpaper, and so on to the constructor. This information helps the constructor to analyze the requirements and prepare an analysis model that describes the requirements. This model also describes what needs to be done to accomplish those requirements. Similarly, an analysis model created for software facilitates software development team to understand what is required in software and then develop it.

Check Your Progress

7. Name the various requirement elicitation techniques.
8. What kind of information is included in scenarios elicitation technique?

In Figure 3.6, analysis model connects system description and design model. System description provides information about the entire functionality of system, which is achieved by implementing software, hardware, and data. In addition, analysis model specifies the software design, in the form of design model, which provides information about software's architecture, user interface, and component level structure.

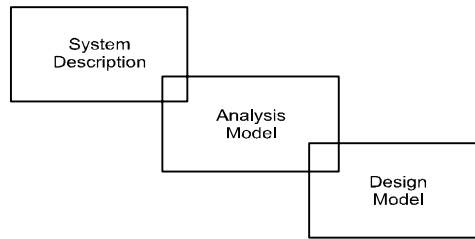


Figure 3.6 Analysis Model as Connector

NOTES

The guidelines followed while creating an analysis model are listed below:

- The model should concentrate on requirements that are present within the problem with detailed information about the problem domain. However, an analysis model should not describe the procedure to accomplish requirements in the system.
- Every element of 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 their 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 ambiguity. Due to this, analysis model comprises of *structured analysis*, *object-oriented modelling*, and *other approaches*. 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 ER modelling and several requirements specification languages and processors.

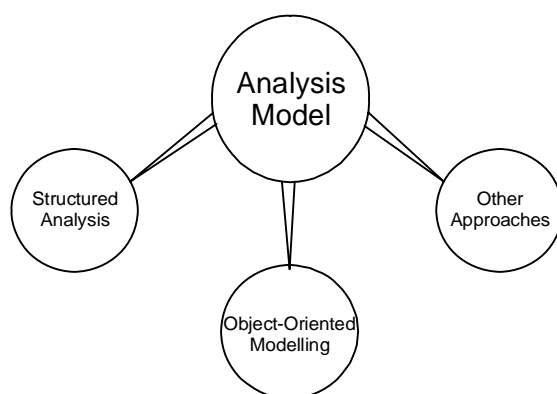


Figure 3.7 Analysis Model

3.5.1 Structured Analysis

Structured analysis is a top-down approach, which focuses on refining the problem with the help of functions performed in the problem domain and data produced by these functions. The basic principles of this approach are:

- To facilitate software engineer in order to determine the information received during analysis and to organize the information to avoid complexity of the problem.
- To provide a graphical representation to develop new software or enhance the existing software.

NOTES

Generally, structured analysis is represented using *data flow diagram*.

(a) Data Flow Diagram (DFD): IEEE defines data flow diagram (also known as **bubble chart** or **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”. DFD allows software development team to depict flow of data from one or more processes to another. In addition, DFD accomplishes the objectives listed below:


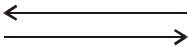
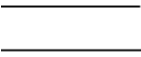

- Represents system data in hierarchical manner and with required level of detail.
- Depicts processes according to defined user requirements and software scope.

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 by 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 to the input data within the system in order to produce an output.

DFD should not be confused with a flowchart. A DFD represents the flow of data whereas flowchart depicts the flow of control. Also, a DFD does not depict the information about the procedure to be used for accomplishing the task. Hence, while making DFD, procedural details about the processes should not be shown. DFD helps a software designer to describe the transformations taking place in the path of data from input to output

DFD comprises of four basic notations (symbols), which help to depict information in a system. These notations are listed in Table 3.2.

Table 3.2 DFD Notations

Name	Notation	Description
External entity		Represents the source or destination of data within the system. Each external entity is identified with a meaningful and unique name.
Data flow		Represents the movement of data from its source to destination within the system.
Data store		Indicates the place for storing information within the system.
Process		Shows a transformation or manipulation of data within the system. A process is also known as bubble.

While creating a DFD, certain guidelines are followed to depict the data flow of system requirements effectively. These guidelines help to create DFD in an understandable manner. The commonly followed guidelines for creating DFD are listed below:

- DFD notations should be given meaningful names. For example, verb should be used for naming a process whereas nouns should be used for naming external entity, data store, and data flow.
- Abbreviations should be avoided in DFD notations.
- Each process should be numbered uniquely but the numbering should be consistent.

- DFD should be created in an organized manner so that it is easily understandable.
- Unnecessary notations should be avoided in DFD in order to avoid complexity.
- DFD should be logically consistent. For this, processes without any input or output and any input without output should be avoided.
- There should be no loops in DFD.
- DFD should be refined until each process performs a simple function so that it can be easily represented as a program component.
- DFD should be organized in a series of levels so that each level provides more detail than the previous level.
- The name of a process should be carried to the next level of DFD.
- Each DFD should not have more than six processes and related data stores.
- The data store should be depicted at the context level where it first describes an interface between two or more processes. Then, the data store should be depicted again in the next level of DFD that describes the related processes.

There are various levels of DFD, which provide detail about the input, processes, and output of a system. Note that the level of detail of process increases with increase in level(s). However, these levels do not describe the systems' internal structure or behaviour. These levels are listed below:

- **Level 0 DFD** (also known as **context diagram**): Shows an overall view of the system.
- **Level 1 DFD**: Elaborates level 0 DFD and splits the process into a detailed form.
- **Level 2 DFD**: Elaborates level 1 DFD and displays the process(s) in a more detailed form.
- **Level 3 DFD**: Elaborates level 2 DFD and displays the process(s) in a detailed form.

To understand various levels of DFD, let us consider an example of banking system. In Figure 3.8, level 0 DFD is drawn, this DFD represents how 'user' entity interacts with 'banking system' process and avails its services. The level 0 DFD depicts the entire banking system as a single process. There are various tasks performed in a bank, such as transaction processing, pass book entry, registration, demand draft creation, and online help. The data flow indicates that these tasks are performed by both the user and bank. Once the user performs transaction, the bank verifies whether the user is registered in the bank or not.

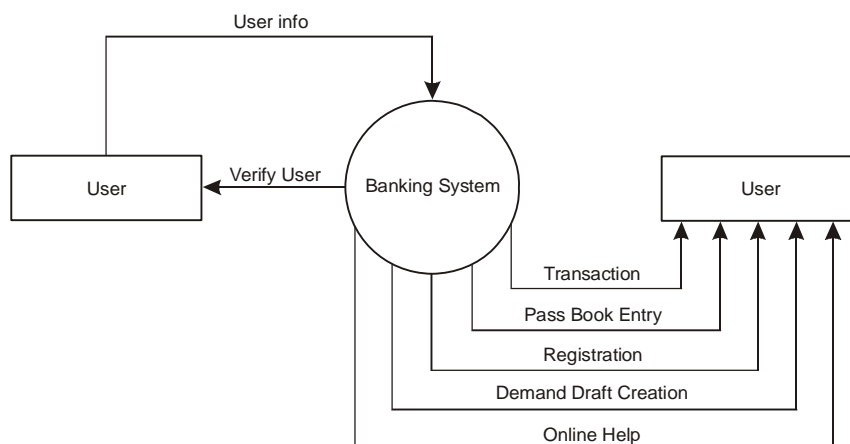


Figure 3.8 Level 0 DFD of Banking System

NOTES

NOTES

The level 0 DFD is expanded in level 1 DFD (see Figure 3.9). In this DFD, 'user' entity is related to several processes in the bank, which include 'register', 'user support', and 'provide cash'. Transaction can be performed if user is already registered in the bank. Once the user is registered, user can perform transaction by the processes, namely, 'deposit cheque', 'deposit cash', and 'withdraw cash'. Note that the line in the process symbol indicates the level of process and contains a unique identifier in the form of a number. If user is performing transaction to deposit cheque, the user needs to provide cheque to the bank. The user's information, such as name, address, and account number is stored in 'user_detail' data store, which is a database. If cash is to be deposited and withdrawn, then, the information about the deposited cash is stored in 'cash_detail' data store. User can get demand draft created by providing cash to the bank. It is not necessary for the user to be registered in that bank to have demand draft. The details of amount of cash and date are stored in 'DD_detail' data store. Once the demand draft is prepared, its receipt is provided to the user. The 'user support' process helps users by providing answers to their queries related to the services available in the bank.

Level 1 DFD can be further refined into level 2 DFD for any process of banking system that has detailed tasks to perform. For instance, level 2 DFD can be prepared to deposit cheque, deposit cash, withdraw cash, provide user support, and to create demand draft. However, it is important to maintain the continuity of information between the previous levels (level 0 and level 1) and level 2 DFD. As mentioned earlier, the DFD is refined until each process performs a simple function, which is easy to implement.

Let us consider the 'withdraw cash' process (as shown in Figure 3.9) to illustrate level 2 DFD. The information collected from level 1 DFD acts as an *input* to level 2 DFD. Note that 'withdraw cash' process is numbered as '3' in Figure 3.9 and contains further processes, which are numbered as '3.1', '3.2', '3.3', and '3.4' in Figure 3.10. These numbers represent the sublevels of 'withdraw cash' process. To withdraw cash, bank checks the status of balance in user's account (as shown by 'check account status' process) and then allots token (shown as 'allot token' process). After the user withdraws cash, the balance in user's account is updated in the 'user_detail' data store and statement is provided to the user.

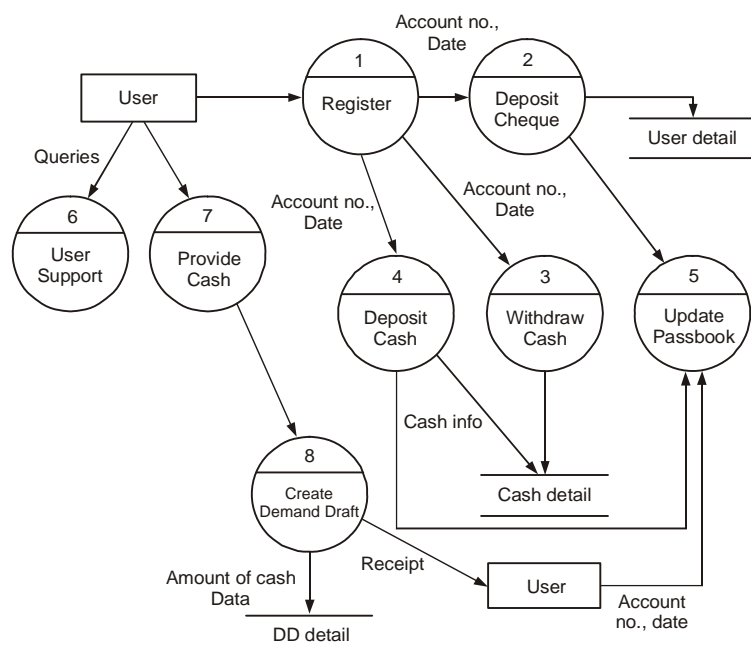


Figure 3.9 Level 1 DFD to Perform Transaction

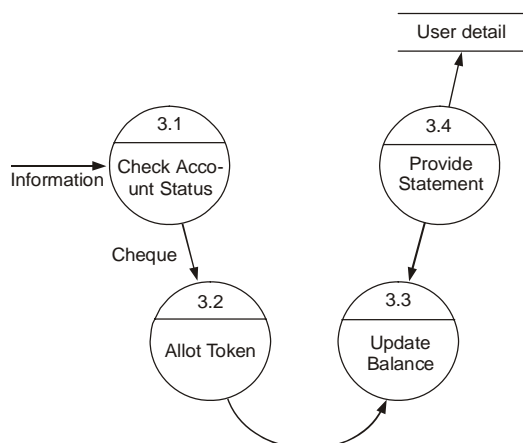


Figure 3.10 Level 2 DFD to Withdraw Cash

If a particular process of level 2 DFD requires elaboration, then this level is further refined into level 3 DFD. Let us consider the process 'check account status' (see Figure 3.10) to illustrate level 3 DFD. In Figure 3.11, this process contains further processes numbered as '3.1.1' and '3.1.2', which describe the sublevels of 'check account status' process. To check the account status, the bank fetches the account detail (shown as 'fetch account detail' process) from the 'account_detail' data store. After fetching the details, the balance is read (shown as 'read balance' process) from the user's account. Note that the requirements engineering process of DFDs continues until each process performs a function that can be easily implemented as an individual program component.

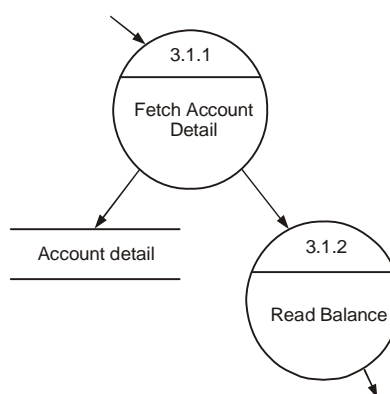


Figure 3.11 Level 3 DFD to Check Account Status

(b) Data Dictionary: Although data flow diagrams contain meaningful names of notations, they do not provide complete information about the structure of data flows. For this, data dictionary is used, which is a *repository* that stores description of data objects to be used by the software. Data dictionary stores an organized collection of information about data and their relationships, data flows, data types, data stores, processes, and so on. In addition, a data dictionary helps users to understand the data types and processes defined along with their uses. It also facilitates the validation of data by avoiding duplication of entries and provides online access to definitions to the users.

Data dictionary comprises of the source of data, which are data objects and entities. In addition, it comprises of the elements listed below:

- **Name:** Provides information about the primary name of the data store, external entity, and data flow.
- **Alias:** Describes different names of data objects and entities used.
- **Where-used/how-used:** Lists all the processes that use data objects and entities and how they are used in the system. For this, it describes the inputs to the process, output from the process, and the data store.
- **Content description:** Provides information about the content with the help of data dictionary notations (such as '=', '+', and '* *').
- **Supplementary information:** Provides information about data types, values used in variables, and limitations of these values.

NOTES

NOTES

3.5.2 Object-oriented Modelling

Now a days object-oriented approach is used to describe system requirements using prototypes. This approach is performed using object-oriented modelling (also known as **object-oriented analysis**), which analyzes the problem domain and then partitions the problem with the help of objects. An object is an entity that represents a concept and performs a well-defined task in the problem domain. For this, an object contains information of the state and provides services to entities, which are outside the object(s). The state of an object changes when it provides services to other entities.

The object-oriented modelling defines a system as a set of objects, which interact with each other by the services they provide. In addition, objects interact with users through their services so that they can avail the required services in the system.

To understand object-oriented analysis, it is important to understand the various concepts used in object-oriented environment. Some of the commonly used these concepts are listed in Table 3.3.

Table 3.3 Object-Oriented Concepts

Object-Oriented Concepts	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 methods and services, provides a means to modify the state of a class.
Super-class	Also known as base class, is a generalization of a collection of classes related to it.
Sub-class	A specialization of superclass and 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.

Generally, it is considered that object-oriented systems are easier to develop and maintain. Also, it is considered that the transition from object-oriented analysis to object-oriented design can be done easily. This is because object-oriented analysis is resilient to changes as objects are more stable than functions that are used in structured analysis. Note that object-oriented analysis comprises a number of steps, which includes *identifying objects*, *identifying structures*, *identifying attributes*, *identifying associations*, and *defining services*.

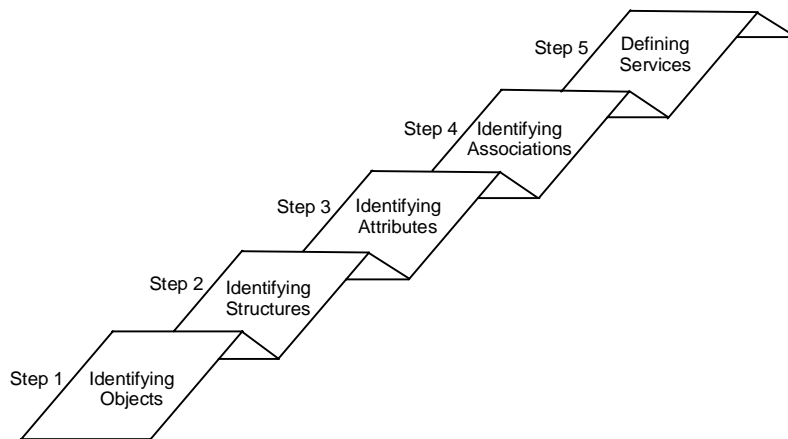


Figure 3.12 Steps in Object-oriented Analysis

NOTES

(a) Identifying Objects: While performing analysis, an object encapsulates the attributes on which it provides the services. Note that an object represents entities in a problem domain. The identification of the objects starts by viewing the problem space and its description. Then, a summary of the problem space is gathered to consider the 'nouns'. Nouns indicate the entities used in problem space and which will further be modelled as objects. Some examples of nouns that can be modelled as objects are structures, events, roles, and locations.

(b) Identifying Structures: Structures depict the hierarchies that exist between the objects. Object modelling applies the concept of generalization and specialization to define hierarchies and to represent the relationships between the objects. As mentioned earlier, superclass is a collection of classes, which can further be refined into one or more subclasses. Note that a subclass can have its own attributes and services apart from the attributes and services inherited from its superclass. To understand generalization and specialization, consider an example of class 'car'. Here, 'car' is a superclass, which has attributes, such as wheels, doors, and windows. There may be one or more subclasses of a superclass. For instance, superclass 'car' has subclasses 'Mercedes' and 'Toyota', which have the inherited attributes along with their own attributes, such as comfort, locking system, and so on.

It is essential to consider the objects that can be identified as generalization so that the classification of structure can be identified. In addition, the objects in the problem domain should be determined to check whether they can be classified into specialization or not. Note that the specialization should be meaningful for the problem domain.

(c) Identifying Attributes: Attributes add details about an object and store the data for the object. For example, the class 'book' has attributes, such as author name, ISBN, and publication house. The data about these attributes is stored in the form of values and are hidden from outside the objects. However, these attributes are accessed and manipulated by the service functions used for that object. The attributes to be considered about an object depend on the problem and the requirement for that attribute. For example, while modelling the student admission system, attributes, such as age and qualification are required for the object 'student'. On the other hand, while modelling for hospital management system, the attribute 'qualification' is unnecessary and requires other attributes of class 'student', such as gender, height, and weight. In short, it can be said that while using an object, only the attributes that are relevant and required by the problem domain should be considered.

(d) Identifying Associations: Associations describe the relationship between the instances of several classes. For example, an instance of class 'University' is related to an instance of

NOTES

class 'person' by 'educates' relationship. Note that there is no relationship between the class 'University' and class 'person', however only the instance(s) of class 'person' (that is, student) is related to class 'University'. This is similar to entity relationship modelling, where one instance can be related by 1:1, 1:M, and M:M relationships.

An association may have its own attributes, which may or may not be present in other objects. Depending on the requirement, the attributes of the association can be 'forced' to belong to one or more objects without losing the information. However, this should not be done unless the attribute itself belongs to that object.

(e) Defining Services: As mentioned earlier, an object performs some services. These services are carried out when an object receives a message for it. Services are a medium to change the state of an object or carry out a process. These services describe the tasks and processes provided by a system. It is important to consider the 'occur' services in order to create, destroy, and maintain the instances of an object. To identify the services, the system states are defined and then the external events and the required responses are described. For this, the services provided by objects should be considered.

3.5.3 Other Approaches

Many other approaches have been proposed for requirements analysis and specification. These approaches help to arrange information and provide an automated analysis of requirements specification of the software. In addition, these approaches are used for organizing and specifying a requirement. The specification language used for modelling can be either graphical (depicting requirements using diagrams) or textual (depicting requirements in text form). Generally, approaches used for analysis and specification include *structured analysis and design technique* and *entity relationship modelling*.

(a) Structured Analysis and Design Technique: Structured analysis and design technique (SADT) uses a graphical notation and is generally applied in information processing systems. The SADT language is known as **language of structured analysis (SA)**. SADT comprises of two parts, namely, structured analysis and design technique (DT). SA describes the requirements with the help of diagrams, whereas DT specifies how to interpret the results.

The model of SADT consists of an organized collection of SA diagrams. These diagrams facilitate a software engineer to identify the requirements in a structured manner by following top-down approach and decomposing system activities, data, and their relationships. The text embedded in these diagrams is written in natural language, thus specification language is a combination of both graphical language and natural language. The commonly used SA diagrams include activity diagram (actigram) and data diagram (datagram). These diagrams use input, output, control, and mechanism for providing a reference in an SA diagram. For this, both activity diagram and data diagram comprise of nodes and arcs. Note that each diagram must consist of 3 to 6 nodes including the interconnecting arcs. These diagrams are similar to data flow diagram as they follow top-down approach but differ from DFD as they may use loops, which are not used in it.

In Figure 3.13, an activity diagram is shown with nodes and arcs. The nodes represent the activities and arcs describe the data flow between the activities. Four different types of arcs can be connected to each node, namely, *input data*, *control data*, *processor*, and *output data*. **Input data** is the data that are transformed to output(s).

Control data is the data that constrain the kind or extent of process being described. **Processor** describes the mechanism, which is in the form of tools and techniques to perform the transformation.

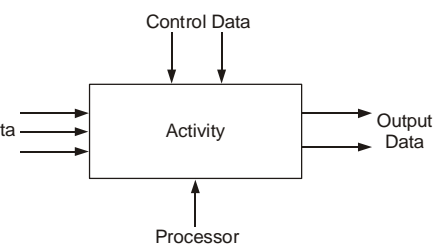


Figure 3.13 Activity Diagram

Output data is the result produced after sending input, performing control activity, and mechanism in a system. The arcs on the left side of a node indicate inputs and the arcs on the right-side indicate outputs. The arcs entering from the top of a node describe the control, whereas the arcs entering from the bottom describe the mechanism. The data flows are represented with the help of inputs and outputs while the processors represent the mechanism.

In Figure 3.14, a data diagram is shown with nodes and arcs, which are similar to that of an activity diagram. The nodes describe the data objects and the arcs describe the activities. A data diagram also uses four different types of arcs. The arcs on the left side indicate inputs and the arcs on the right side indicate the output. Here, input is the activity

that creates a data object, whereas output is the activity that uses the data object. The 'control activity' (arcs entering from top) controls the conditions in which the node is activated and the 'storage device' (arcs entering from bottom) indicates the mechanism for storing several representations of a data object. Note that in both the diagrams, controls are provided by the external environment and by the outputs from other nodes.

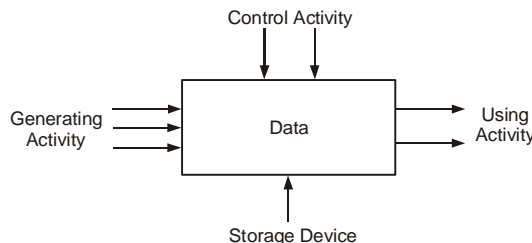


Figure 3.14 Data Diagram

Structured analysis and design technique provides a notation and a set of techniques, which facilitate to understand and record the complex requirements clearly and concisely. The top-down approach used in SADT helps to decompose high-level nodes into subordinate diagrams and to differentiate between the input, output, control, and mechanism for each node. In addition, this technique provides actigrams and datagrams and the management techniques to develop and review an SADT model. Note that SADT can be applied to all types of systems and is not confined only to software applications.

(b) Entity Relationship Modelling: IEEE defines entity relationship (ER) diagram as “a diagram that depicts a set of real-world entities and the logical relationships among them”. This diagram depicts entities, the relationships between them, and the attributes pictorially in order to provide a high-level description of conceptual data models. ER diagram is used in different phases of software development.

Once an ER diagram is created, the information represented by it is stored in the database. Note that the information depicted in an ER diagram is independent of the type of database and can later be used to create database of any kind, such as relational database, network database, or hierarchical database. ER diagram comprises of *data objects and entities, data attributes, relationships, and cardinality and modality*.

Data Objects and Entities: Data object is a representation of composite information used by software. Composite information refers to different features or attributes of a data object and this object can be in any of the form listed below:

- **External entity:** Describes the data that produces or accepts information. For example, a report.
- **Occurrence:** Describes an action of a process. For example, a telephone call.
- **Event:** Describes a happening that occurs at a specific place or time. For example, an alarm.
- **Role:** Describes the actions or activities assigned to an individual or object. For example, a systems analyst.
- **Place:** Describes location of objects or storage area. For example, a wardrobe.
- **Structure:** Describes the arrangement and composition of objects. For example, a file.

NOTES

NOTES

An entity is the data that stores information about the system in a database. Examples of an entity include real world objects, transactions, and persons.

Data Attributes Data attributes describe the properties of a data object. Attributes that identify entities are known as **key attributes**. On the other hand, attributes that describe an entity are known as **non-key attributes**. Generally, a data attribute is used to perform the functions listed below:

- Naming an instance (occurrence) of data object.
- Description of the instance.
- Making reference to another instance in another table.

Data attributes help to identify and classify an occurrence of entity or a relationship. These attributes represent the information required to develop software and there can be several attributes for a single entity. For example, attributes of 'account' entity are 'number', 'balance', and so on. Similarly, attributes of 'user' entity are 'name', 'address', and 'age'. However, it is important to consider the maximum attributes during requirements elicitation because with more attributes, it is easier for software development team to develop software. In case, some of the data attributes are not applicable, they can be discarded at later stage.

Relationships: Entities are linked to each other in different ways. This link or connection of data objects or entities with each other is known as **relationship**. Note that there should be at least two entities to establish relationship between them. Once the entities are identified, software development team checks whether relationship exists between them or not. Each relationship has a name, optionality (the state when relationship can be possible but not necessary), and degree (how many). These attributes confirm the validity of a given relationship. Based on this, three types of relationships exist among entities. These relationships are listed below:

- **One-to-one relationship (1:1):** Indicates that one instance of an entity is related only to another instance of another entity. For example, in a database of users in a bank, each user is related to only one account number.
- **One-to-many relationship (1:M):** Indicates that one instance of an entity is related to several instances of an entity. For example, one user can have many accounts in different banks.
- **Many-to-many relationship (M:M):** Indicates that many instances of entities are related to several instances of another entity. For example, many users can have their accounts in many banks.

To understand *entities*, *data attributes*, and *relationship*, let us consider an example. Suppose in a computerised banking system, one of the processes is to use saving account, which includes two entities, namely, 'user' and 'account'. Each 'user' has a unique 'account number', which makes it easy for the bank to refer to a particular registered user. On the other hand, account entity is used to deposit cash and cheque and to withdraw cash from the saving account. Depending upon the type and nature of transactions, it can be of various types, such as current account, saving account, or over draft account. The relationship between user and account can be described as 'user has account in a bank'.

In Figure 3.15, entities are represented by rectangles, attributes are represented by ellipses, and relationships are represented by diamond symbols. A key attribute is also depicted by an ellipse but with a line below it. This line below the text in the ellipse indicates the uniqueness of each entity.

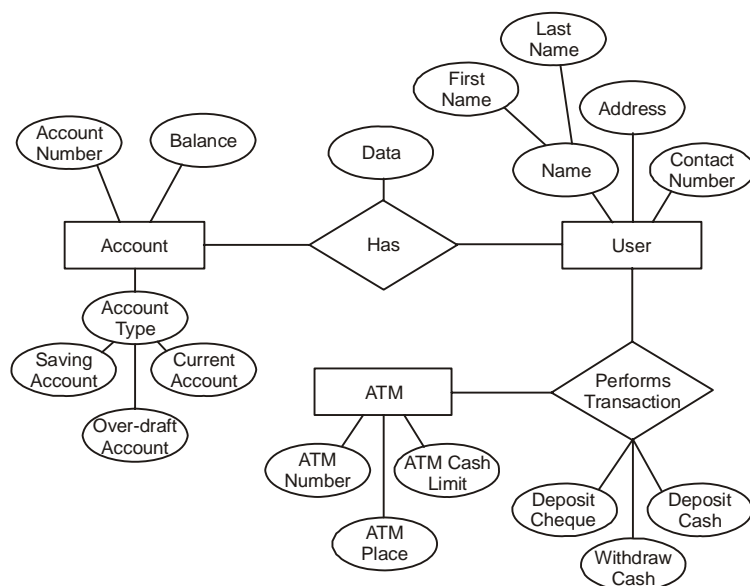


Figure 3.15 ER Diagram of Banking System

Cardinality and Modality: Although data objects, data attributes, and relationships are essential for structured analysis, additional information about them is required to understand the information domain of the problem. This information includes cardinality and modality. **Cardinality** specifies the number of occurrences (instances) of one data object or entity that relates to the number of occurrence of another data object or entity. It also specifies the number of entities that are included in a relationship. **Modality** describes the possibility whether a relationship between two or more entities and data objects is required or not. The modality of a relationship is 0 if the relationship is optional. However, the modality is 1 if an occurrence of the relationship is essential.

To understand the concept of cardinality and modality properly, let us consider an example. In Figure 3.16, user entity is related to order entity. Here, cardinality for 'user' entity indicates that user places an order, whereas modality for 'user' entity indicates that it is necessary for a user to place an order. Cardinality for 'order' indicates that a single user can place many orders, whereas modality for 'order' entity indicates that a user can arrive without any 'order'.

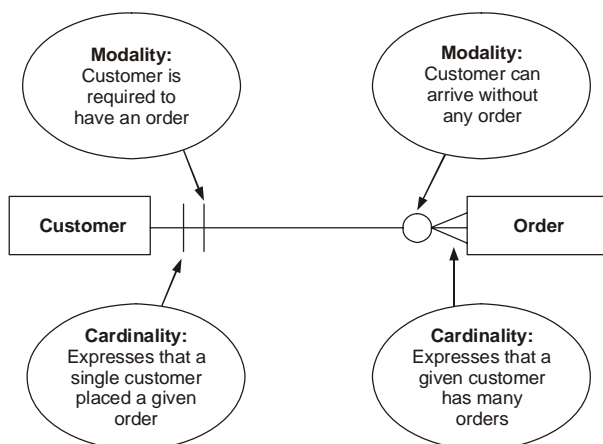


Figure 3.16 Cardinality and Modality

NOTES

Check Your Progress

9. How does IEEE define requirement analysis?
10. What are the basic principles of top down approach of requirement analysis?
11. Explain briefly two approaches used for requirement analysis and specification.

NOTES

3.6 REQUIREMENTS SPECIFICATION

The output of requirements phase of software development process is the **software requirement specification** document (also known as **requirements document**). This document lays a foundation for software engineering activities and is created when entire requirements are elicited and analyzed. 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.

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.

Essentially, what passes from requirement analysis activity to the specification activity is the knowledge acquired about the system. The need for maintaining 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 recovery are clearly specified in the requirements document. This information helps in properly developed design of a system. Various other purposes served by SRS are listed below:

- **Feedback:** Provides a feedback, which 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:** Organises the information and divides the problem into its component parts in an orderly manner.
- **Validation:** Uses validation strategies, applied to the requirements to acknowledge that requirements are stated properly.
- **Input to design:** Contains sufficient detail in the functional system requirements to devise a design solution.
- **Basis for agreement between user and organization:** 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:** 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:** Determines the requirements of the system and thus enable the developer to have a ‘rough’ estimate of the total cost and the schedule of the project.

Requirements document is used by various individuals in the organization As shown in Figure 3.17, system customers needs SRS to specify and verify whether requirements meet the desired needs or not. In addition, SRS enables the managers to plan for the system development processes. System engineers need requirements document to understand what system is to be developed. These engineers also require this document to develop validation test for the required system. Lastly, requirements document is required by system maintenance engineers to use the requirement and the relationship between its parts.

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.

NOTES

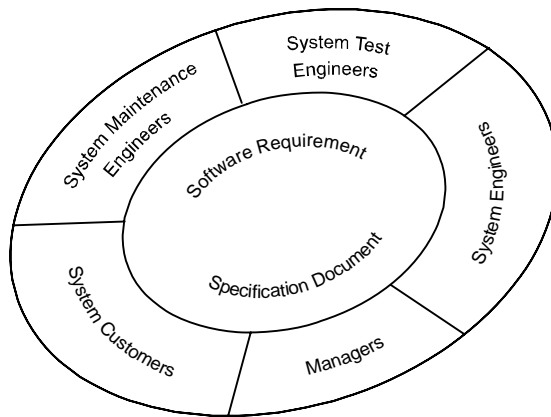


Figure 3.17 SRS Users

Characteristics of Software Requirements Specification: Software requirement specification should be accurate, complete, efficient, and of high-quality, so that it does not affect the entire project plan. A SRS is said to be of high quality when the developer and user easily understand the prepared document. Other characteristics of SRS are listed below:

- **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 where 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 it 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.

NOTES

- **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 subset of individual requirements defined does 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'.

3.6.1 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 very detail. Similarly, when flexibility is required in the requirements and where an in-house development takes place, requirements documents can be much 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 SRS. These guidelines are listed below:

- Functionality should be separate from implementation.
- Analysis model 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.
- Cognitive model (express a system as perceived by the users) should be developed instead of developing a design or implementation model.
- The content and structure of the specification should be flexible enough to accommodate changes.
- Specification 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. Suppose, 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 requirements document. However, the most widely used standard is by IEEE, which acts as a general framework. This general framework can be customised and adapted to meet the needs of a particular organization.

Each SRS fits 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 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 same.

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

NOTES

Figure 3.18 Software Requirements Specification Document

A software requirement specification document is shown in Figure 3.18. This document has many sections, which are listed below:

- **Introduction:** 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, this section describes definitions, abbreviations, and the acronyms used. The references used in SRS provide a list of documents that are referenced in the document.
- **Overall description:** Determines factors, which affects 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 of various sub-sections listed below:
 - **Product perspective:** 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. In addition, it also defines memory constraints and operations utilised by the user.
 - **Product functions:** Provide a summary of the functions to be performed by the software. The functions are organised in a list so that it is easily understandable to the user.
 - **User characteristics:** Determine general characteristics of the users.

NOTES

- **Constraints:** Provide the general description of the constraints, such as regulatory policies, audit functions, reliability requirements, and so on.
- **Assumption and dependency:** Provides a list of assumptions and factors that affect the requirements as stated in this document.
- **Apportioning of requirements:** Determine requirements that can be delayed until release of future versions of the system.
- **Specific requirements:** Determine all requirements in detail so that the designers can design the system according to the requirements. The requirements include description of every input and output of the system and functions performed in response to the input provided. It comprises of various sub-sections listed below:
 - **External interface:** Determines the interface of software with other system, which 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 specify 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:** 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:** 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 user 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:** 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:** 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 software can operate on existing hardware or some pre-determined hardware. This can impose restrictions while developing software design. Hardware limitations include hardware configuration of the machine and operating system to be used.
 - **Software system attributes:** Provide attributes, such as, reliability, availability, maintainability, and portability. It is essential to describe all these attributes to verify that these attributes are achieved in the final system.
 - **Organizing specific requirements:** Determine the requirements so that they can be properly organised for optimal understanding. The requirements can be organised on the basis of mode of operation, user classes, objects, feature, response, and functional hierarchy.

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

NOTES

3.7 REQUIREMENTS VALIDATION

The development of software starts, 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 begin. To check all the issues related to requirements, requirements validation is performed.

In validation phase, the work products produced 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 listed below:

- Certify that the SRS contains an acceptable description of the system to be implemented.
- Ensure that the actual requirements of the system are reflected in SRS.
- Check requirements document for completeness, accuracy, consistency, requirement conflict, conformance to standards, and technical errors.

Requirements validation is similar to requirements analysis as both these 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?



Figure 3.19 Requirements Validation

In Figure 3.19, 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.

Check Your Progress

12. Define Software requirement specification (SRS).
13. List the guidelines for preparing SRS.

NOTES

The output of requirement validation is a list of problems and agreed actions of the problems. 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.

3.7.1 Requirement Review

Requirements validation determines whether the requirements are substantial to design the system or not. Various problems are encountered during requirements validation. These problems are listed below:

- 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 above, 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 detected and corrected. In addition, the review team checks whether the work products produced during requirements phase conform to standards specified for the process, project and the product or not.

In review meeting, each participant goes over the requirements before the meeting starts and marks the items, which are dubious or they feel need for further clarification. Checklists are often used for identifying such items. Checklists ensure that no source of errors whether major or minor are overlooked by the reviewers. A 'good' checklist consists of the following:

- Is the initial state of the system defined?
- Does a conflict between one requirement and the other exist?
- 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 have 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 pattern used to simplify the requirements model?
- Are the requirements consistent with overall objective specified for the system/product?
- Have all hardware resources been defined?
- Is 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 'groundwork' for design. Using checklist, the participants specify the list of potential errors they have uncovered. Lastly, the requirement analyst either agrees to the presence of errors or clarifies that no errors exist.

3.7.2 Other Requirement 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 listed below:

- **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 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 computer aided software engineering (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.

3.8 REQUIREMENTS MANAGEMENT

Once a system has been deployed, new requirements inevitably emerge. It is difficult for the users to anticipate the effect of these new requirements (if a new system is developed for these requirements) on the organization. Thus, to understand and control changes to system requirements, requirements management is performed.

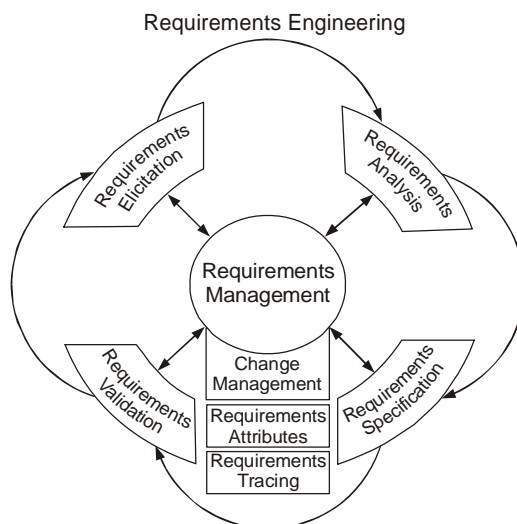


Figure 3.20 Requirements Management

Requirements management can be defined as a process of eliciting, documenting, organizing and controlling changes to the requirements. Generally, the process of requirements management begins as soon as requirements document is available, but 'planning' for

NOTES

Check Your Progress

14. What are the objectives to prepare software validation?
15. What is the output of requirement validation phase?

NOTES

managing the changing requirements should start during requirement elicitation process. The essential activities performed in requirements management are listed below:

- Recognises the need of the change to the requirements.
- Establishes a relationship amongst stakeholders and involve them in the requirements engineering process.
- Identifies and tracks requirements attributes.

Requirements management enables the development team to identify, control, and track requirements and changes that occur as the software development process progresses. Other advantages associated with the requirements management are listed below:

- **Better control of complex projects:** Provides the development team with a clear understanding of *what*, *when* and *why* software is to be delivered. The resources are allocated according to user-driven priorities and relative implementation effort.
- **Improves software quality:** Ensures that the software performs according to requirements to enhance software quality. This can be achieved when the developers and testers have a concise understanding of what to develop and test.
- **Reduced project costs and delays:** Minimizes errors early in the development cycle, as it is expensive to 'fix' errors at the later stages of the development cycle. As a result, the project costs also reduce.
- **Improved team communications:** Facilitates early involvement of users to ensure that their needs are achieved.
- **Easing compliance with standards and regulations:** Ensures that standards involved with software compliance and process improvement have thorough understanding of requirement management. For example, capability maturity model (CMM) addresses requirements management as one of the first steps to improve software quality.

All the user requirements are specified in the software requirement specification. The project manager as part of requirements management tracks the requirement for the current project and those requirements, which are planned for the next release.

3.8.1 Requirements Management Process

Requirements management starts with planning, which establishes the level of requirements management needed. After planning, each requirement is assigned a unique 'identifier' so that it can be crosschecked by other requirements. Once requirements are identified, requirements tracing is performed.

Requirement tracing is a medium to trace requirements from the start of development process till the software is delivered to the user. The objective of requirement tracing is to ensure that all the requirements are well understood and are included in test plans and test cases. Various advantages of requirement tracing are listed below:

- Verifies whether user requirements are implemented and adequately tested or not.
- Enables the user understanding of impact of changing requirements.

Traceability techniques facilitate the impact of analysis on changes of the project, which is under development. Traceability information is stored in a **traceability matrix**, which relates requirements to stakeholders or design module. Traceability matrix refers to a table that correlates high-level requirements with the detailed requirements of the product. Mainly, five types of traceability tables are maintained. These are listed in Table 3.4.

In traceability matrix each requirement is entered in a row and column of the matrix. The dependencies between different requirements are represented in the cell at a row and column intersection. In Figure 3.21, 'U' in the row and column intersection indicates the dependencies of the requirement in the row on the column and 'R' in the row and column intersection indicates the existence of some other weaker relationship between the requirements.

Table 3.4 Types of Traceability Tables

Traceability Table	Description
Features traceability	Indicates how requirements relate to important features specified by the user.
Source traceability	Identifies the source of each requirement by linking the requirements to the stakeholders who proposed them. When a change is proposed, information from this table can be used to find and consult the stakeholders.
Requirement traceability	Indicates how dependent requirements in the SRS are related to one another. Information from this table can be used to evaluate the number of requirements that will be affected due to the proposed change(s).
Design traceability	Links the requirements to the design modules where these requirements are implemented. Information from this table can be used to evaluate the impact of proposed requirements changes on the software design and implementation.
Interface traceability	Indicates how requirements are related to internal interface and external interface and external interface of a system.

NOTES

Note that tracing matrix is useful when less number of requirements are to be managed. However, traceability matrices are expensive to maintain when a large system with large requirement is to be developed. This is because large requirements are not easy to manage. Due to this, the traceability information of large system is stored in the ‘requirement database’ where each requirement is explicitly linked to related requirements. This helps to assess, how a change in one requirement affects the different aspects of the system to be developed.

Req. ID	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		U	R					
1.2			U			R		U
1.3	R			R				
2.1			R		U			U
2.2								U
2.3		R		U				
3.1								R
3.2							R	

Figure 3.21 Traceability Matrix**3.8.2 Requirements Change Management**

Requirements change management is used when there is a request or proposal for a change to the requirements. The advantage of this process is that the changes to the proposals are managed consistently and in a controlled manner.

**Figure 3.22** Requirement Change Management

NOTES

An efficient requirements change management process undergoes a number of stages for changes to the requirement, which are shown in Figure 3.22. These stages are listed below:

- **Problem analysis and change specification:** The entire process begins with identification of problems to the requirements. The problem or proposal is analyzed to verify whether the change is valid or not. The outcome of the analysis is provided to the 'change requester' and a more specific requirements change proposals is then made.
- **Change analysis and costing:** The effect of a change requested on the requirement is assessed according to traceability information. The cost for this can be estimated on the basis of modification made to the design and implementation. After analysis is over, a decision is made as to whether changes are to be made or not.
- **Change implementation:** Finally, the changes are made to the requirements document, system design and implementation. The requirements document is organised in such a manner so that changes to it can be made without extensive rewriting. Minimising the external references and making document sections modular achieve changeability in the document. By doing this, individual sections can be changed and replaced without affecting other parts of the document.

3.9 CASE STUDY: STUDENT ADMISSION AND EXAMINATION SYSTEM

ABC University wants to automate its admission and examination system for the two years course of masters in business administration (MBA). The main objective of developing this software is to help the university to manage the database of students efficiently. This software will maintain the electronic record related to personal and academic data of each student.

3.9.1 Problem Statement

The problem statement provides an outline of the system from user's perspective. ABC University offers IV-semester MBA programme. This statement has three modules, namely, *registration module*, *examination module*, and *result generation module*.

- **Registration module:** To be a part of the university, an applicant must be registered, for which the applicant should pay the required registration fee. This fee can be paid through demand draft or cheque drawn from a nationalized bank. After successful registration an enrolment number is allotted to each student, which makes the student eligible to appear in the examination.
- **Examination module:** The examination of the MBA programme comprises of assignments, theory papers, practical papers, and a project.
 - **Assignments:** Each subject has an associated assignment, which is compulsory and should be submitted by the student before a specified date. Each assignment carries 20 marks where student obtaining 40% or more (≥ 8 marks) is said to have passed.
 - **Theory papers:** The theory papers can be core or elective. **Core** papers are mandatory papers, while in **elective** papers, students have a choice to select two out of three papers. Note that in first three semesters there are four core papers and three elective papers out of which two papers are to be chosen. Also, the student is required to prepare a project in the IVth semester. Each theory paper carries 50 marks where student obtaining 40% or more (≥ 20 marks) is said to have passed.
 - **Practical papers:** The practical papers are mandatory and every semester has three of them. Each practical paper carries 30 marks where student obtaining 40% or more (≥ 12 marks) is said to have passed.

Check Your Progress

- Describe the role of requirement tracing in the process of requirements management.
- Explain three stages of requirement change management.

- **Project:** Students need to submit a project in the IVth semester. This project carries 100 marks where student obtaining 50% or more (≥ 50 marks) is said to have passed. Also, students are required to appear for a viva-voce session, which will be related to the project.
- **Result generation module:** The result is declared on the university's website. This website contains mark sheets of the students who have appeared in the examination of the said semester (for which registration fee has been paid). Note that to view the result student can use enrolment number as password.

NOTES

3.9.2 Data Flow Diagrams

The data flow diagrams of various levels are shown as follows:

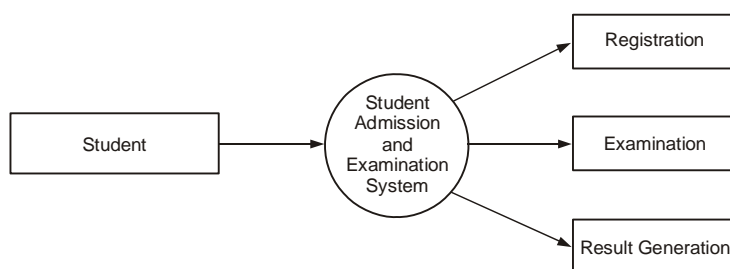


Figure 3.23 Level 0 DFD

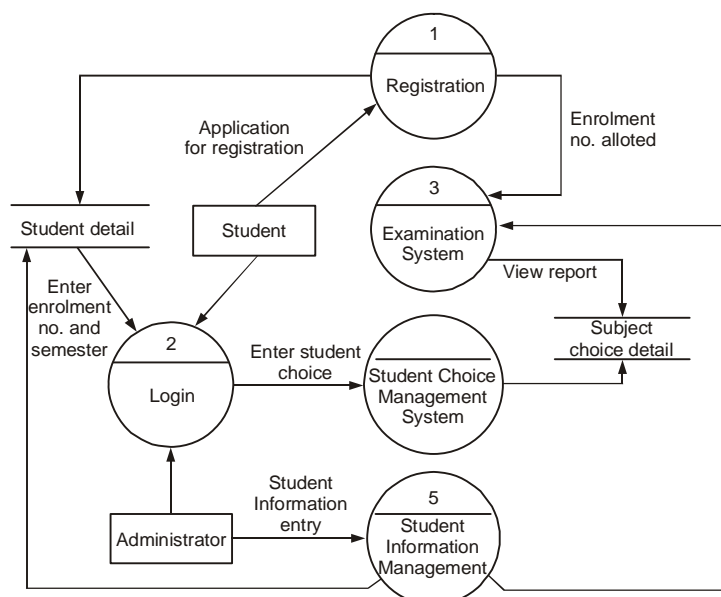


Figure 3.24 Level 1 DFD of Student Admission and Examination System

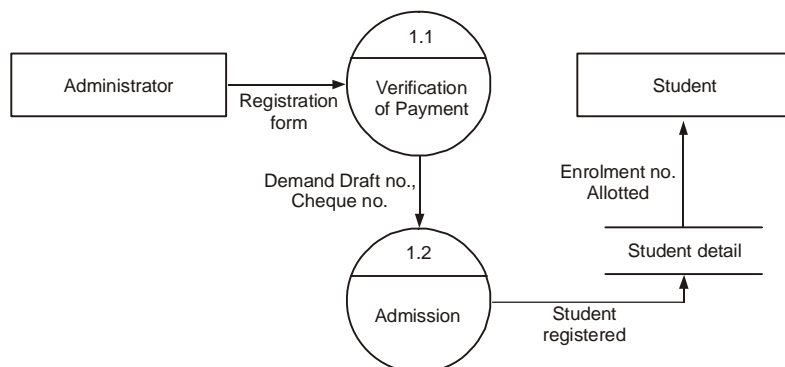


Figure 3.25 Level 2 DFD of Registration

NOTES

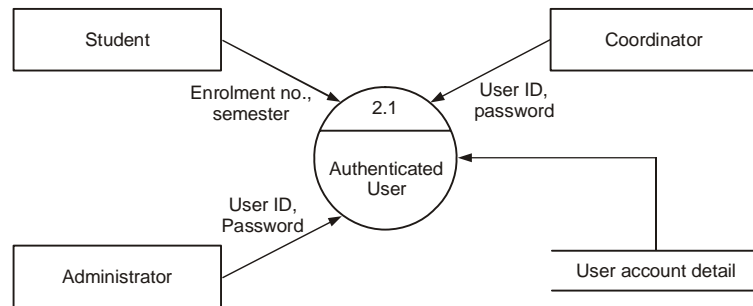


Figure 3.26 Level 2 DFD of Marks Information System

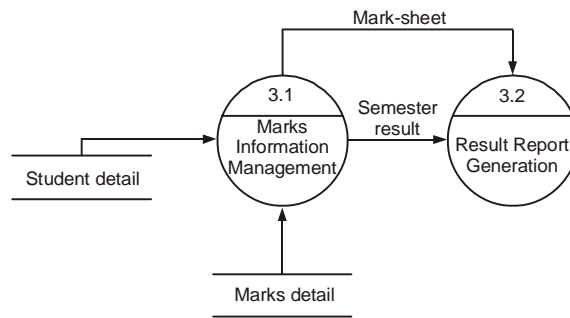


Figure 3.27 Level 2 DFD of Examination

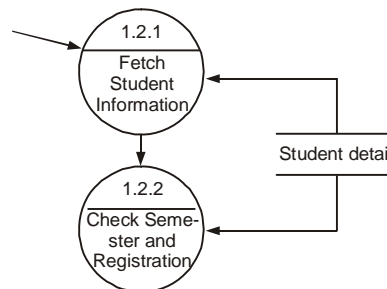


Figure 3.28 Level 3 DFD Registration

3.9.3 Entity Relationship Diagram

The ER diagram of student admission and examination system is shown in Figure 3.29.

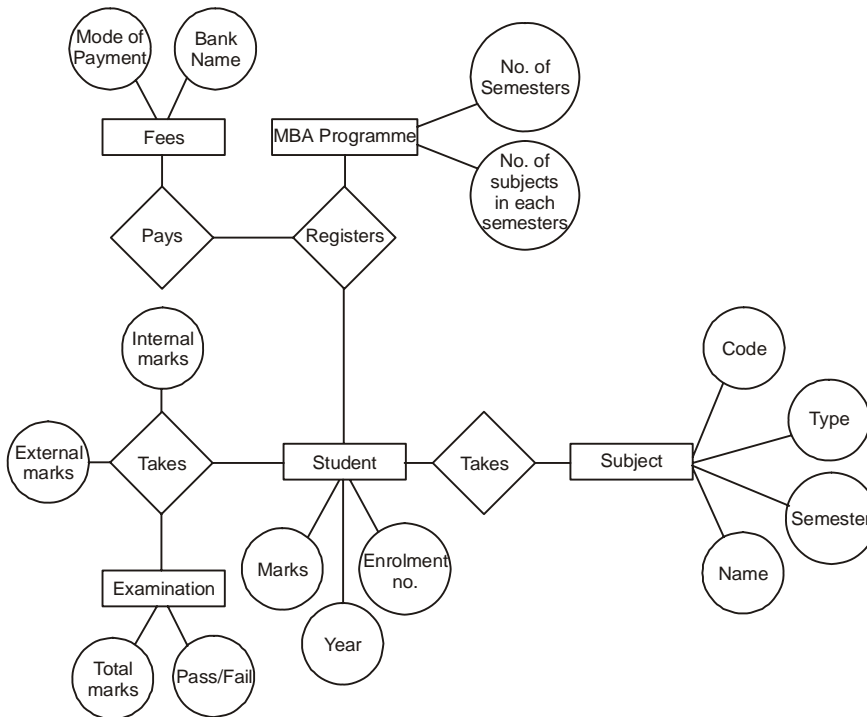


Figure 3.29 ER Diagram of Student Management System

3.9.4 Software Requirements Specification Document

The SRS document describes the overall requirements of ABC University to automate the proposed system. This document follows IEEE guidelines for requirements specification document with some variations.

1. Introduction

This section specifies the overall requirements of the software. The final software will have features according to this document and assumptions for any additional features should be made by individuals involved in developing/ testing/ implementing/ using this product.

(a) Purpose

The requirements specification document determines the capability of the software to be developed. In addition, it specifies constraints required by the system.

(b) Scope

The final software when developed will help the university in registering students and conducting examination. In addition, this will manage the record of the subjects offered in different semesters, the students' choice of elective papers and the marks obtained by them in different subjects in various semesters.

(c) Definitions, acronyms, and abbreviations

Following abbreviations are used in the entire specification document.

MBA: Master in business administration

DB: Database

DBMS: Database management system

NOTES

NOTES

RAM: Random access memory

MB: Megabyte

(d) References

University website: Provides information about the course, result, and other information.

(e) Overview

The SRS document provides description about the system requirements, interfaces, features, and functionalities.

2. Overall description

The proposed system will maintain information about the students who are enrolled in the MBA programme. In addition, it will manage the record of the subjects taken by the students in different semesters, choice of elective paper and marks obtained by the students in different semesters. In the Ist, IInd, and IIIrd semesters, students have to appear in six theory subjects and three practical papers. It is mandatory to submit a project report in IVth semester, which is followed by viva-voce for the same.

(a) Product perspective

The application will be Windows-based and an independent software application.

(i) System interface

None

(ii) Interface

The application will have a menu screen, which will have the following options:

- **Login screen:** Enter the user name, password, and role (student, administrator, and coordinator). Note that role is defined to know the information about the individual(s) accessing the software. This is essential to prevent the students from modifying the result in the database. Hence, the students will have access to the information about whether they have been successfully registered or not and can view the subject-wise result of each semester or year.
- **Subject screen:** Enter information regarding the subjects offered in different semesters. In addition, the subject screen displays the information about the assignments, subjects (that is, core or elective), and the project.
- **Examination screen:** Enter information about the registered students who seek to take examination.
- **Student screen:** Enter information about the student enrolled for MBA in different semesters.
- **Marks screen:** Enter information about the marks of assignments, theory papers, and practical papers. In addition, marks screen displays the information of the subjects successfully completed. Marks of the student will be displayed in the form of printable mark-sheet, which includes total marks and percentage of the student.

(iii) Hardware interface

Screen resolutions with minimum of 800 × 600 pixels should be used. It should also support output devices like printer.

(iv) Software interface

The software interfaces that will be used for the proposed system are listed below:

- Windows-based operating system (such as, Windows 95/98/XP/NT).
- Oracle 8i as the database management system (DBMS) to store files and other related information.
- Crystal reports 8 to generate and view reports.
- Visual Basic 6.0 as a front-end tool for coding and designing the software.
- Internet Explorer 5.5 or higher to view results of the examination on the Internet.

(v) Communication interface

None

(vi) Memory constraints

Intel Pentium III processor or higher with a minimum of 128 MB RAM and 600 MB of hard disk space will be required so that software performs its functions in an optimum manner.

(vii) Operations

The software release will not include automated and maintenance of database. The university is responsible for manually deleting old/outdated data and managing backup and recovery of data.

(viii) Site adaptations requirements

The terminals at the user's end will have to support the interfaces (both hardware and software) as mentioned above.

(b) Product functions

The system will allow access only to authorized users like student, administrator and coordinator depending upon the role. Some of the functions that will be performed by the software are listed below:

- Login facility for authorized users.
- Perform modification (by administrator only), such as adding or deleting the marks obtained by the students.
- Provide a printable version of the mark-sheet (result) of the students.
- Use of 'clear' function to delete the existing information in the database.

(c) User characteristics

None.

(d) Constraints

- As Oracle 8i is a powerful database, it can store a large number of records.
- The university should have a security policy to maintain information related to marks, which are to be modified by administrator.

(e) Assumptions and dependencies

- The subjects taken by the students in the semester will not change.

NOTES

NOTES

- The number of semesters and elective subjects offered by the university will not change.

(f) Apportioning of requirements

Not required.

3. Specific requirements

This section provides the information required by the developers to develop the system.

(a) External interface

This contains complete description of inputs and outputs from the software system.

(b) Functions

None.

(c) Performance requirements

None.

(d) Logical database requirements

The information that will be stored in the database is listed below:

- **Student detail:** Stores information about student's enrolment number, student name, the year of enrolment, and fees details according to the semester.
- **Subject choice detail:** Stores information about subject name, code, and semester. In addition, it stores information about enrolment number, semester, and the subject chosen by the student.
- **Marks detail:** Stores information about student's enrolment number and the subject-wise marks secured by the student.
- **User account detail:** Stores information about user name, password, and role.

(e) Design constraints

None.

(f) Software system attributes**(i) Security**

The application will be password protected and hence, will require users to enter their login ID (user name) and password.

(ii) Maintainability

The application will be designed in a manner that it is easy to modify the software system later, when required and to incorporate new requirements in the individual modules, such as subject information, marks information, and user accounts.

(iii) Portability

The application will be easily portable on any Windows-based system that has Oracle 8i installed on it.

4. Change management process

In case the university desires to modify the criteria to select the elective papers or change the number of practical papers in each semester, then the changes will be updated and reflected in SRS document accordingly.

5. Document approvals

When the requirements are gathered according to the user, SRS is then finally reviewed, approved, and signed by the developer and user (university). This SRS serves as a contract for software development activities.

6. Supporting information

None.

NOTES

3.10 DATA DICTIONARY

The data dictionary for the data stores used in the level 1 DFD can be shown as in Figure 3.30

User detail = *Entity. Stores the personal information of the user. * <u>Account_no.</u> + First_name + Middle_name + Last_name + Address + Phone
Cash detail = * Entity. Stores the details of the cash deposited or withdrawn * Amount + No_of_notes + Total_amount
DD detail: *Entity. Stores the details of demand drafts * <u>DD_number</u> + DD_date + DD_amount + In_favour_of + Payable_at

Figure 3.30 Data Dictionary

3.11 LET US SUMMARIZE

1. A requirement is defined 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).
2. Guidelines act as an efficient method of expressing requirements, which also provide a basis for software development, system testing, and user satisfaction.
3. Various requirements considered before starting software development are generally classified into three categories, namely, functional requirements, non-functional requirements, and domain requirements.
4. 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.
5. The non-functional requirements, also known as quality requirements, relate to system attributes, such as reliability and response time. Different types of non-functional

NOTES

requirements include product requirements, organizational requirements, and external requirements.

6. 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.
7. The requirements engineering process is a series of activities that are performed in requirements phase in order to express requirements in software requirements specification (SRS) document. This process focuses on understanding the requirement and its type so that an appropriate technique is determined to carry out the requirements engineering process.
8. Various steps of requirements engineering process include feasibility study, requirements elicitation, requirements analysis, requirements specification, requirements validation, and requirements management.
9. Feasibility refers to the evaluation of the software process, design, procedure, or plan in order to determine whether they can be successfully accomplished in the software in the allotted time or not. To evaluate feasibility, a feasibility study is performed, which determines whether the solution considered to accomplish the requirements is practically workable in the software or not.
10. The commonly considered types of feasibility include technical feasibility, operational feasibility, and economic feasibility.
11. Technical feasibility assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget.
12. Operational feasibility assesses the extent to which the required software performs series of steps to solve business problems and user requirements.
13. Economic feasibility determines whether the required software is capable of generating economic benefits for an organization or not.
14. Requirements elicitation, also known as requirements capture and requirements acquisition, is a process of collecting information about software requirements from different individuals, such as users and other stakeholders.
15. The commonly followed elicitation techniques include interviews, scenarios, prototypes, and quality function deployment.
16. Requirements analysis is (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.
17. Analysis model comprises of structured analysis, object-oriented modelling, and by applying other approaches.
18. Structured analysis is a top-down approach, which focuses on refining the problem with the help of functions performed in the problem domain and data produced by these functions. Generally, the structured analysis is depicted by a data flow diagram, which uses several levels to provide detailed information about the system.
19. A data dictionary is an organized collection of information about data and their relationships, data flows, data types, data stores, processes, and so on. It also helps users to understand the data types and processes defined along with their uses.
20. The object-oriented modelling defines a system as a set of objects, which interact with each other by the services they provide. In addition, objects interact with users through their services so that they can avail the required services in the system. Various concepts used in object-oriented modelling are objects, classes, attributes, operations, superclass, subclass, and so on.

21. Structured analysis and design technique (SADT), also known as language of structured analysis, uses a graphical notation and is generally applied in information processing systems. SADT comprises of two parts, namely, structured analysis (SA) and design technique (DT). SA describes the requirements with the help of diagrams, whereas DT specifies how to interpret the results.
22. ER diagram is a diagram that depicts a set of real-world entities and the logical relationships among them. This diagram depicts entities, the relationships between them, and the attributes pictorially in order to provide a high-level description of conceptual data models. ER diagram comprises of data objects and entities, data attributes, relationships, and cardinality and modality.
23. 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.
24. Requirements validation determines whether the requirements are substantial to design the system or not. Requirements validation techniques include test case generation, automated consistency analysis, and prototyping.
25. Requirements management is a process of eliciting, documenting, organizing and controlling changes to the requirements. The process of requirements management begins as soon as requirements document is available, but 'planning' for managing the changing requirements should start during requirement elicitation process.
26. Requirements change management is used when there is a request or proposal for a change to the requirements. The advantage of this process is that the changes to the proposals are managed consistently and in a controlled manner.

NOTES

3.12 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The requirements, which are commonly considered, are classified into three categories, namely, functional requirements, non-functional requirements, and domain requirements.
2. The requirements engineering (RE) process is a series of activities that are performed in requirements phase in order to express requirements in software requirements specification (SRS) document. This process focuses on understanding the requirement and its type so that an appropriate technique is determined to carry out the requirements engineering process.
3. The objective of feasibility study is to establish the reasons for developing software that is acceptable to users, adaptable to change, and conformable to established standards. Various other objectives of feasibility study are listed below:
 - Analyze whether the software will meet organizational requirements or not.
 - Determine whether the software can be implemented using current technology and within the specified budget and schedule or not.
 - Determine whether the software can be integrated with other existing software or not.
4. Operational feasibility performs tasks listed below:
 - Determines whether the problems proposed in user requirements are of high priority or not.
 - Determines whether the solution suggested by software development team is acceptable or not.

NOTES

- Analyses whether users will adapt to new software or not.
 - Determines whether the organization is satisfied by the alternative solutions proposed by software development team or not.
5. Information assessment identifies information about whether the system helps in achieving the objectives of the organisation. In addition it verifies that the system can be implemented using new technology and within the budget. It also verifies whether the system can be integrated with the existing system.
 6. **Functional objective:** Provides information about functions of the system, such as new services, increased capacity, and so on.

Performance objective: Provides information about performance objectives, such as reduced staff and equipment cost, increased processing speed of software, and improved controls.
 7. The commonly followed elicitation techniques are listed below:
 - Interviews
 - Scenarios
 - Prototypes
 - Quality function deployment (QFD)
 8. Generally, a scenario comprises of the information listed below:
 - Description of what users expect when scenario starts.
 - Description of how to handle the situation when software is not operating correctly.
 - Description of the state of software when scenario ends.
 9. IEEE defines requirements 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”.
 10. Structured analysis is a top-down approach, which focuses on refining the problem with the help of functions performed in the problem domain and data produced by these functions. The basic principles of this approach are:
 - To facilitate software engineer in order to determine the information received during analysis and to organize the information to avoid complexity of the problem.
 - To provide a graphical representation to develop new software or enhance the existing software.
 11. Generally, approaches used for analysis and specification include structured analysis and design technique and entity relationship modelling.
 - Structured analysis and design technique (SADT) uses a graphical notation and is generally applied in information processing systems. The SADT language is known as language of structured analysis (SA). SADT comprises of two parts, namely, structured analysis and design technique (DT). SA describes the requirements with the help of diagrams, whereas DT specifies how to interpret the results.
 - IEEE defines entity relationship (ER) diagram as “a diagram that depicts a set of real-world entities and the logical relationships among them”. This diagram depicts entities, the relationships between them, and the attributes pictorially in order to provide a high-level description of conceptual data models. ER diagram is used in different phases of software development.
 12. 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.”

13. 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 SRS. These guidelines are listed below:

- Functionality should be separate from implementation.
- Analysis model 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.
- Cognitive model (express a system as perceived by the users) should be developed instead of developing a design or implementation model.
- The content and structure of the specification should be flexible enough to accommodate changes.
- Specification should be robust. That is, it should be tolerant towards incompleteness and complexity.

14. In validation phase, the work products produced 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 validation are listed below:

- Certify that the SRS contains an acceptable description of the system to be implemented.
- Ensure that the actual requirements of the system are reflected in SRS.
- Check requirements document for completeness, accuracy, consistency, requirement conflict, conformance to standards, and technical errors.

15. The output of requirement validation is a list of problems and agreed actions of the problems. 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.

16. Requirement tracing is a medium to trace requirements from the start of development process till the software is delivered to the user. The objective of requirement tracing is to ensure that all the requirements are well understood and are included in test plans and test cases.

17. An efficient requirements change management process undergoes a number of stages for changes to the requirement.

These stages are listed below:

- **Problem analysis and change specification:** The entire process begins with identification of problems to the requirements. The problem or proposal is analyzed to verify whether the change is valid or not. The outcome of the analysis is provided to the ‘change requester’ and a more specific requirements change proposals is then made.
- **Change analysis and costing:** The effect of a change requested on the requirement is assessed according to traceability information. The cost for this can be estimated on the basis of modification made to the design and implementation. After analysis is over, a decision is made as to whether changes are to be made or not.
- **Change implementation:** Finally, the changes are made to the requirements document, system design and implementation. The requirements document is organised in such a manner so that changes to it can be made without extensive

NOTES

rewriting. Minimising the external references and making document sections modular achieve changeability in the document. By doing this, individual sections can be changed and replaced without affecting other parts of the document.

NOTES

3.13 QUESTIONS AND EXERCISES

I. Fill in the Blanks

1. The different types of software system requirements are _____, non-functional requirements, and _____.
2. The notations used to depict information in a data flow diagram include _____, data flow, _____, and process.
3. _____ defines a system as a set of objects, which interact with each other by the services they provide.
4. For an SRS document to be accurate and efficient, it should be correct, _____, _____, and verifiable.

II. Multiple Choice Questions

1. Which of the following is not a step of requirements engineering process?
 - (a) Requirements specification
 - (b) Requirements analysis
 - (c) Feasibility study
 - (d) Requirements prioritization
2. Which of the following is the user requirement identified in quality function deployment?
 - (a) Expected requirements
 - (b) General requirements
 - (c) Both (a) and (b)
 - (d) None of the above
3. SADT stands for:
 - (a) Software analysis and development technique
 - (b) Structured analysis and design technique
 - (c) System analysis and design technique
 - (d) Structured analysis and development technique
4. Which one of the following is a requirements validation technique?
 - (a) Interviews
 - (b) Automated consistency analysis
 - (c) SADT
 - (d) Quality function deployment

III. State Whether True or False

1. Both DFD and flowchart depict the flow of data.
2. An entity diagram depicts a set of real-world entities and the logical relationships among them.
3. The structure of an SRS document changes depending upon the project and requirements.
4. Traceability matrix refers to a table that correlates user requirements with the organizational requirements.

IV. Descriptive Questions

1. “It is easy for software engineers to develop software according to user requirements even if they are incomplete as software engineers can consider the user requirements of earlier developed software”. Do you agree with this statement? Why or why not? Give reasons in support of your answer.
2. What is requirements management? Describe its process.
3. Consider a student admission system for XYZ University, which is to be automated. For this system, create the following:
 - (a) Make DFDs of 2–3 levels.
 - (b) Draw ER diagram
4. Prototyping is advantageous to understand the problem and user requirements. Do you think it is always advantageous to perform prototyping? Are there any disadvantages of this approach?
5. HEP university decides to design software for its library information system, which should allow only the authorised person to insert, delete, upgrade, and select records related to books in the system. Also, the system should maintain information related to the issue and return of books to members. For this system, create the following:
 - (a) Develop the software requirements specification.
 - (b) Design DFDs of 2–3 levels.
 - (c) Identify various modules and their operation.
 - (d) Design ER diagram depicting the library information system.

NOTES

3.14 FURTHER READING

1. Software Engineering: A Practitioner’s Approach – *Roger Pressman*
2. Software Engineering – *Ian Sommerville*
3. An Integrated Approach to Software Engineering – *Pankaj Jalote*

UNIT 4 SOFTWARE DESIGN

Structure

- 4.0 Introduction
- 4.1 Unit Objectives
- 4.2 Basics of Software Design
 - 4.2.1 Principles of Software Design; 4.2.2 Software Design Concepts
 - 4.2.3 Developing a Design Model
- 4.3 Data Design
- 4.4 Architectural Design
 - 4.4.1 Architectural Design Representation; 4.4.2 Architectural Styles
- 4.5 Procedural Design
 - 4.5.1 Functional Independence
- 4.6 User Interface Design
 - 4.6.1 User Interface Rules; 4.6.2 User Interface Design Process
 - 4.6.3 Evaluating User Interface Design
- 4.7 Software Design Notation
- 4.8 Software Design Reviews
 - 4.8.1 Types of Software Design Reviews; 4.8.2 Software Design Review Process
 - 4.8.3 Evaluating Software Design Reviews
- 4.9 Software Design Documentation (SDD)
- 4.10 Case Study: Higher Education Online Library System
 - 4.10.1 Data Design; 4.10.2 Architectural Design; 4.10.3 Procedural Design
 - 4.10.4 User Interface Design
- 4.11 Object-oriented Concepts
- 4.12 Let us Summarize
- 4.13 Answers to 'Check Your Progress'
- 4.14 Questions and Exercises
- 4.15 Further Reading

NOTES

4.0 INTRODUCTION

Once the requirements document for the software to be developed is available, the software design phase begins. While the requirement specification activity deals entirely with the problem domain, design is the first phase of transforming the problem into a solution. In design phase, customer and business requirements and technical considerations all come together to formulate a product or a system.

Design process comprises of a set of principles, concepts, and practices, which allows a software engineer to model the system or product that is to be built. This model known as design model is assessed for quality and reviewed before code is generated and tests are conducted. The design model provides detail about software data structures, architecture, interfaces, and components, which are required to implement the system. This chapter discusses the design elements required to develop a software design model. It also discusses the design patterns, design notations and design documentation used to represent software design.

4.1 UNIT OBJECTIVES

After reading this unit, the reader will understand:

- Why software design is considered to be an important software engineering activity?

NOTES

- Various software design principles, which act as a framework for the designers to follow a good design practice.
- Various software design concepts, which form the base for software design process.
- Elements of software design model, which include data design, architecture design, procedural design, and interface design.
- How data design leads to better program structure, effective modularity, and reduced complexity?
- Architectural design, which acts as a preliminary ‘blueprint’ from which software can be developed.
- Procedural design, which is created by transforming the structural elements defined by the software architecture into procedural descriptions of software components.
- How user interface design creates effective communication medium between a human and a computing machine?
- How design notations are used to represent software design.
- How software design reviews are used to evaluate the adequacy of the design requirements?
- The importance of software design documentation.

4.2 BASICS OF SOFTWARE DESIGN

Software design is a software engineering activity where software requirements are analyzed in order to produce a description of the internal structure and organization of the system that serves as a basis for its construction (coding). 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 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.

Objectives of Software Design: The main objective of software design phase is to develop a blue print which serves as a base while developing the software system. The other objectives of software design are listed below:

- To produce various models that can be analyzed and evaluated to determine if they will allow the various requirements to be fulfilled.
- To examine and evaluate various alternative solutions and trade-offs involved.
- To plan subsequent software development activities.

4.2.1 Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Since problems in the design phase can be very expensive to solve in later stages of the software development, a number of principles are considered while designing the software. These principles act as a framework for the designers to follow a good design practice. Some of the commonly followed design principles are listed below:

- **Software design should be traceable to the analysis model:** As a single design element often relates to multiple requirements, it becomes essential to have a means for tracking how requirements are satisfied by the design model.
- **Choose the right programming paradigm:** A programming paradigm is the framework used for designing and describing the structure of the software system. The two most

popular programming paradigms are the procedural paradigm and the object-oriented paradigm. The paradigm should be chosen keeping in mind constraints, such as time, availability of resources, and nature of user's requirements.

- **Software design should demonstrate uniformity and integration:** In most cases, rules, format, and styles are defined in advance to the design team before the design work begins. The design is said to be integrated and uniform 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 appraise to minimise conceptual (semantic) errors:** 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:** Software should be designed to handle unusual changes and circumstances, and if need arise for termination; it must do so in a proper manner so that functionality of the software is not affected.

NOTES

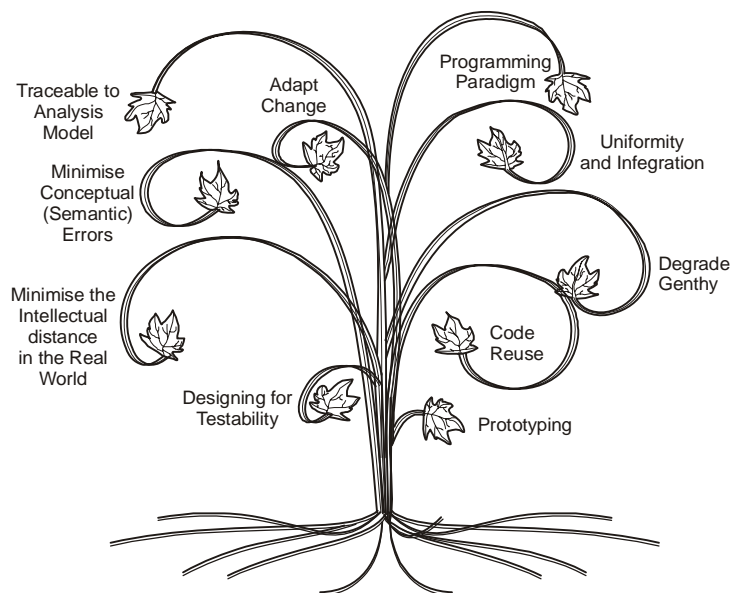


Figure 4.1 Design Principles of Software

- **Software design should ‘minimise the intellectual distance’ 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 separate testing from design and implementation. That is, the software is designed, 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 discovering these types of errors after implementation usually requires the entire or a substantial part of the software to be redone. Thus, the test engineers should be involved from the very beginning. For example, they should work with the requirements analysts to devise tests that will determine whether the software meets the requirements or not.

NOTES

- **Prototyping:** Prototyping should be used to explore those aspects of the requirements, user interface, or software's internal design, which are not easily understandable. Using prototyping a quick 'mock-up' of the system can be developed. This mock up can be used as a highly effective means to highlight misconceptions and reveal hidden assumptions about the user interface and how the software should perform. Prototyping also reduces the risk of designing software that does not fulfil customer's requirements.

Note that 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.

4.2.2 Software Design Concepts

Every software process is characterised 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 development of 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.

(a) **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*". Abstraction is used both as a process and as an entity. As a **process**, it denotes the extraction of the essential details about an item, or a group of items, while ignoring non-essential details. As an **entity**, it denotes a model, a view, or some other focused representation of an actual item.

Each step in the software process is accomplished through various levels of abstraction. At the highest level of abstraction, a solution is broadly stated using the language of the problem environment. At lower levels of abstraction, a detailed description about the solution is presented. For example, during system engineering, software is viewed as an element of computer based engineering, while in requirement analysis, 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:** Involves use of parameterised subprograms. Functional abstraction can be generalised 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:** 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:** States the desired effect, without stating the exact mechanism of control. 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.

(b) Architecture: Software architecture refers to the structure of the components of a program/system, their interrelationships, and guidelines governing their design and evolution over time. Software architecture can be defined as a program or computing system, which comprises of software elements, the externally visible properties of those elements, and the relationships amongst them. The software architecture must extract some information from the system and provide enough information to form a basis for analysis, decision-making, and ways of handling risk. The software architecture:

- Provides an insight to all the interested stakeholders that enable all these stakeholders to communicate amongst them.
- Highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase.
- Creates intellectual model of how the system is structured and how the components function together in the system.

Currently, representations of software architecture are informal and ad-hoc. While architectural concepts are often embodied in infrastructure to support specific architectural styles and in the initial conceptualisation of a system configuration, the lack of an explicit independently characterised architecture significantly limits the benefits of this design concept in the present scenario. Note that software architecture comprises of two elements of design model: data design and architectural design. Both these elements have been discussed later in the chapter.

(c) Patterns: A pattern describes a problem, which occurs over and over again in our environment, and then describes a solution to that problem, such 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.

(d) 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:

- **Architectural patterns:** These patterns are high-level strategies that are concerned with large-scale components and global properties and mechanisms of a system. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationship between them. Note that architectural patterns often equate to software architecture and generally affect the overall structure and the organization of a software system.
- **Design patterns:** These patterns are medium-level strategies that are used to solve design problems. They provide a scheme for refining the subsystems or components of a software system, or the relationship between them. It addresses a specific element of the design such as relationship among components or mechanisms that affect component-to-component interaction. Note that design patterns often equate to *software components*.
- **Idioms:** These patterns are low-level patterns, which are specific to a programming language. An idiom describes how to implement particular aspects of components using the features of the given language. Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning program. Note that software components often equate to design patterns with emphasis on *reusability*.

The difference between these three kinds of patterns mentioned above is according to their corresponding levels of abstraction. Architectural patterns are high-level strategies that

NOTES

NOTES

concern large-scale components, global properties, and the mechanisms of a system. Design patterns are a medium-scale strategy that elaborates some of the structure and behaviour of entities and their relationships. Idioms are paradigm-specific and language specific programming techniques that fill in low-level internal or external details of the structure or the behaviour of the component.

(e) **Modularity:** Software architecture and design patterns represent modularity. Modularity is achieved by dividing the software into uniquely named and addressable *components*, which are also known as **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.

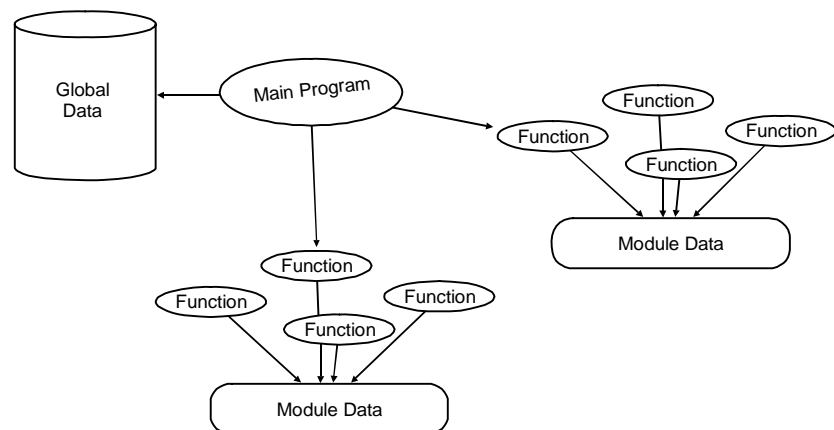


Figure 4.2 Modules in Software Programs

Modularising 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. Module-level design also referred as procedural design or component-level design is discussed in detail in Section 4.4.

(f) **Information Hiding:** Modules should be specified and designed in such a way that information contained within one module is inaccessible to other modules that do not require such information. 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 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 testing and maintenance phase. In object-oriented design, information hiding gives rise to the concepts of encapsulation and modularity, and is associated with the concept of abstraction.

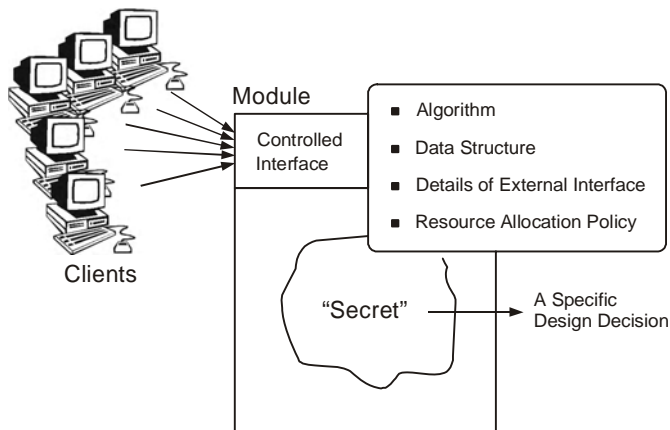


Figure 4.3 Information Hiding

Some of the advantages associated with information hiding are listed below:

- Leads to low coupling.
- Emphasises communication through controlled interfaces.
- Reduces the likelihood of adverse effects.
- Limits the global impact of local design decisions.
- Results in higher quality software.

(g) 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.

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 of *inputs*, *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.

NOTES

NOTES

(h) Refactoring: Refactoring is an important design activity that simplifies 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 refactoring process, the existing design is checked for unused design elements, redundancy, inefficient or poorly constructed algorithms and data structures, or any other flaws in the existing design that can be improved to yield a better 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 results in software that is easier to integrate, test, and maintain.

(i) 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 (shaded boxes in Figure 4.4 (a)) are used to communicate between functions and execute the functions. Structural partitioning provides the following benefits:

- Results in software that is easier to test and maintain.
- Results in less propagation of adverse affects.
- Results in software that is easier to extend.

However, the disadvantage of using horizontal partitioning is that more data has to be passed across module interface. This complicates the overall control flow of the problem especially while processing rapid movement from one function to another.

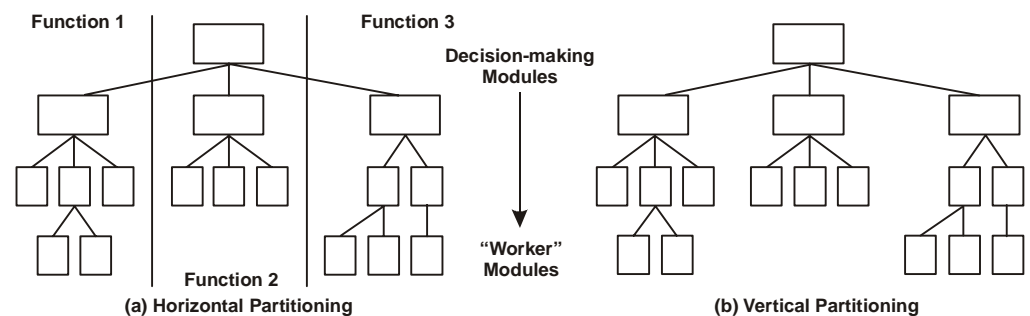


Figure 4.4 Horizontal and Vertical Partitioning

In **vertical partitioning**, the control (decision-making) modules are located at the top and work is distributed in a top-down manner. That is, top-level modules perform control function and do little processing, while low-level modules perform all input, computation and output tasks.

4.2.3 Developing a Design Model

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

- **Data design:** Creates data structure by converting data objects specified during 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.
- **Architectural design:** Specifies the relationship between structural elements of software, design patterns, architectural styles, and the factors affecting the way in which architecture can be implemented.

Check Your Progress

1. Define software design.
2. Define abstraction.
3. Define software architecture.
4. Describe modularity.

- **Component-level design/Procedural design:** Converts the structural elements of software architecture into a procedural description of software components.
- **Interface design:** Depicts how software communicates with the system that interoperates with it and with the end-users.

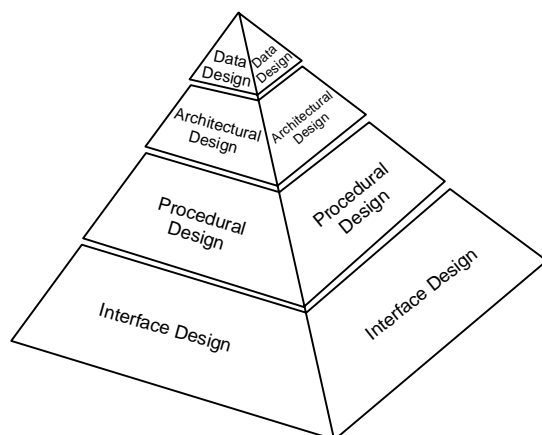


Figure 4.5 Design Model and its elements

4.3 DATA DESIGN

Data design is the first of the design activities, which leads to better program structure, effective modularity, and reduced complexity. Data design is developed by transforming the data dictionary and entity relationship diagram (identified during the requirements phase) into data structures that are required to implement the software. The data design process includes identifying the data, defining specific data types and storage mechanisms, and ensuring data integrity by using business rules and other run-time enforcement mechanisms.

The selection process may involve algorithmic analysis of alternative structures in order to determine the most efficient design or the use of a set of modules that provides operations on some representation of objects. Some principles are followed while specifying the data, which are listed below:

- All data structures and the operations to be performed should be identified.
- Data dictionary should be established and used.
- Low-level data design decisions should be deferred until late in the design process.
- The representation of data structure should be known to only those modules that directly use the data.
- A library of useful data structure and the operations that may be applied to them should be developed.
- Language should support abstract data types.

The structure of data can be viewed at three levels, namely, program component level, application level, and business level. At the **program component level**, the design of data structures and the algorithms required to manipulate them is necessary if a high-quality software is desired. At the **application level**, the translation of a data model into a database is essential to achieve the specified business objectives of a system. At the **business level**, the collection of information stored in different databases should be reorganised into data warehouse, which enables data mining that has influential impact on the business.

***Note:** Data design helps to represent the data component in the conventional systems and class definitions in object-oriented systems.*

NOTES

Check Your Progress

5. What is data design?
6. Explain the levels at which structure of data can be viewed.

NOTES

4.4 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 (also called system design), which acts as a preliminary 'blueprint' from which software can be developed. 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 building a *physical model* using recognised software engineering methods.

The physical model describes the solution in concrete and implementation terms, which is used to produce a structured set of component specifications (each specification defines the functions, inputs and outputs of the component) that are consistent, coherent and complete. An architectural design performs the following functions:

- Provides a level of abstraction at which the software designers can specify the system behaviour (such as function and performance).
- Serves as the 'conscience' for a system as it evolves. By characterizing the crucial system design assumptions, a good architectural design guides the process of system enhancement indicating what aspects of the system can be easily changed without compromising system integrity.
- Evaluates all top-level designs.
- Develops and documents top-level design for the external and internal interfaces.
- Develops preliminary versions of user documentation.
- Defines and documents preliminary test requirements and the schedule for software integration.

Architectural design is derived from the following sources:

- 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, and more are addressed. This task is cumbersome as the software engineering paradigm is shifting from monolithic, stand-alone, built-from-scratch systems to componentised, evolvable, standards-based, and product line oriented systems. Also, one key challenge for designers is to know precisely how to proceed from requirements to architectural design. To avoid all these problems, designers adopt strategies such as reusability, componentisation, platform-based, standards-based, and so on.

While the architectural design is the responsibility of developers, participants in the architectural design phase should also include user representatives, systems engineers, hardware engineers, and operations personnel. In reviewing the architectural design, project management should ensure that all parties are consulted in order to minimise the risk of incompleteness and error.

4.4.1 Architectural Design Representation

Architectural design can be represented using various models, which are listed below:

- **Structural model:** Illustrates architecture as an ordered collection of programs components.

- **Framework model:** Attempts to identify repeatable architectural design patterns, which are encountered in similar types of application. This leads to an increase in the level of abstraction.
- **Dynamic model:** Specifies the behavioural aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in external environment.
- **Process model:** Focuses on the design of the business or technical process, which must be implemented in the system.
- **Functional model:** Represents 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 models, and a dynamic process model that shows how the system is organized into process at run-time. This document gives the developers' solution to the problem stated in the software requirements specification (SRS) but avoids the detailed consideration of software requirements that do not affect the structure. In addition to ADD, other outputs of the architectural design are:

- Progress reports, configuration status accounts and audit reports.
- Various plans for detailed design phase, which includes:
 - Software project management plan.
 - Software configuration management plan.
 - Software verification and validation plan.
 - Software quality assurance plan.

4.4.2 Architectural Styles

Architectural styles define a family of systems in terms of a pattern of structural organization. It also characterises a family of systems that are related by sharing 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 reengineered, then imposition of an architectural style results in fundamental changes in the structure of the system. This change also includes reassignment of the functionality performed by the components.

By constraining the design space, an architectural style often permits specialised and style-specific analyses. Also, an architectural style makes it easier for other stakeholders to understand a system's organization if conventional structures are used.

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

- *Computational components* (such as clients, server, filter, database) that perform a function required by the system.
- A set of *connectors* that enable interactions and co-ordination among these components (such as procedure call, events broadcast, database protocols, and pipes).
- *Constraints* that define integration of components to form a system.
- *Semantic model*, which enables software designer to understand the overall properties of a system by analysing the known properties of its constituent parts.

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. Note that the use of an appropriate architectural style promotes design reuse, leads to code reuse, and supports interoperability.

NOTES

NOTES

(a) Data-flow Architecture: A data-flow architecture is used when input data is transformed through series of computational component in order to produce the output data. Each component has a set of input and output terminals. A component reads a stream of data on its input terminal and produces a stream of data on its output terminal. Input is transformed both locally and incrementally so that output begins before input is consumed. In this type of style, components are called *filters* and connectors' conduits for the information streams are termed as *pipes*.

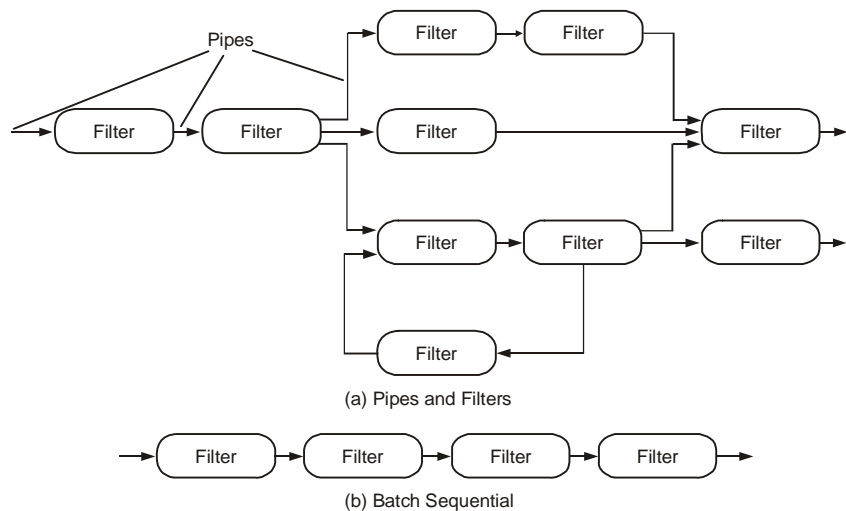


Figure 4.6 Dataflow Architecture

Each filter works as an independent entity, which may not know the identity of upstream or downstream filters. They may specify input format and guarantee what appears as an output, but they may not know which components appear at the ends of those pipes. A degenerated version occurs when each filter processes all of its input as a single entity. This is known as **batch sequential system**. In these systems, pipes no longer provide a stream of data. The best-known example of data flow architectures is Unix shell programmes where components are represented as Unix processes and pipes are created through the file system. Other examples include compilers, signal-processing systems, parallel programming, functional programming, and distributed systems. Some advantages associated with the data-flow architecture are listed below:

- Supports reusability.
- Easy to maintain and enhance.
- Supports specialised analysis and concurrent execution.

Some disadvantages associated with the data-flow architecture are listed below:

- Often lead to batch organization of processing.
- Poor for interactive applications.
- Difficult to maintain synchronisation between two related streams.

(b) Object-oriented Architecture In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data. The components of this style are the *objects* and connectors, which operate through *procedure calls* (methods). This architectural style has two important characteristics:

- Objects are responsible for maintaining the integrity of a resource.
- Representation of the object is hidden from other objects.

Some of the advantages associated with object-oriented architecture are listed below:

- Hidden implementation details allow object to be changed without affecting the accessing routine of other objects.
- Data allows designers to decompose problems into collections of interacting agents.

(c) **Layered Architecture:** A layered architecture is organised hierarchically with each layer providing service to the layer above it and serving as a client to the layer below it. In some systems, inner layers are hidden from all, except the adjacent outer layer. In this type of architectural style, connectors are defined by the protocols that determine how layers will interact. An example of this architectural style is the layered communication protocols OSI-ISO (open systems interconnection-international organization for standardization) communication system. In these systems, lower levels describe hardware connections and higher levels describe application. Layered systems support designs based on increasing levels of abstraction.

NOTES

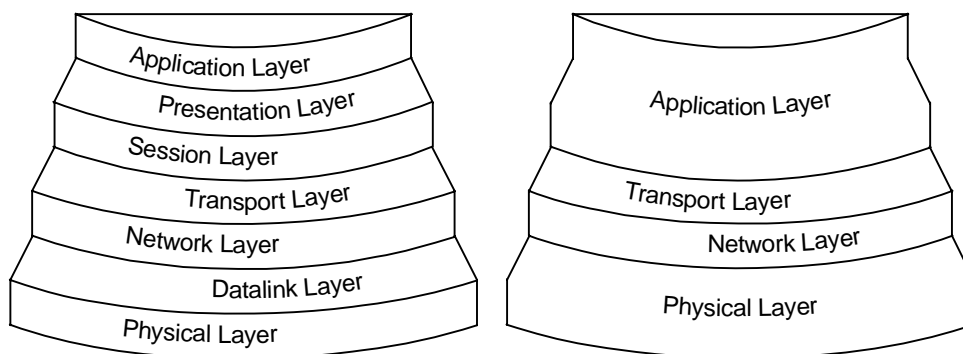


Figure 4.7 OSI and Internet Protocol Suite

(d) **Data-centered Architecture:** A data-centered architecture has two distinct components: a **central data structure** or data store (central repository), which represents the current state and a **collection of independent components** (client software), which operate on the data-store (like a database or a file). The client software accesses the data independent of any changes to the data or the actions of other client software.

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.

Control methods for these systems are of two types.

- If input transactions select the processes to execute, then a *traditional database* is used as a repository.
- If the state of the data-store is the main trigger for selecting processes, then the repository is a *blackboard*. Blackboard systems have been used for applications that require complex interpretation of signal processing for example, speech recognition and in web-based applications. A blackboard model usually has three components:
 - **Knowledge source:** Contains independent pieces of application specific knowledge. Interaction between knowledge sources takes place only through the blackboard.
 - **Blackboard data structure:** Stores data in an organized way, into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution of a problem.
 - **Control:** Driven by the state of the blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.

NOTES

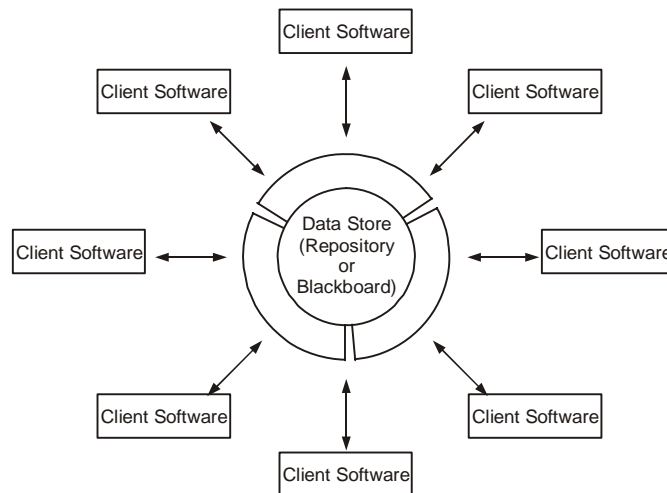


Figure 4.8 Data-centered Architecture

Some of the advantages of data centered system are listed below:

- Clients are relatively independent of each other.
- Data store is independent of the clients.
- Adds scalability (that is, new clients can be added easily).
- Supports modifiability.
- Achieves data integration in component-based development using blackboard.

(e) **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.

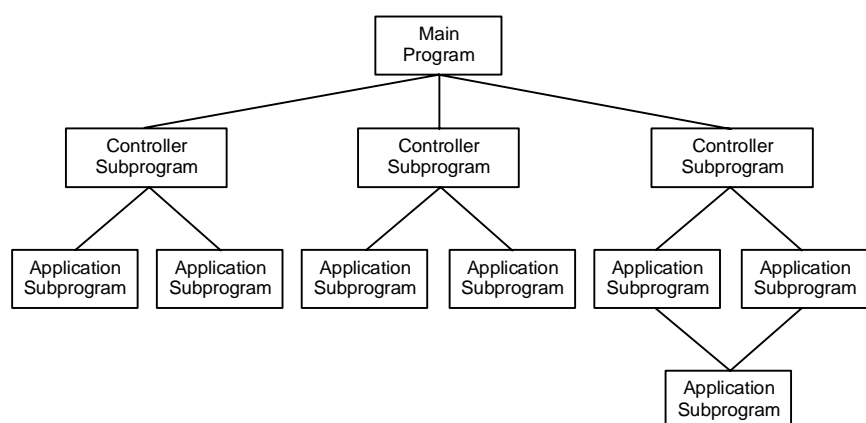


Figure 4.9 Main program/subprogram architecture

Check Your Progress

7. Define architectural design.
8. List the advantages of object-oriented architecture.
9. Mention the subtypes of call and return architecture style.

- **Remote procedure call architecture:** In this, components of main or sub program architecture are distributed over a network across multiple computers.

4.5 PROCEDURAL DESIGN

As soon as first iteration of architectural design is complete, component-level design also called **procedural design** takes place. Component-level 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. Component also known as module, resides within the software architecture and serves one of the three roles listed below:

- 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 supports the processing required in the problem domain.

Component-level design is used to define the data structures, algorithms, interface description, and communication mechanisms allocated to each module. Note that a module or a component can be defined as a modular building block for the software. However, 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, design, and testing but also allows development costs to be amortised 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 that can benefit from a common implementation or that 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.

4.5.1 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 uniquely performs given sets of function without interacting with other parts of the system. Software that uses property of functional independence is easier to develop because its function can be categorised 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.

NOTES

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.

NOTES

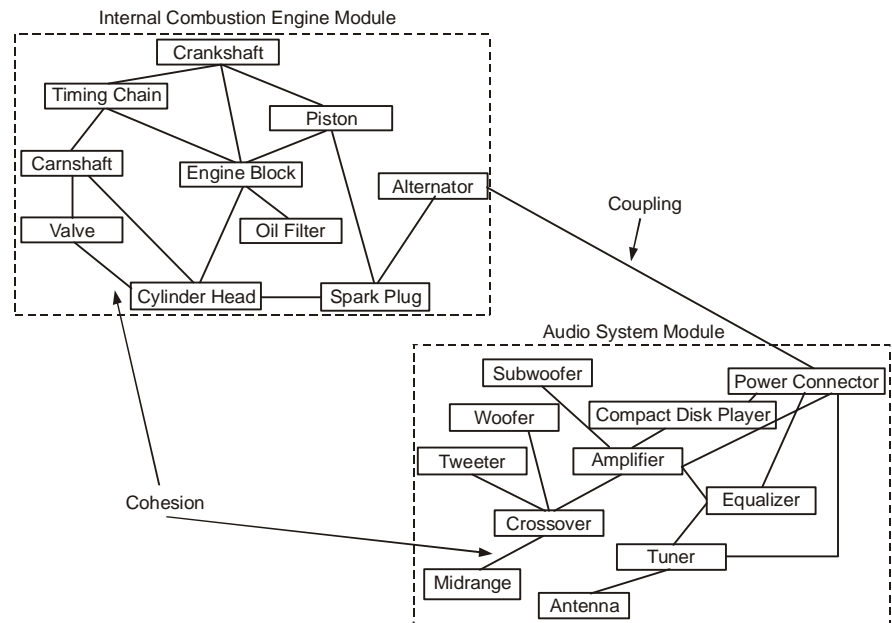


Figure 4.10 Coupling and Cohesion

(a) Coupling: Coupling is the measure of interdependence between one module and another. Coupling depends on the interface complexity between components, the point at which 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 minimises the 'ripple effect' where changes in one module cause errors in other modules. Module coupling is categorised into the following types.

- **No direct coupling:** Two modules are no direct coupled when they are independent of each other. In Figure 4.11, Module 1 and Module 2 are no directly coupled.
- **Data coupling:** Two modules are data coupled if they communicate by passing parameters. In Figure 4.11, Module 1 and Module 3 are data coupled.

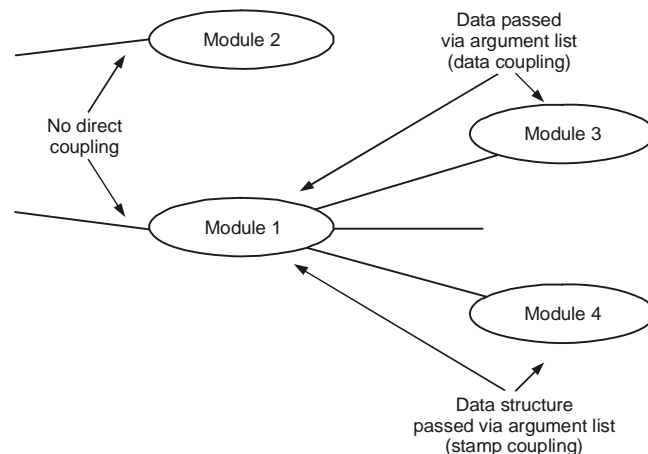


Figure 4.11 No Direct, Data, and Stamp Coupling

- **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 4.11, data structure is passed between modules 1 and 4. Therefore, Module 1 and Module 4 are stamp coupled.
- **Control coupling:** Two modules are control coupled if they communicate (passes 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 4.12, when Module 1 passes control flag to Module 2, Module 1 and Module 2 are said to be control coupled.

NOTES

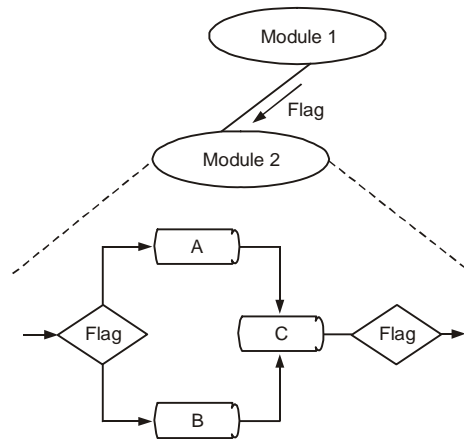


Figure 4.12 Control Coupling

- **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 4.13, 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 4.13, Modules C and Module N are common coupled.

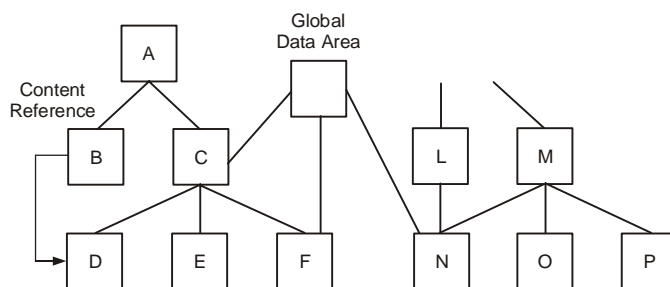


Figure 4.13 Content and Common Coupling

(b) **Cohesion:** Cohesion is the measure of strength of the association of elements within a module. A cohesive module performs a single task within a software procedure, which has less interaction with procedures in other part of the program. In practice, designer should avoid low-level of cohesion when designing a module. Generally, low coupling results in high cohesion and vice versa. The various types of cohesion are listed below:

- **Functional cohesion:** In this, the elements within the modules contribute to the execution of one and only one problem related task.

NOTES

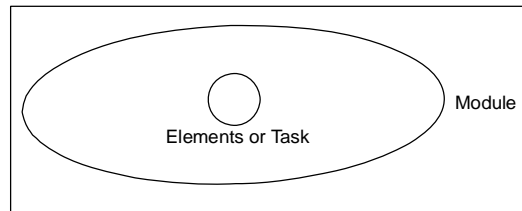


Figure 4.14 Functional Cohesion

- **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.

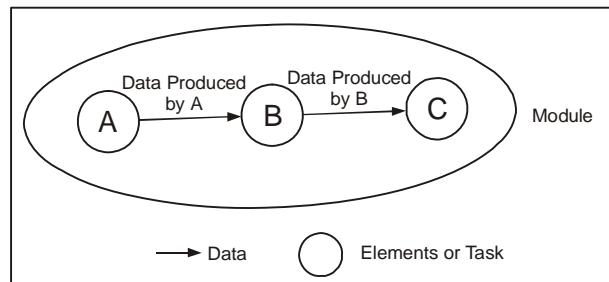


Figure 4.15 Sequential Cohesion

- **Communicational cohesion:** In this, the elements within the modules perform different functions, yet each function references the same input or output information.

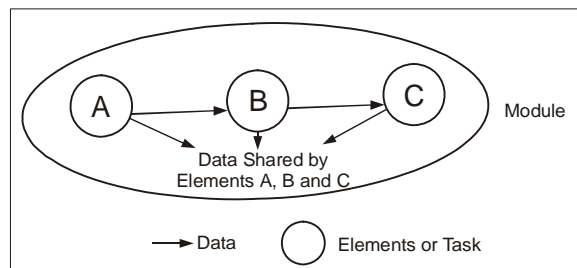


Figure 4.16 Communicational Cohesion

- **Procedural cohesion:** In this, the elements within the modules are involved in different and possibly unrelated activities.

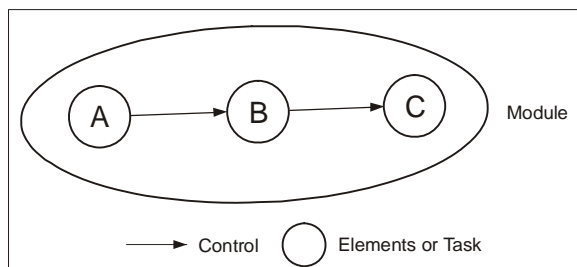


Figure 4.17 Procedural Cohesion

- **Temporal cohesion:** In this, the elements within the modules contain unrelated activities that can be carried out at the same time.

NOTES

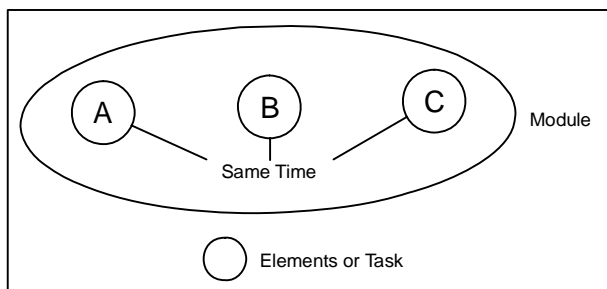


Figure 4.18 Temporal Cohesion

- **Logical cohesion:** In this, the elements within the modules perform similar activities, which are executed from outside the module.

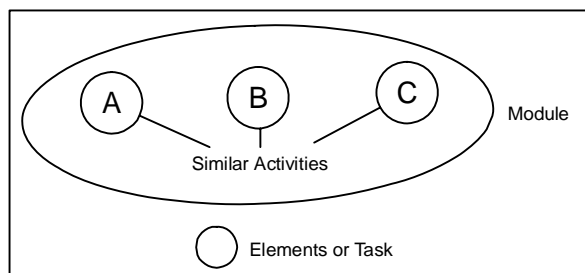


Figure 4.19 Logic Cohesion

- **Coincidental cohesion:** In this, the elements within the modules perform activities with no meaningful relationship to one another.

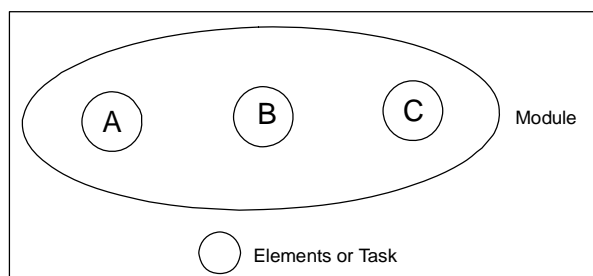


Figure 4.20 Coincidental Cohesion

After having discussed various types of cohesions, Figure 4.21 illustrates the procedure, which can be used in determining the types of module cohesion for software design.

NOTES

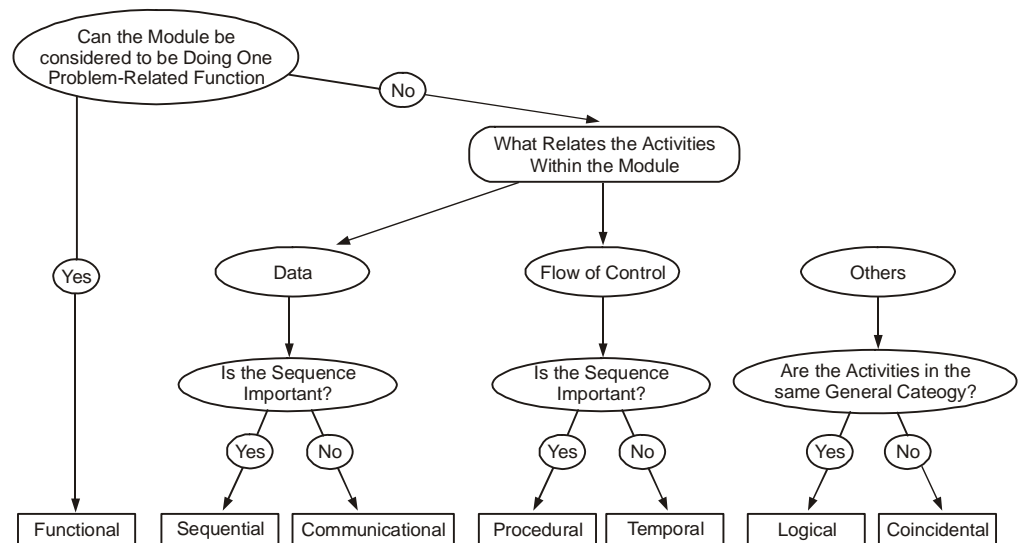


Figure 4.21 Selection Criteria of Cohesion

4.6 USER INTERFACE DESIGN

User interfaces determine the way in which users interact with the software. The user interface design creates 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 needs and wants from the user interface.

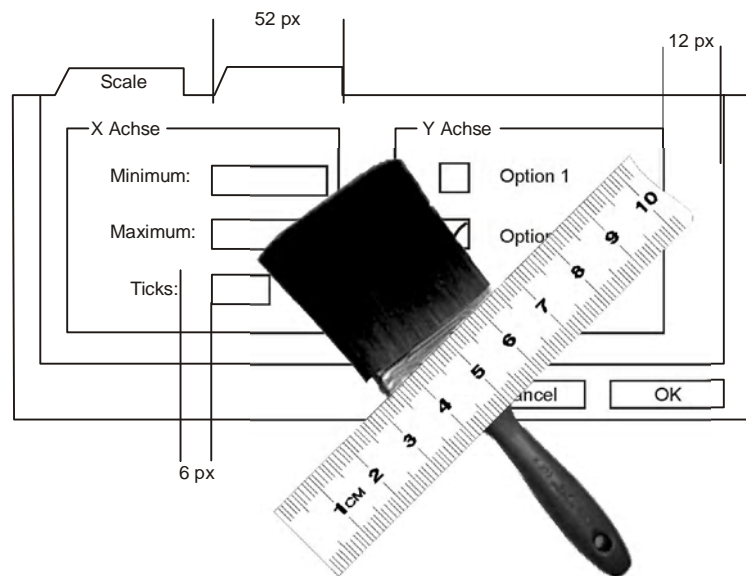


Figure 4.22 Simple User Interface Design

Since user is the 'central' while developing the software, user interface must also be central while designing the software. It is important to first know the person (user) for whom the user interface is being designed before designing the user interface. Direct contact between end-users and developers often improves the user interface design. The result of this communication helps the designers to know user's goals, skills, and needs.

Note: For a successful project, the overall software design and the user interface design proceed in lockstep. That is, each step forward on one side helps to refine the other side.

Check Your Progress

10. Define procedural design.
11. Describe functional independence.
12. Describe various roles served by components contained in the software architecture.

4.6.1 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*.

(a) Ease of Learning: Ease of learning describes how quickly and effortlessly users learn to use the software. Ease of learning is primarily important for new users. However, even experienced users face a learning experience problem when they attempt to expand their usage of the product or when they use a new version of the software. Here, the *principle of state visualisation* is applied, which states that each change in the behaviour of the software should be accompanied by a corresponding change in the appearance of the interface.

Developers of software applications with a very large feature set can expect only few users to have mastered the entire feature set. Thus, designers of these applications should be concerned about the ease of learning for otherwise experienced users. Generally to ease the task of learning, designers make use of the tools listed below:

- **Affordance:** Provides clues that suggest what a machine or tool can do and how to use it. For example, the style of a door handle on the doors of many departmental stores, offices, and shops suggest whether to pull a door or push a door to open. If the wrong style door handle is used, people struggle with the door. In this way, the door handle is more than just a tool for physically helping you to open the door; it is also an affordance showing you how the door opens. Similarly, software designers while developing user interface should offer hints as to what each part does and how it works.
- **Consistency:** Designers strive to maintain consistency within the interface. Every aspect of the interface, including seemingly minor details, such as font usage and colours is kept consistent when the behaviour is consistent. Here, *principle of coherence* (that is behaviour of the program should be internally and externally consistent) is applied. **Internal consistency** means that the program’s behaviour must make “sense” with respect to other parts of the program. For example, if one attribute of an object (for example, colour) is modified using a pop-up menu, then it is expected that other attributes of the object will also be edited in a similar manner. **External consistency** means that the program is consistent with the environment in which it runs. This includes consistency with both the operating system and other suite of applications that run within that operating system.

(b) Efficiency of Use: Once a user knows how to perform tasks, the next question is how efficiently can the user solve problems with the software? Efficiency can be evaluated reasonably only if users are no longer engaged in learning how to do the task and are rather engaged in performing the task.

Defining an efficient interface requires a deep understanding of the behaviour of target audience. How frequently do they perform the task? How frequently do they use the interface devices? How much training do they have? How distracted are they? A few guidelines help in designing an efficient interface.

- The task should require minimal physical actions. The desire of experienced users for hot keys and to shortcuts to pull-down menu actions is a well-known example of reducing the number of actions required to perform a task.
- The task should require minimal mental effort as well. A user interface, which requires the user to remember specific details will be less popular than one that remembers those details for the user. Similarly, an interface, which requires the user to make many

NOTES

NOTES

decisions, particularly non-trivial decisions, will be less popular than the one that requires the user to make fewer or simpler decisions.

(c) Aesthetically Pleasing: Today, look and feel is one of the biggest USP (unique selling point) while designing software. An attractive user interface improves the sales because people like to have things that look nice. An attractive user interface makes the user feel better (as it provides ease of use), while using the product. Many software organisations focus specifically on designing software, which has attractive look and feel so that they can lure customers/users towards their product(s).

In addition to the above-mentioned goals, there exist 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 a complex software system can be understood easily if the user interface is created in a way that resembles an already developed system. For example, the popular Windows operating system uses similar (not same) look and feel in all of its operating system so that the users are able to use it in a user-friendly manner.

4.6.2 User Interface Design Process

User interface design, like all other software design elements, is an iterative process. Each step in user interface design occurs a number of times, each 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.

1. Using information stated in the requirements document, each task and actions are defined.
2. A complete list of events (user actions), which causes the state of the user interface to change is defined.
3. Each user action is assigned iteration.
4. Each user interface state, as it will appear (look) to the user is depicted.
5. Indicate how user interprets the state of the system using information provided through the interface.

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 analysed 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, following points must be kept in mind:

- Follow the rules stated in section 4.5.1. Any interface that fails to achieve any of these rules to a reasonable degree needs to be re-designed.
- Determine how interface will be implemented.
- Consider the environment (like operating system, development tools, display properties, and so on).

4.6.3 Evaluating User Interface Design

User interface design process generates a useful interface, however, no designer should expect to achieve a high quality interface on the first go. 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. This prototype is evaluated, discrepancies are detected and accordingly re-design takes place. This process carries on until a good interface evolves.

To evaluate an interface, the prototype must include the flow of the user interface and range of options offered. The prototype does not, however, need 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: To provide a feedback for the next iteration, each interface must be evaluated in some manner. This evaluation must indicate what works well with the interface, and more importantly, what are the areas of improvement.

Some of the evaluation techniques used to evaluate the user interface design are: *use it yourself*, *colleague evaluation*, *user testing*, and *heuristic evaluation*. Each technique offers unique advantages and limitations. To a varying degree, they highlight ease of learning or efficiency issues. The third goal of aesthetic pleasing largely depends on user to user and can be best evaluated by observing what people find pleasing.

- **Use it yourself:** In this evaluation technique working through a number of the tasks defined by the requirements often indicates a myriad of problems in the initial design(s). This evaluation technique helps in identifying the entire missing pieces of the interface and significant inefficient usage issues.
- **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. However, note that unless the prototype is inherently useful in its current state, colleagues are unlikely to use the prototype for sufficient time to discover many efficiency issues.
- **User testing:** This testing is considered to be the most realistic form of interface evaluation. Users can test the prototypes, with the expected differences recorded in a feedback. The most useful feedback is often the comments the users make to each other as they try to understand how to accomplish the task. Before performing user testing, the designer should choose one or more tasks that the user will be expected to perform. Any necessary background information must be prepared in advance, with the user having sufficient time to understand this background before beginning the test. User testing is one of the best tools available for evaluating ease of learning. However, it offers little or no assistance in discovering efficiency issues.
- **Heuristic evaluation:** Such evaluations are inherently subjective, where each evaluator finds a different set of problems in the interface. Generally, experienced user interface designers are able to find more errors as compared to less experienced designers. Heuristic evaluation provides a checklist of issues, which should be considered for each iteration of user interface design. This checklist considers the following categories:
 - **Simple and natural dialogue:** An interface provides a simple and natural dialogue if it gently leads the user from one action to the next for common tasks.
 - **Graphic design and colour:** Consistency in the use of colour and other graphic design elements helps the user. The colour and font can be used to convey information subtly.
 - **Less is more:** Keep the information being presented to minimum. A cluttered interface is both aesthetically displeasing and confusing to navigate.
 - **Speak the user's dialogue:** Always use the user's terminology and phrase text from the user's perspective. For example, messages should display "you have purchased..." and not "we have sold you...". When you choose terminology and metaphors, be sure that they are consistent, both internally and with outside expectations.
 - **Minimize the user's memory load:** Minimize the user's memory load by presenting as much information as appropriate, when required by the user.
 - **Be consistent:** Consistency is one of the designer's most desired objectives. Be sure that similar visual descriptions for similar items are used.

NOTES

NOTES

- **Provide feedback:** Always provide the user with clear feedback about what the program is doing.
- **Provide shortcuts:** Providing shortcuts, such as hot keys, are an important aspect of efficiency. These shortcuts enhance the work experience for experienced users.
- **Provide error messages:** Appropriate error messages let the user know exactly what is wrong and how to fix it. The message should indicate which part of the input is incorrect.

Careful evaluation under these guidelines helps in discovering most of the learning problems and efficiency problems.

4.7 SOFTWARE DESIGN NOTATION

To represent software design, design notations are used. These notations are important as they help designers to represent modules, interfaces, hidden information, concurrency, message passing, invocation of operations and overall program structure in a comprehensive manner. A design representation serves purpose to two individuals who are listed below:

- The designers themselves who try to detect missing or inconsistent aspects of a proposed solution at the earlier stages of design.
- Other stakeholders (programmers, testers, or maintainers) who 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. Note that a software design notation can be diagrammatic, symbolic, or textual. The various notations that are commonly used are *flow charts*, *data flow diagrams*, *structure charts*, *HIPO diagram*, *decision table*, and *program design language*.

4.8 SOFTWARE 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.

4.8.1 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. Firstly, **preliminary design review** is conducted with the customers and users to ensure that the conceptual design (this design gives an idea to the user of what the system will look like) satisfies their requirements. Next, **critical design review** is conducted with analysts and other developers to check the technical design (this design is used by the developers to specify *how* the system will work) in order

Check Your Progress

13. Define user interface design.
14. Describe the rule of aesthetically pleasing.
15. Mention various evaluation techniques used to evaluate the user interface design.

to critically evaluate technical merits of the design. Next, **program design review** is conducted with the programmers in order to get feedback before the design is implemented.

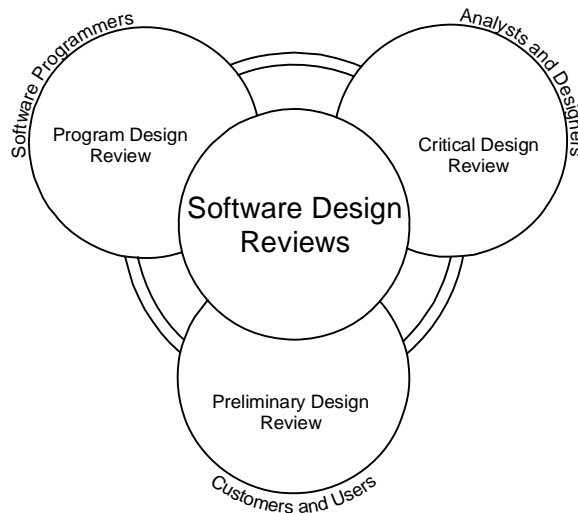


Figure 4.23 Types of Design Reviews

NOTES

(a) 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 is in conformance with 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 authorised to make necessary comments and decisions. This review team comprises of individuals listed below:

- **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, moderator is responsible for smooth functioning of the review.
- **Secretary:** Secretary is a silent observer who does not take part in the review process but instead 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 these people can advise 'fresh' ideas, address issues of correctness, consistency, and good design practice.

If any discrepancies are noted in the review process then the faults are assessed on the basis of its severity. That is, if there exists a minor fault then the fault is resolved among the review team. However, if there exists a major fault, then the review team may agree to

NOTES

revise the proposed conceptual design. Note that preliminary design review is again conducted to assess the effectiveness of the revised (new) design.

(b) Critical Design Review: Once the preliminary design review is successfully completed and the customer(s) is satisfied with the proposed design, 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 are used (sometimes both) to evaluate the alternative design strategies and how and why the major design decisions have been taken. Just like the preliminary design review, to carry out critical design review a review team is formed. In addition to the team members involved in preliminary design review, this team also comprises of individuals listed below:

- **System tester:** Understands the technical issues of design and compares with design created for similar projects
- **Analyst:** Responsible for writing system documentation.
- **Program designers for this project:** Understands the design in order to derive detailed program designs.

Note: This review does not involve customers.

Similar to preliminary design review if any discrepancies are noted in the critical design review process then the faults are assessed on the basis of its severity. A minor fault is resolved among the review team. While if there exist a major fault, then the review team may agree to revise the proposed technical design. Note that critical design review is again conducted to assess the effectiveness of the revised (new) design.

(c) Program Design Review: On successful completion of critical design review, program design review is conducted to get feedback on the designs before implementation (coding) begins. This review is conducted between the designers and developers with the purpose to:

- Ensure that the detailed design is feasible.
- Ensure that the interface is consistent with architectural design.
- Specify whether design is amenable to implementation language.
- Ensure that structured programming constructs are used throughout.
- Ensures that the implementation team will be able to understand the proposed design.

To conduct a program design review, a review team is formed, which comprises of system designers, system tester, moderator, secretary, and analyst. In addition, the review team 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 comment and suggestions. Note that a successful program design review presents considerations relating to coding plans before coding begins.

4.8.2 Software Design Review Process

Design reviews are considered important as in these reviews the product is logically viewed as the collection of various entities/components and various use cases. These reviews occur at all levels of the software design from top-level through detailed, and encompass

through all parts of the software units. Generally, the review process comprises of three criteria's, which are listed below:

- **Entry criteria:** Software design is ready for review.
- **Activities:** This criteria involves steps listed below:
 1. Select software design review team participants, assign roles, and prepare schedule for the review.
 2. Distribute 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. Recorder documents defects, action items, and recommendations.
 4. 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:** Software design is approved.

NOTES

4.8.3 Evaluating Software Design Reviews

The software design review process is beneficial for everyone as the faults can be detected at the 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 emphasises that the common objective of developing a high rated design is achieved. To check the effectiveness of the design, the review team members should address the questions listed below:

- Is the solution achieved with the developed design?
- Is design reusable?
- Is design well structured, and easy to understand?
- Is 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 the above-mentioned questions, if the proposed system is developed using a phased development (like waterfall model), then the phases should be interfaced sufficiently so that easy transition takes place from one phase to the other.

4.9 SOFTWARE DESIGN DOCUMENTATION (SDD)

IEEE defines software design documentation as “*a description of software created to facilitate analysis, planning, implementation, and decision making. This design description is used as a medium for communicating software design information, and can be considered as a blueprint or model of the system*”.

While developing SDD, engineers should describe the design in sufficient detail that no further refinement of the tasks, databases, inter-task communications, libraries, module structure and interfaces, data structures, and databases is required in the code.

The information to be included in software design document depends on a number of factors, such as the type of software being developed and the approach used in its

development. A number of standards have been suggested to develop software design document, however, the most widely used standard is by IEEE (see Figure 4.24), which acts as a general framework. This general framework can be customised and adapted to meet the needs of a particular organization.

NOTES

1.	Scope
2.	References
3.	Definition
4.	Purpose of an SDD
5.	Design Description Information Content
5.1	Introduction
5.2	Design entity
5.3	Design entity attributes
5.3.1	Identification
5.3.2	Type
5.3.3	Purpose
5.3.4	Function
5.3.5	Subordinates
5.3.6	Dependencies
5.3.7	Interface
5.3.8	Resources
5.3.9	Processing
5.3.10	Data
6.	Design Description Organisation
6.1	Design views
6.1.1	Decomposition description
6.1.2	Dependency description
6.1.3	Interface description
6.1.4	Detailed design description

Annex A Sample table of contents for an SDD

Annex B Guidelines for compliance with IEEE/EIA 12207.1-1997

Figure 4.24 Software Design Description

The SDD consist of several sections, which are discussed below:

- **Scope:** Identifies the release or version of the system being designed. The system is divided into modules; relationship between them and functionalities will be defined. Each iteration of the SDD document describes and identifies the software modules to be added or changed in a release.
- **References:** Lists references (both hardware and software documents and manuals) used in the creation of the SDD that may be of use to the designer, programmer, and user or to management personnel. This document is also considered useful for the readers of the document. In this section, any references made to the other documents including references to related project documents, especially the SRS is also listed. In addition, the existing software documentation (if any) is also listed.
- **Definition:** Provides a glossary of technical terms used in the document along with their definitions.
- **Purpose:** States the purpose of this document and its intended audience. This is primarily meant for the individuals who will be implementing the system.
- **Design description information content:** Consists of the sub-sections listed below:
 - **Introduction:** Since SDD is a representation of the design to be implemented, it should partition the system into design entities and describe the important properties and relationship among those entities.

NOTES

- **Design entity:** A design entity is a component of a design that is different in terms of structure and functions. The objective of creating design entities is to divide the system into separate components that can be considered, changed, and implemented. Each design entity possesses a unique name, purpose, and function but have common characteristics.
- **Design entity attributes:** A design entity attribute is a property of the design entity, which provides factual information regarding the entity. Every attribute has an attached description, which includes references and design considerations. The attributes and their associated information are listed in Table 4.1.

Table 4.1 Attributes and Description

Attributes	Description
Identification	Identifies name of the entity. All the entities have a unique name.
Type	Describes the kind of entity. This specifies the nature of the entity.
Purpose	Specifies why the entity exists.
Function	Specifies what the entity does.
Subordinates	Identifies subordinate entity of an entity.
Dependencies	Describes relationships that exist between one entity and other entities.
Interface	Describes how entities interact among themselves.
Resources	Describes elements used by the entity that are external to the design.
Processing	Specifies rules used to achieve the specified functions.
Data	Identifies data elements that form part of the internal entity.

- **Design description organization:** Consists of the following sub-section:
 - **Design views:** Design views provide a comprehensive description of the design in a concise and usable form that simplifies information access and assimilation. There exist a number of ways to view the design where every design view represents a distinct concern about a system. The various design views and their attributes are listed in Table 4.2.

Table 4.2 Design views and their description

Design View	Description	Attribute
Decomposition description	Partitions the system into design entities	Identification, type, purpose, function, and subordinate
Dependency description	Describes relationships between entities.	Identification, type, purpose, dependencies, and resources
Interface description	Consists of list that is required by the stakeholders (designers, developers and testers) in order to design entities.	Identification, function, and interfaces
Detail description	Describes internal details of the design entity.	Identification, processing, and data

4.10 CASE STUDY: HIGHER EDUCATION ONLINE LIBRARY SYSTEM

NOTES

The objective of the Higher Education online library system is to provide an efficient, modular design that will reduce the system's complexity, facilitate change, and result in an easy implementation. This can be accomplished by designing a strongly cohesion system with minimal coupling. Also, the system will provide interface design models that are consistent, user friendly, and will provide straightforward transitions through the various system functions.

The Higher Education system is developed keeping in mind current and forthcoming technologies. The Internet must be able to communicate with a browser client in HTML, ASP as well as JavaScript. The server must run on a Windows 2000 server, or higher. The client's computer must have Windows XP operating system. The entire structure of the design of the online library system is given below in Table 4.3.

Table 4.3 Design Structure of Online Library System

Design	Description
Data design	Includes an enhanced ERD as well as the data object design and the data file design.
System architecture design	Includes detailed diagrams of the system, server, and client architecture.
Procedural design	Includes the functional partitioning from the requirements specifications document, and goes into great detail to describe each function (module/component).
User interface design	Includes the graphical user interfaces that will be seen by the user when operating the Higher Education online library system.

4.10.1 Data Design

The data object USER ACCOUNT contains four types of users: STUDENT, FACULTY, LIBRARY STAFF, and ADMINISTRATOR. All of these accounts type have an inheritance relationship with the USERACCOUNT data object. MEDIA RESOURCES is the central data object and has objects related by both inheritance and association. The data object BOOKS, MAGAZINES/PERIODICALS, and MULTI-MEDIA are all types of MEDIA RESOURCES (inheritance). All other data objects are related to MEDIA RESOURCES by association.

Data Objects: The various data objects that make up the online library system are listed below:

- **Student object:** Associates with book and multi-media object when items are checked out or reserved.
- **Faculty object:** Contains information such as the faculty's full name, social security number, PIN number, e-mail address, and so on. This object associates book and multi-media object when items are checked out or reserved.
- **Library Staff object:** Represents a staff member in the user database object.
- **Administrator object:** Represents an administrator.
- **Book object:** Represents a book in the media database. This object is updated when a book is issued, returned, or reserved.
- **Status object:** Associates the book and user objects.

- **User Database object:** Maintains data for the all user types in the database.
- **Multi-media object:** Contains information about a multi-media item including title and subject and provides a unique index number, which serves as a primary key in the database.

NOTES

4.10.2 Architectural Design

The Higher Education system is a client-server based system, which contains a number of layers namely, user interface, internet/local area network (LAN) communication, functional service, and data storage layers. Data transfers occur in both directions in the system. The users input or data request is sent using either an Internet browser or through the Windows client. This data then connects to the system either through the internet or, in the case of an onsite connection, through the LAN connection of an internet connection, the data is required to pass through the system's firewall, for security purposes, prior to connecting to the web server. Local personnel, once validated within the system, will be connected directly to the application server. In the functional services layer, the data input or request is routed to the appropriate functional module in accordance with the user's login and account type. Through these modules, the users will interact with the database through the SQL server.

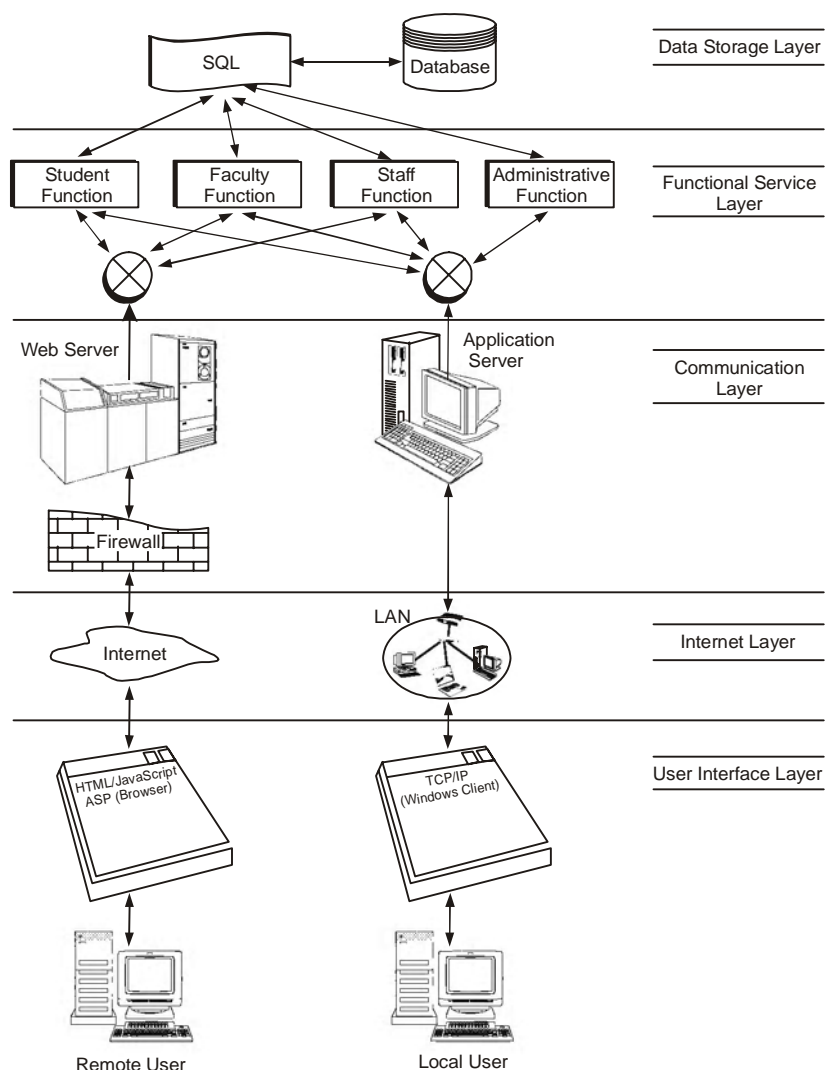


Figure 4.25 System Architecture

NOTES

(a) Server Architecture: The server architecture contains two logical servers namely web server and application server. Web server will interface with remote users, while the application server will interface with local users. The web server will communicate using active server pages (ASP) and hypertext markup language (HTML) as shown in the communication interface block within the following diagram. While, application server will communicate with local users through transmission control protocol (TCP)/internet protocol (IP). Both logical servers will have common functionality in order to facilitate all users, and will interact with the database through structured query language (SQL)/application programming interface (API).

(b) Client Architecture: The client architecture is available for the Microsoft Windows only. Client architecture resides above the Windows API layer, which interfaces with the operating system. Utility functions include print, tool bar, and help functions.

4.10.3 Procedural Design

When examining an existing information system or analyzing the information that is going to be designed, it is important to recognize what the data is, where the data comes from, how it passes from one point to another within the information system, and how it will be used by the intended audience or user. The following data flow diagrams (DFDs) represent the movement of data within the system. They concentrate less on the actual functions and data constructs of programmers and more on the general processes inherent to the overall system. Note that the amount of detail specified in this case study includes a level two representation for all the functions and diagrams include references to additional levels wherever applicable.

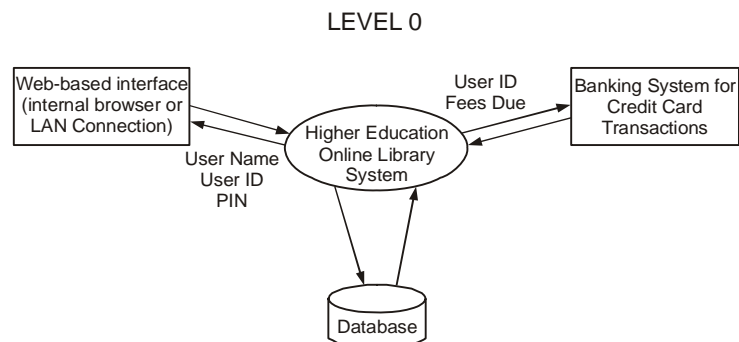


Figure 4.26 Context-level Diagram

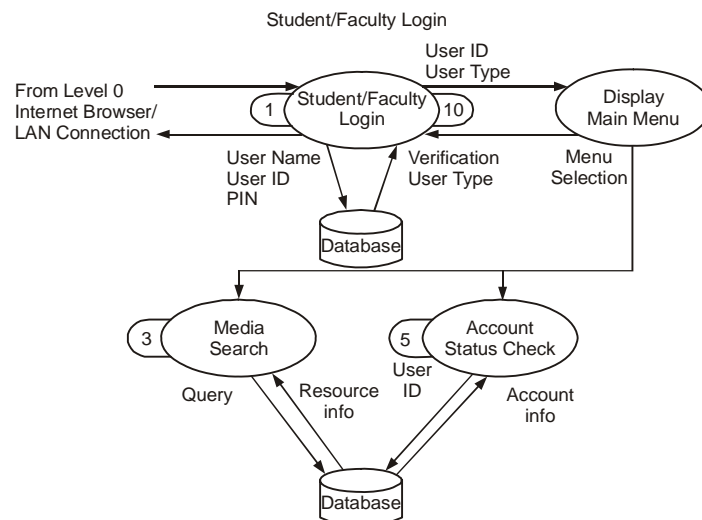


Figure 4.27 1 level Data Flow Diagram

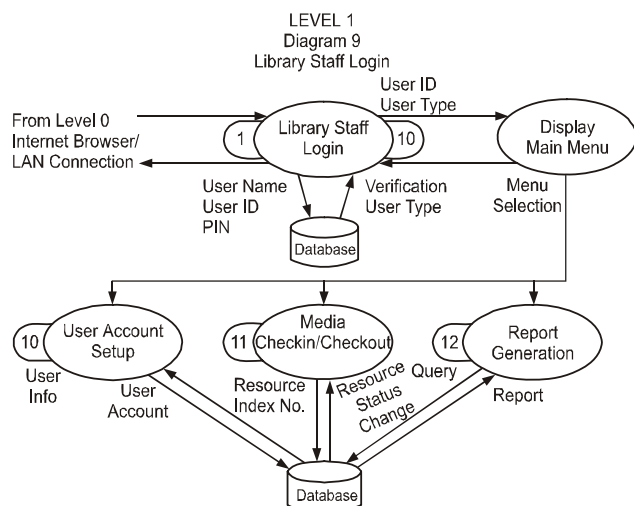


Figure 4.28 1 level Data Flow Diagram

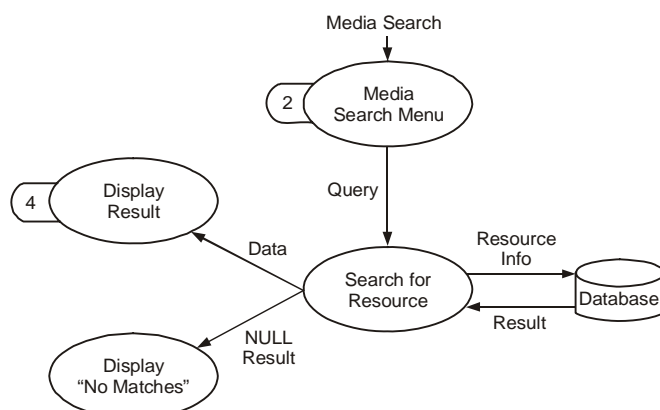


Figure 4.29 2 level Data Flow Diagram

Note that the above shown diagrams are not complete depiction of the DFDs that can be drawn for the Higher Education online library system. In fact, many levels of DFDs can be drawn for different functions (such as media reservation, account status check, late fees due/payment, login, user account set up). The functional partitioning of the entire system is shown in Figure 4.30. Using this functional partitioning various levels of DFDs for various functions can be easily drawn.

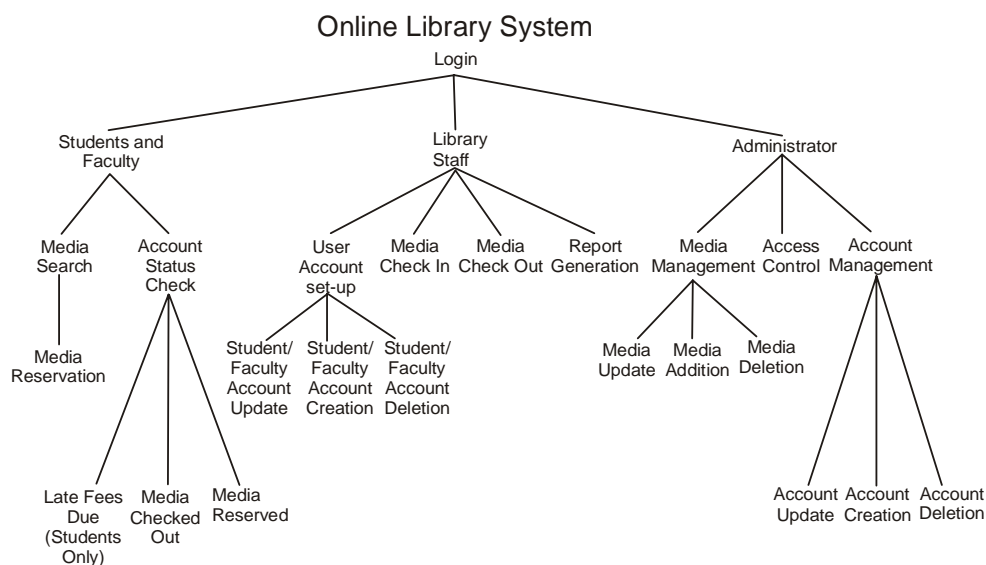


Figure 4.30 Functional Partitioning

NOTES

Check Your Progress

16. Why are software design notations used?
17. Design representation serves the purpose for two individuals. Name them.

Component/Functional Description: The various functions used in the proposed system are listed in Table 4.4.

Table 4.4 Functional Description of Higher Education Online Library System

NOTES

Function	Function Name	Description
Function 1	Login Function	Provides security and to control the user's level of access.
Function 2	Media Search Function	Searches the media database for books, magazines/periodicals, and multi-media.
Function 3	Media Reservation Function	Allows users to reserve media resources that are currently checked out.
Function 4	Account Status Check Function	Allows users to check the status of their library account.
Function 5	Overdue Fee Payment Function	Allows users to pay overdue fees through online banking system.
Function 6	User Account Set-up Function	Allows library staff to add, delete, and update user accounts.
Function 7	Media Check in/Check out Function	Allows library staff to check media in and out.
Function 8	Report Generation Function	Allows library staff and administrator to generate reports.
Function 9	Media Management Function	Allows administrator to add, delete, and update media resources.
Function 10	Access Control Function	Controls the users level of access and provides user verification.
Function 11	Account Management Function	Allows administrator to add, delete, and update library staff accounts.

4.10.4 User Interface Design

This section provides the graphic user interface for the online library system. The interface design for each screen is based on the functionality described in the function description section. References are provided as appropriate to the corresponding function descriptions. Student and faculty users will be able to log onto the system from computers both within the library or from any computer connected to the Internet. The library's computers will access the system through a LAN connection while remote computers will access the system through an Internet browser. The user interfaces displayed in this document will reflect the screens that will be seen when using the library's computers. The user interface will be almost identical when viewed through the Internet browser. The user login will be the same for all types of users. The access control function will determine the level of access based on the user type. The user type will be triggered by the user ID, and the appropriate menu will be displayed. The following screen will be displayed for the initial login to the system (Function 1).

Higher Education Online Library System

Welcome to the Library System ...
Please log in

,
Last Name First Name

- -
Library Membership Number

PIN

Figure 4.31 Login in

The data submitted will be verified via the access control function. If the user name or library membership number does not match the PIN, an error message will be displayed. The user will have the option to try again, or cancel the operation. Upon successful login, students and faculty will see the options as shown in Figure 4.32.

Higher Education Online Library System

Please make Selection

Figure 4.32 Successful Login in

If the user selects **Perform Media Search** (Function 2), the following screen will be displayed to enter the user's search criteria.

Higher Education Online Library System

Please Input your Search Criteria

Title

Author

Subject

ISBN No.

Publication

Figure 4.33 Performing Media Search

NOTES

NOTES

If no matches are found, then a message is displayed, saying, “There were no items matching your request. Please try again”. Now, the user will be given the opportunity to provide new search criteria. If the searched book is found then the database will be queried and all matches will be displayed. If the user wishes to reserve a book then reserve button can be used to reach the reservation screen (Function 3). This is shown in Figure 4.47.

<u>Title</u>	<u>Author</u>	<u>Locator ID</u>	<u>Checked Out</u>	<u>Reserved</u>
Introduction to Computer Science	Maya Nandita	1136.89	Y	N <input type="button" value="Reserve"/>
Introduction to Software Engineering	Nitin Sanchita	4298.12	N	N <input type="button" value="Reserve"/>
Introduction to Information Technology	Ira Anshu	2521.51	Y	N <input type="button" value="Reserve"/>
Introduction to Networking	Tushar Alkesh	3221.91	Y	N <input type="button" value="Reserve"/>
Introduction to Computer Applications	Shilpi Sarita	5171.11	N	N <input type="button" value="Reserve"/>

[More Matches...](#)

Figure 4.34 Searched matches

The user can also check their account status by selecting **Check Account Status** (Function 4) from the main menu. The user may check the status of media that is currently issued or reserved to them. The title, author, and due date for each item will be displayed.

The books currently issued to you are:

<u>Title</u>	<u>Author</u>	<u>Date Due</u>
Mastering Web Design	Vipul Pankaj	01/12/05
Introduction to Internet Basics	Prashant	31/11/05

The books currently reserved by you are:

<u>Title</u>	<u>Author</u>	<u>Expected Availability</u>
Computer Programming in 'C'	Dev	05/12/05
Using Unix	Rakesh Sanjay	12/12/05

Figure 4.35 Current User Status

In addition to the above-mentioned functions, there can be many such functions, which can be shown in the *user interface design*. However, since the objective of this case study is to make students understand the design phase in a clear and concise manner, rest of the interface design is left for the students to prepare themselves.

4.11 OBJECT-ORIENTED CONCEPTS

Object-oriented approach is the latest and the most adopted approach for developing software nowadays. Unlike procedural approach of software development, this approach views a problem in terms of *objects* rather than *procedures*. This approach models the real world objects very well. It emphasizes on data rather than functions or procedures. The behaviour of the system is achieved by exchanging messages among these objects. The state of the system is the combined state of all the objects in it. The object-oriented approach uses some fundamental terms and concepts. They are discussed below.

- **Object:** It is defined as an instance of a particular class. It is an identifiable entity either physical, software or even conceptual with a well-defined boundary. It consists of a state and and behaviour. The **state** of an object is one of the possible conditions that an object can exist in and is represented by its characteristics or attributes. The **behaviour** of an object determines how an object acts or behaves and is represented by the operations that it can perform. An object is what actually runs in the computer. They are the building blocks of object-oriented programming. Although, two or more objects can have same attributes, still they are separate and independent objects with their own **identity**.
- **Class:** It is a collection of similar objects. It can be treated as a blueprint or a template from which individual objects are created. In object-oriented programming, the attributes of an object are defined as variables and behaviour of an object is represented by functions or methods. All the objects in a given class are identical in form and behaviour but may contain different data in their variables. Figure 4.36 illustrates the example of a class *car* with attributes *colour*, *gears*, *speed*, *model*, *doors* and *door_locks* and functions *running*, *applying brakes* and *increasing speed*. It also shows the objects *Santro* and *Maruti 800* of class *car* that have the attributes *blue colour*, *5 gears*, *60 kmph speed*, *model 99*, *4 doors* and *2 door_locks* and *white colour*, *4 gears*, *50 kmph speed*, *model 95*, *4 doors* and *1 door_locks* respectively.

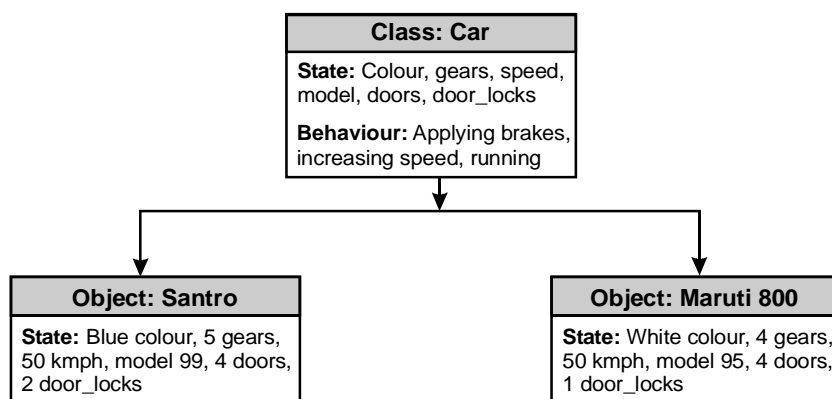


Figure 4.36 Class and Objects

Abstraction: It is a mechanism used to reduce or hide unimportant details and represent only the essential features so that one can focus on few concepts at a time. It allows us to manage complex systems by concentrating on the essential features only. In object-oriented approach, one can abstract both data and functions. The data or the functions that are to be hidden from the user can be put in the *private* section of the class. 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 is not concerned with the internal details of the car like motor, engine, wiring, etc.

NOTES

Check Your Progress

18. Define software design review.
19. Mention various steps of software design reviews.
20. What is the purpose behind conducting program design review?

NOTES

Encapsulation: It is the technique of binding or keeping together the data structures and the methods or functions (that act on the data) in a single unit called a class. Encapsulation is the way to implement data abstraction. Well-encapsulated objects act as a “black box” for other parts of the program, that is, they provide a service to the objects that interact with it, but the calling objects do not need to know the details of how the service is provided. The functions act as the interface to access the data. If the implementation of the function is changed at any time, it makes no difference to the user as long as the interface remains the same. For example, all the data related to a car and the functions associated with it are combined to form a class *car*.

Inheritance: As stated earlier, all the objects of similar kind are grouped together to form a class. But, sometimes the situation arises when the different kinds of objects have certain (not all) common characteristics. Object-oriented programming allows classes to *inherit* commonly used state and behaviour from other classes. **Inheritance** can be defined as the process whereby one class acquires characteristics from one or more other class. If a class acquires properties from a single class, it is termed as **single inheritance** and if it acquires characteristics from two or more classes, it is known as **multiple inheritance**. The class, which is inherited by other classes, is known as super-class or base class and the class, which inherits the properties of the base class, is called sub-class or derived class. In other words, the super-class is the generalization of a collection of classes related to it and the sub-class is the specialized version of the base class. The main advantage of inheritance is the code-reusability. Software developers can simply reuse the existing classes having the same behaviour that they need in the new software, instead of writing a new code. For example, the class *car* is inherited from the class *automobiles* and *automobiles* is further inherited from the class *vehicles*.

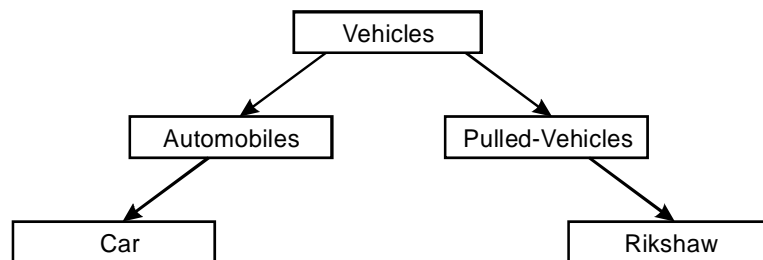


Figure 4.37 Inheritance

Polymorphism: Polymorphism (from the Greek meaning “having multiple forms”) is the ability of an entity such as variable, function or a message to be processed in more than one form. It can also be defined as the property of an object belonging to a same or different class to respond to the same message or function in a different way. For example, if a message *change_gear* is passed to the class *vehicles* then all the automobiles will behave alike but the vehicles belonging to the class *pulled_vehicles* will not respond to the message.

4.12 LET US SUMMARIZE

1. Software design is a software engineering activity where software requirements are analyzed in order to produce a description of the internal structure and organization of the system that serves as a basis for its construction (coding).
2. Software design principles act as a framework for the designers to follow a good design practice.
3. There are various software design concepts, which lay a foundation for the software design process.

4. 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.
5. Software architecture refers to the structure of the components of a program/system, their interrelationships, and guidelines governing their design and evolution over time.
6. A pattern describes a problem, which occurs over and over again in our environment, and then describes a solution to that problem, such that the solution can be used again and again. Thus, each pattern represents a reusable solution to a recurring problem.
7. Software architecture and design patterns represent modularity. Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as modules.
8. Modules should be specified and designed in such a way that information contained within one module is inaccessible to other modules that do not require such information. The way of hiding unnecessary details is referred to as information hiding.
9. 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.
10. Refactoring is an important design activity that simplifies 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.
11. When the architectural style of a design follows a hierarchical nature, the structure of the program can be partitioned either horizontally or vertically.
12. Data design creates data structure by converting data objects specified during 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.
13. Architectural design specifies the relationship between structural elements of software, design patterns, architectural styles, and the factors affecting the way in which architecture can be implemented.
14. Component-level design/procedural design converts the structural elements of software architecture into a procedural description of software components.
15. Interface design depicts, how software communicates with the system that interoperates with it and with the end-users.
16. To represent software design, design notations are used. These notations are important as they help designers to represent modules, interfaces, hidden information, concurrency, message passing, invocation of operations and overall program structure in a comprehensive manner.
17. The various notations that are commonly used are flow charts, data flow diagrams, structure charts, HIPO diagram, decision table, and program design language.
18. 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.
19. 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 is in conformance with the requirements specified by the users.
20. Once the preliminary design review is successfully completed and the customer(s) is satisfied with the proposed design critical design review is conducted.
21. On successful completion of critical design review, program design review is conducted to get feedback on the designs before implementation (coding) begins.

NOTES

Check Your Progress

21. Define software design documentation.
22. Mention the sections that forms a part of software design documentation.

22. Software design documentation is a description of software created to facilitate analysis, planning, implementation, and decision-making. This design description is used as a medium for communicating software design information, and can be considered as a blueprint or model of the system.

NOTES

4.13 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. Software design is a software engineering activity where software requirements are analyzed in order to produce a description of the internal structure and organization of the system that serves as a basis for its construction (coding). 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. 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”.
3. Software architecture refers to the structure of the components of a program/system, their interrelationships, and guidelines governing their design and evolution over time. Software architecture can be defined as a program or computing system, which comprises of software elements, the externally visible properties of those elements, and the relationships amongst them.
4. Modularity is achieved by dividing the software into uniquely named and addressable *components*, which are also known as **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.
5. Data design is developed by transforming the data dictionary and entity relationship diagram (identified during the requirements phase) into data structures that are required to implement the software. The data design process includes identifying the data, defining specific data types and storage mechanisms, and ensuring data integrity by using business rules and other run-time enforcement mechanisms.
6. The structure of data can be viewed at three levels, namely, program component level, application level, and business level. At the **program component level**, the design of data structures and the algorithms required to manipulate them is necessary if a high-quality software is desired. At the **application level**, the translation of a data model into a database is essential to achieve the specified business objectives of a system. At the **business level**, the collection of information stored in different databases should be reorganized into data warehouse, which enables data mining that has influential impact on the business.
7. 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 (also called system design), which acts as a preliminary ‘blueprint’ from which software can be developed.
8. Some of the advantages associated with object-oriented architecture are listed below:
 - Hidden implementation details allow object to be changed without affecting the accessing routine of other objects.
 - Data allows designers to decompose problems into collections of interacting agents.
9. 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 sub-styles:

- **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.
 - **Remote procedure call architecture:** In this, components of main or sub program architecture are distributed over a network across multiple computers.
10. As soon as first iteration of architectural design is complete, component-level design. Component-level design is created by transforming the structural elements defined by the software architecture into procedural descriptions of software components.
 11. 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 uniquely performs given sets of function without interacting with other parts of the system.
 12. Component also known as module, resides within the software architecture and serves one of the three roles listed below:
 - 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 supports the processing required in the problem domain.
 13. User interfaces determine the way in which users interact with the software. The user interface design creates 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.
 14. Today, look and feel is one of the biggest USP (unique selling point) while designing software. An attractive user interface improves the sales because people like to have things that look nice. An attractive user interface makes the user feel better (as it provides ease of use), while using the product. Many software organisations focus specifically on designing software, which has attractive look and feel so that they can lure customers/users towards their product(s).
 15. Various evaluation techniques used to evaluate the user interface design are: use it yourself, colleague evaluation, user testing, and heuristic evaluation.
 16. Design notations are used to represent software design. These notations are important as they help designers to represent modules, interfaces, hidden information, concurrency, message passing, invocation of operations and overall program structure in a comprehensive manner.
 17. Design representation serves the purpose for two individuals, which are listed below:
 - The designers themselves who try to detect missing or inconsistent aspects of a proposed solution at the earlier stages of design.
 - Other stakeholders (programmers, testers, or maintainers) who try to understand the designer's intent.
 18. 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”.

NOTES

NOTES

19. The review process is carried out in three steps namely preliminary design review, critical design review, and program design review.
20. Program design review is conducted between the designers and developers with the purpose to:
 - Ensure that the detailed design is feasible.
 - Ensure that the interface is consistent with architectural design.
 - Specify whether design is amenable to implementation language.
 - Ensure that structured programming constructs are used throughout.
 - Ensure that the implementation team will be able to understand the proposed design.
21. IEEE defines software design documentation as “a description of software created to facilitate analysis, planning, implementation, and decision making. This design description is used as a medium for communicating software design information, and can be considered as a blueprint or model of the system”.
22. Software design documentation consists of various sections, namely scope, references, definition, purpose, design description information content, and design description organization.

4.14 QUESTIONS AND EXERCISES

I. Fill in the Blanks

1. Software design principles act as a _____ for the designers to follow a good design practice.
2. _____ 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.
3. Software architecture and design patterns represent _____.
4. The way of hiding unnecessary details is referred to as _____.

II. Multiple Choice Questions

1. The various notations that are commonly used are _____, data flow diagrams, structure charts, HIPO diagram, decision table, and program design language.
 (a) Circles (b) Flow charts (c) Gantt charts (d) Rayleigh curves

III. State Whether True or False

1. 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.
2. Modules should be specified and designed in such a way that information contained within one module is accessible to other modules that do not require such information.
3. Software design is a software engineering activity where software requirements are analyzed in order to produce a description of the internal structure and organization of the system that serves as a basis for its analysis.
4. Once the preliminary design review is successfully completed and the customer(s) is satisfied with the proposed design critical design review is conducted.

IV. Descriptive Questions

1. Develop a procedural design for a program that accepts two arbitrarily long integers and produces their sum.

2. Define module coupling and cohesion. Discuss all types of coupling supported by diagrams.
3. How can one test the user interface? Discuss the merits and demerits of user interface rules.
4. Write short notes on:
 - (a) Stepwise Refinement
 - (b) Structural Partitioning
 - (c) Patterns
 - (d) Information hiding
5. Explain software design notations used to represent software design?
6. Develop a software design for a railway reservation system. Include all the design elements discussed in the chapter. Following assumption should be noted.
 - (i) The system should support network
 - (ii) The system should use Oracle as back end.
 - (iii) The system should support Windows 98 and higher versions.

NOTES

4.15 FURTHER READING

1. Software Engineering: A Practitioner's Approach – *Roger Pressman*
2. An Integrated Approach to Software Engineering – *Pankaj Jalote*

UNIT 5 SOFTWARE TESTING

Structure

- 5.0 Introduction
- 5.1 Unit Objectives
- 5.2 Software Testing Basics
 - 5.2.1 Principles of Software Testing; 5.2.2 Testability; 5.2.3 Characteristics of Software Test
- 5.3 Test Plan
- 5.4 Test Case Design
- 5.5 Software Testing Strategies
 - 5.5.1 Unit Testing; 5.5.2 Integration Testing; 5.5.3 System Testing; 5.5.4 Validation Testing
- 5.6 Testing Techniques
 - 5.6.1 White Box Testing; 5.6.2 Black Box Testing;
 - 5.6.3 Difference between White Box and Black Box Testing; 5.6.4 Gray Box Testing
- 5.7 Object-oriented Testing
 - 5.7.1 Testing of Classes; 5.7.2 Developing Test Cases in Object-oriented Testing
 - 5.7.3 Object-oriented Testing Methods
- 5.8 Let us Summarize
- 5.9 Answers to 'Check Your Progress'
- 5.10 Questions and Exercises
- 5.11 Further Reading

NOTES

5.0 INTRODUCTION

Software testing is an essential part of software development process, which is used to identify the correctness, completeness and quality of developed software. It's main objective is to detect errors in the software. Errors prevent software from producing outputs according to user requirements. Errors occur when any aspect of a software product is incomplete, inconsistent, or incorrect. Errors can be broadly classified into three types, namely, requirements errors, design errors, and programming errors. To avoid these errors, it is necessary that: requirements are examined for conformance to user needs, software design is consistent with the requirements and notational convention, and the source code is examined for conformance to the requirements specification, design documentation, and user expectations. All this can be accomplished through efficacious means of software testing.

Software testing involves activities aimed at evaluating an attribute or capability of a program or system and ensuring that it meets its required results. It should be noted that testing is fruitful only if it is performed in a correct manner. Through effective software testing, the software can be examined for correctness, comprehensiveness, consistency, and adherence to standards. This helps in delivering high quality software products and lowering maintenance costs, thus leading to more contented users.

5.1 UNIT OBJECTIVES

After reading this unit, the reader will understand:

- Guidelines that are required to perform efficient and effective testing.
- Test plan, which describes objectives, scope, and method of software testing.

NOTES

- Testing strategies that are used to carry out testing in a planned and systematic manner.
- Various levels of testing: unit testing, integration testing, system testing, and acceptance testing.
- White box testing and black box testing techniques.
- How testing is performed in the object-oriented environment.
- Various tools used in software testing.
- The importance of software test report.

5.2 SOFTWARE TESTING BASICS

Software testing is a process, which is used to identify the correctness, completeness and quality of software. IEEE defines testing as “*the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.*”

Software testing is often used in association with the terms verification and validation.

Verification refers to checking or testing of items, including software, for conformance and consistency with an associated specification. For verification, techniques like reviews, analysis, inspections and walkthroughs are commonly used. While **validation** refers to the process of checking that the developed software is according the requirements specified by the user. Verification and validation can be summarised as follows:

Verification: Are we developing the **software right**?

Validation: Are we developing the **right software**?

(a) **Objectives of Software Testing:** Software testing evaluates software by manual and automated means to ensure that it is functioning in accordance with user requirements. The main objectives of software testing are listed below:

- To remove errors, which prevent software from producing outputs according to user requirements?
- To remove errors that lead to software failure.
- To determine whether system meets business and user needs.
- To ensure that software is developed according to user requirements.
- To improve the quality of software by removing maximum possible errors from it.

(b) **Testing in Software Development Life Cycle (SDLC):** Software testing comprises of a set of activities, which are planned before testing begins. These activities are carried out for detecting errors that occur during various phases of SDLC. The role of testing in software development life cycle is listed in Table 5.1.

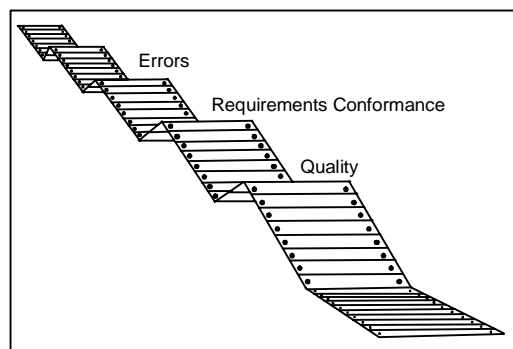


Figure 5.1 Objectives of Software Testing

Table 5.1 Role of Testing in Various Phases of SDLC

Software Development Phase	Testing
Requirements Phase	<ul style="list-style-type: none"> ▪ Determine the test strategy. ▪ Determine adequacy of requirements. ▪ Generate functional test conditions.
Design Phase	<ul style="list-style-type: none"> ▪ Determine consistency of design with requirements. ▪ Determine adequacy of design. ▪ Generate structural and functional test conditions.
Coding Phase	<ul style="list-style-type: none"> ▪ Determine consistency with design. ▪ Determine adequacy of implementation. ▪ Generate structural and functional test conditions for programs/units.
Testing Phase	<ul style="list-style-type: none"> ▪ Determine adequacy of the test plan. ▪ Test application system.
Installation and Maintenance Phase	<ul style="list-style-type: none"> ▪ Place tested system into production. ▪ Modify and retest.

NOTES

(c) Bugs, Error, Fault and Failure: The purpose of software testing is to find bugs, errors, faults, and failures present in the software. **Bug** is defined as a logical mistake, which is caused by a software developer while writing the software code. **Error** is defined as the difference between the outputs produced by the software and the output desired by the user (expected output). **Fault** is defined as the condition that leads to malfunctioning of the software. Malfunctioning of software is caused due to several reasons, such as change in the design, architecture, or software code. Defect that causes error in operation or negative impact is called failure. **Failure** is defined as the state in which software is unable to perform a function according to user requirements. Bugs, errors, faults, and failures prevent software from performing efficiently and hence, cause the software to produce unexpected outputs. Errors can be present in the software due to the reasons listed below:

- **Programming errors:** Programmers can make mistakes while developing the source code.
- **Unclear requirements:** The user is not clear about the desired requirements or the developers are unable to understand the user requirements in a clear and concise manner.
- **Software complexity:** The complexity of current software can be difficult to comprehend for someone who does not have prior experience in software development.
- **Changing requirements:** The user may not understand the effects of change. If there are minor changes or major changes, known and unknown dependencies among parts of the project are likely to interact and cause problems. This may lead to complexity of keeping track of changes and ultimately may result in errors.
- **Time pressures:** Maintaining schedule of software projects is difficult. When deadlines are not met, the attempt to speed up the work causes errors.
- **Poorly documented code:** It is difficult to maintain and modify code that is badly written or poorly documented. This causes errors to occur.

Note: In this chapter, 'error' is used as a general term for 'bugs', 'errors', 'faults', and 'failures'.

(d) Who Performs Testing?: Testing is an organisational issue, which is performed either by the software developers (who originally developed the software) or by an *independent test group* (ITG), which comprises of software testers. The software developers are

NOTES

considered to be the best persons to perform testing as they have the best knowledge about the software. However, since software developers are involved in the development process, they may have their own interest to show that the software is error free, meets user requirements, and is within schedule and budget. This vested interest hinders the process of testing.

To avoid this problem, the task of testing is assigned to an independent test group (ITG), which is responsible to detect errors that may have been neglected by the software developers. ITG tests the software without any discrimination since the group is not directly involved in the development process. However, the testing group does not completely take over the testing process, instead it works with the software developers in the software project to ensure that testing is performed in an efficient manner. During the testing process, developers are responsible for correcting the errors uncovered by the testing group.

Generally, an independent testing group forms a part of the software development project team. This is because the group becomes involved during the specification activity and stays involved (planning and specifying test procedures) throughout the development process.

- The various advantages and disadvantages associated with independent testing group are listed in Table 5.2.

Table 5.2 Advantages and Disadvantages of Independent Test Group

Advantages	Disadvantages
<ul style="list-style-type: none"> • Independent testing is typically more efficient at detecting defects related to special cases, interaction between modules, and system level usability and performance problems. • Programmers are neither trained, nor motivated to test. Thus ITG serves as an immediate solution. • Test groups can provide insight into the reliability of the software before it is delivered to the user. 	<ul style="list-style-type: none"> • Keeping independent test groups can result in duplication of effort. For example, the test group may use resources to perform tests that have already been performed by the developers. • Problem can arise when the test group is not physically collocated with the design group. • The cost of maintaining separate test groups is very high.

To plan and perform testing, software testers should have the knowledge about the function for which the software has been developed, the inputs and how they can be combined, and the environment in which the software will eventually operate. This process is time-consuming and requires technical sophistication and proper planning on the part of the testers. To achieve technical know-how, testers must not only have good development skills but also possess knowledge about formal languages, graph theory, and algorithms. Other factors that should be kept in mind while performing testing are:

- Time available to perform testing.
- Training required acquainting testers about the software.
- Attitude of testers.
- Relationship between testers and developers.

Note: Along with software testers, customers, end-users, and management also play an important role in software testing.

5.2.1 Principles of Software Testing

There are certain principles that are followed during software testing. These principles act as a standard to test software and make testing more effective and efficient. The commonly used software testing principles are listed below:

- **Define the expected output:** When programs are executed during testing, they may or may not produce the expected outputs due to different types of errors present in the software. To avoid this, it is necessary to define the expected output before software testing begins. Without knowledge of the expected results, testers may fail to detect an erroneous output.
- **Inspect output of each test completely:** Software testing should be performed once the software is complete in order to check its performance and functionality. Also, testing should be performed to find the errors that occur in various phases of software development.

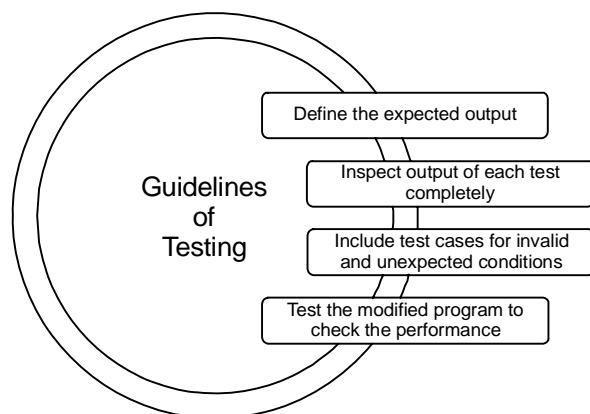


Figure 5.2 Software Testing Guidelines

- **Include test cases for invalid and unexpected conditions:** Generally, software produces correct outputs when it is tested using accurate inputs. However, if unexpected input is given to the software, it may produce erroneous outputs. Hence, test cases that detect errors even when unexpected and incorrect inputs are specified should be developed.
- **Test the modified program to check its expected performance:** Sometimes, when certain modifications are made in software (like adding of new functions) it is possible that software produces unexpected outputs. Hence, software should be tested to verify that it performs in the expected manner even after modifications.

5.2.2 Testability

The ease with which a program is tested is known as **testability**. Testability can be defined as the degree to which a program facilitates the establishment of test criteria and execution of tests to determine whether the criteria have been met or not. There are several characteristics of testability, which are listed below:

- **Easy to operate:** High quality software can be tested in a better manner. This is because if software is designed and implemented considering quality, then comparatively fewer errors will be detected during the execution of tests.
- **Observability:** Testers can easily identify whether the output generated for certain input is accurate or not simply by observing it.
- **Decomposability:** By breaking software into independent modules, problems can be easily isolated and the modules can be easily tested.
- **Stability:** Software becomes stable when changes made to the software are controlled and when the existing tests can still be performed.
- **Easy to understand:** Software that is easy to understand can be tested in an efficient manner. Software can be properly understood by gathering maximum information about it. For example, to have a proper knowledge of software, its documentation can be

NOTES

used, which provides complete information of software code thereby increasing its clarity and making testing easier. Note that documentation should be easily accessible, well organised, specific, and accurate.

NOTES

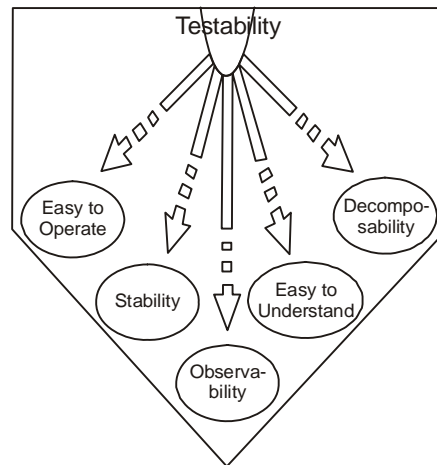


Figure 5.3 Testability

5.2.3 Characteristics of Software Test

There are several tests (such as unit and integration) used for testing software. Each test has its own characteristics. Following points should be noted:

- **High probability of finding errors:** To find maximum errors, the tester should understand the software thoroughly and try to find the possible ways in which the software can fail. For example, in a program to divide two numbers, the possible way in which the program can fail is when 2 and 0 are given as inputs and 2 is to be divided by 0. In this case, a set of tests should be developed that can demonstrate an error in the division operator.
- **No redundancy:** Resources and testing time are limited in software development process. Thus, it is not beneficial to develop several tests, which have the same intended purpose. That is, every test should have a distinct purpose.
- **Choose most appropriate test:** There can be different tests that have the same intent but due to certain limitations, such as time and resource constraint, only few of them are used. In such a case, the tests that have the highest probability of finding errors should be considered.
- **Moderate:** A test is considered good if it is neither too simple nor too complex. Many tests can be combined to form one test case. However, this can increase the complexity and leave many errors undetected. Hence, all the tests should be performed separately.

Check Your Progress

1. Define verification and validation. Differentiate between them.
2. Explain the terms, bugs, error, fault, and failure.
3. What function is performed by independent test group?

5.3 TEST PLAN

A test plan describes how testing would be accomplished. A test plan is defined as a document that describes the objectives, scope, method, and purpose of software testing. This plan identifies test items, features to be tested, testing tasks and the persons involved in performing these tasks. It also identifies the test environment and the test design and measurement techniques that are to be used. Note that a properly defined test plan is an agreement between testers and users describing the role of testing in software.

A complete test plan helps people outside the test group to understand the ‘why’ and ‘how’ of product validation. Whereas an incomplete test plan can result in a failure to check how the software works on different hardware and operating systems or when software is used with other software. To avoid this problem, IEEE states some components that should be covered in a test plan. These components are listed in Table 5.3.

Table 5.3 Test Plan Components

Component	Purpose
Responsibilities	Assigns responsibilities and keeps people on track and focused.
Assumptions	Avoids misunderstandings about schedules.
Test	Outlines the entire process and maps specific tests. The testing scope, schedule, and duration are also outlined.
Communication	Communication plan (who, what, when, how about the people) is developed.
Risk Analysis	Identifies areas that are critical for success.
Defect Reporting	Specifies how to document a defect so that it can be reproduced, fixed, and retested.
Environment	Specifies the technical environment, data, work area, and interfaces used in testing. This reduces or eliminates misunderstandings and sources of potential delay.

Steps in Development of Test Plan: A carefully developed test plan facilitates effective test execution, proper analysis of errors, and preparation of error report. To develop a test plan, a number of steps are followed, which are listed below:

- 1. Set objectives of test plan:** Before developing a test plan, it is necessary to understand its purpose. The objectives of a test plan depend on the objectives of software. For example, if the objective of software is to accomplish all user requirements, then a test plan is generated to meet this objective. Thus, it is necessary to determine the objective of software before identifying the objective of test plan.
- 2. Develop a test matrix:** Test matrix indicates the components of software that are to be tested. It also specifies the tests required to test these components. Test matrix is also used as a test proof to show that a test exists for all components of software that require testing. In addition, test matrix is used to indicate the testing method which is used to test the entire software.
- 3. Develop test administrative component:** It is necessary to prepare a test plan within a fixed time so that software testing can begin as soon as possible. The test administrative component of test plan specifies the time schedule and resources (administrative people involved while developing the test plan) required to execute the test plan. However, if implementation plan (a plan that describes how the processes in software are carried out) of software changes, the test plan also changes. In this case, the schedule to execute the test plan also gets affected.
- 4. Write the test plan:** The components of test plan, such as its objectives, test matrix, and administrative component are documented. All these documents are then collected together to form a complete test plan. These documents are organised either in an informal or formal manner. In informal manner, all the documents are collected and kept together. The testers read all the documents to extract information required for testing software. On the other hand, in formal manner, the important points are extracted

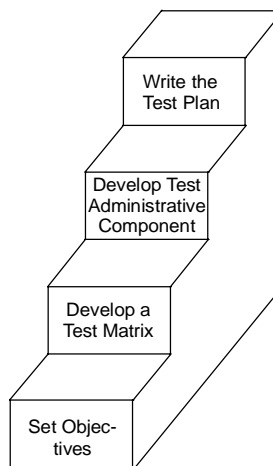


Figure 5.4 Steps in Test Plan

NOTES

from the documents and kept together. This makes it easy for testers to extract important information, which they require during software testing.

A test plan is shown in Figure 5.5. This plan has many sections, which are listed below:

NOTES

1.0	Overview
1.1	Project Objectives
1.2	System Description
1.3	Plan Objectives
1.4	References
1.5	Issues, Assumptions
2.0	Test Scope
2.1	Features to be tested
2.2	Features not to be tested
3.0	Test Methodologies
3.1	Testing Approach
3.2	Test Data
3.3	Test Documents
3.4	Requirements Validation
3.5	Control Procedures
4.0	Test Phases
4.1	Definition
4.2	Participants
4.3	Source of Data
4.4	Entrance and Exit Criteria
4.5	Requirements
4.6	Work Products
4.7	Test Completion Acceptance
5.0	Test Environment
5.1	Hardware
5.2	Software
5.3	Location
5.4	Staffing and Training
6.0	Schedule
7.0	Approvals and Distribution

Figure 5.5 Test Plan

- **Overview:** Describes the objectives and functions of the software to be performed. It also describes the objectives of test plan, such as defining responsibilities, identifying test environment and giving a complete detail of the sources from where the information is gathered to develop the test plan.
- **Test scope:** Specifies features and combination of features, which are to be tested. These features may include user manuals or system documents. It also specifies the features and their combinations that are not to be tested.
- **Test methodologies:** Specifies types of tests required for testing features and combination of these features, such as regression tests and stress tests. It also provides description of sources of test data along with how test data is useful to ensure that testing is adequate, such as selection of boundary or null values. In addition, it describes the procedure for identifying and recording test results.
- **Test phases:** Identifies various kinds of tests, such as unit testing, integration testing and provides a brief description of the process used to perform these tests. Moreover, it identifies the testers that are responsible for performing testing and provides a detailed description of the source and type of data to be used. It also describes the procedure of evaluating test results and describes the work products, which are initiated or completed in this phase.
- **Test environment:** Identifies the hardware, software, automated testing tools, operating system, compilers, and sites required to perform testing. It also identifies the staffing and training needs.
- **Schedule:** Provides detailed schedule of testing activities and defines the responsibilities to respective people. In addition, it indicates dependencies of testing activities and the time frames for them.

Check Your Progress

4. What is a test plan?
5. Briefly describe components of a test plan.

- **Approvals and distribution:** Identifies the individuals who approve a test plan and its results. It also identifies the people to whom test plan document(s) is distributed.

5.4 TEST CASE DESIGN

A test case is a document that describes an input, action, or event and its expected result, in order to determine whether the software or a part of the software is working correctly or not. IEEE defines test case as “*a set of input values, execution preconditions, expected results and execution post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement*”. Generally, a test case contains particulars, such as test case identifier, test case name, its objective, test conditions/setup, input data requirements, steps, and expected results.

Incomplete and incorrect test cases lead to incorrect and erroneous test outputs. To avoid this, a test case should be developed in such a manner that it checks software with all possible inputs. This process is known as **exhaustive testing** and the test case, which is able to perform exhaustive testing, is known as **ideal test case**. Generally, a test case is unable to perform exhaustive testing therefore, a test case that gives satisfactory results is selected. In order to select a test case, certain questions should be addressed.

- How to select a test case?
- On what basis certain elements of program are included or excluded from a test case?

To provide an answer to the above-mentioned questions, a test selection criterion is used. For a given program and its specifications, a test selection criterion specifies the conditions that should be satisfied by a set of test cases. For example, if the criterion is that all the control statements in a program are executed at least once during testing, then a set of test cases which ensures that the specified condition is met, should be selected.

(a) Test Case Generation: The process of generating test cases helps in locating problems in the requirements or design of software. To generate a test case, initially a criterion that evaluates a set of test cases is specified. Then, a set of test cases that satisfy the specified criterion is generated. There are two methods used to generate test cases, which are listed below:

- **Code based test case generation:** This approach, also known as structure based test case generation is used to analyse the entire software code to generate test cases. It considers only the actual software code to generate test cases and is not concerned with the user requirements. Test cases developed using this approach are generally used for unit testing. These test cases can easily test statements, branches, special values, and symbols present in the unit being tested.
- **Specification based test case generation:** This approach uses specifications, which indicate the functions that are produced by software to generate test cases. In other words, it considers only the external view of software to generate test cases. Specification based test case generation is generally used for integration testing and system testing to ensure that software is performing the required task. Since this approach considers only the external view of the software, it does not test the design decisions and may not cover all statements of a program. Moreover, as test cases are derived from specifications, the errors present in these specifications may remain uncovered.

Several tools known as **test case generators** are used for generating test cases. In addition to test case generation, these tools specify the components of software that are to be tested. An example of test case generator is ‘astra quick test’, which captures business processes in the visual map and generates data driven tests automatically.

NOTES

NOTES

(b) Test Case Specifications: The test plan is not concerned with the details of testing a unit. Moreover, it does not specify the test cases to be used for testing units. Thus, test case specification is done in order to test each unit separately. Depending on the testing method specified in test plan, features of unit that need to be tested are ascertained. The overall approach stated in test plan is refined into specific test methods and into the criteria to be used for evaluation. Based on test methods and criteria, test cases to test the unit are specified.

For each unit being tested, these test case specifications provide test cases, inputs to be used in test cases, conditions to be tested by tests cases and outputs expected from test cases. Generally, test cases are specified before they are used for testing. This is because, testing has many limitations and effectiveness of testing is highly dependent on the nature of test cases.

Test case specifications are written in the form of a document. This is because the quality of test cases needs to be evaluated. To evaluate the quality of test cases, test case review is done for which a formal document is needed. The review of test case document ensures that test cases satisfy the chosen criteria and are consistent with the policy specified in the test plan. The other benefit of specifying test cases formally is that it helps testers to select a good set of test cases.

5.5 SOFTWARE TESTING STRATEGIES

Software testing strategies can be considered as various levels of testing that are performed to test the software. The first level starts with testing of individual units of software. Once the individual units are tested, they are integrated and checked for interfaces established between them. After this, entire software is tested to ensure that the output produced is according to user requirements. As shown in Figure 5.6, there are four levels of software testing, namely, unit testing, integration testing, system testing, and acceptance testing.

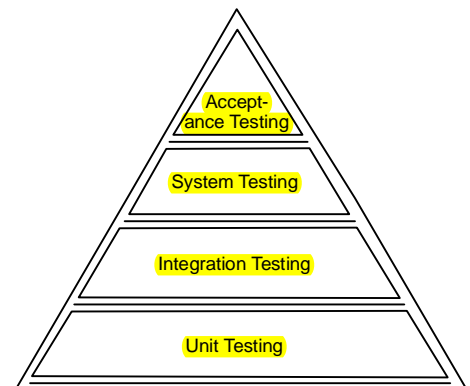


Figure 5.6 Levels of Software Testing

5.5.1 Unit Testing

Unit testing is performed to test the individual units of software. Since software is made of a number of units/modules, detecting errors in these units is simple and consumes less time, as they are small in size. However, it is possible that the outputs produced by one unit become input for another unit. Hence, if incorrect output produced by one unit is provided as input to the second unit, then it also produces wrong output. If this process is not corrected, the entire software may produce unexpected outputs. To avoid this, all the units in software are tested independently using unit testing.

Unit level testing is not just performed once during the software development, rather it is repeated whenever software is modified or used in a new environment. The other points noted about unit testing are listed below:

- Each unit is tested in isolation from other parts of a program.
- The developers themselves perform unit testing.
- Unit testing makes use of white box testing methods.

Check Your Progress

6. Define test case.
7. What is exhaustive testing and ideal test case?
8. Define the role played by test case specification.

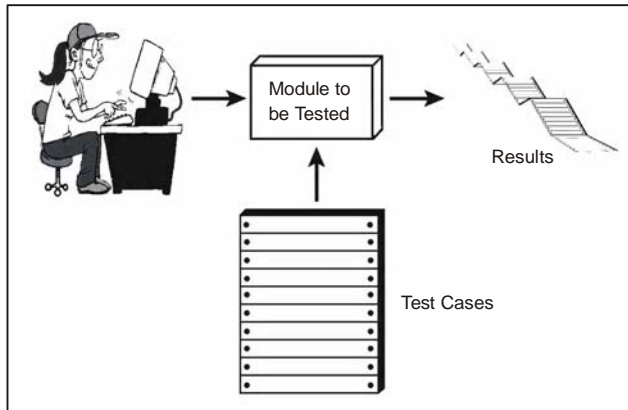


Figure 5.7 Unit Testing

Unit testing is used to verify the code produced during software coding and is responsible for assessing the correctness of a particular unit of source code. In addition, unit testing performs the functions listed below:

- Tests all control paths to uncover maximum errors that occur during the execution of conditions present in the unit being tested.
- Ensures that all statements in the unit are executed at least once.
- Tests data structures (like stacks, queues) that represent relationships among individual data elements.
- Checks the range of inputs given to units. This is because every input range has a maximum and minimum value and the input given should be within the range of these values.
- Ensures that the data entered in variables is of the same data type as defined in the unit.
- Checks all arithmetic calculations present in the unit with all possible combinations of input values.

(a) Types of Unit Testing: A series of stand-alone tests are conducted during unit testing. Each test examines an individual component that is new or has been modified. A unit test is also called a module test because it tests the individual units of code that form part of the program and eventually the software. In a conventional structured programming language, such as C, the basic unit is a *function* or *sub-routine* while, in object-oriented language such as C++ the basic unit is a *class*.

The various tests that are performed as a part of unit testing are listed below:

- **Module interface:** These are tested to ensure that information flows in a proper manner into and out of the 'unit' under test. Note that test of data flow (across a module interface) is required before any other test is initiated.
- **Local data structure:** These are tested to ensure that the temporarily stored data maintains its integrity while an algorithm is being executed.

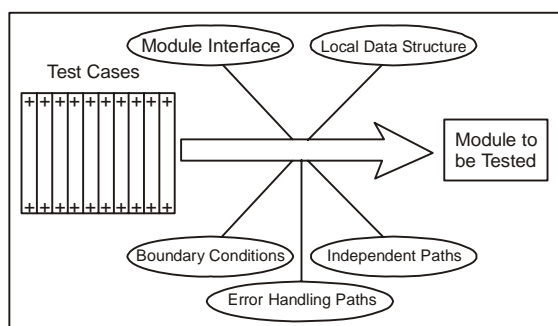


Figure 5.8 Various Unit Testing Methods

NOTES

NOTES

- **Boundary conditions:** These are tested to ensure that the module operates as desired within the specified boundaries.
- **All independent paths:** These are tested to ensure that all statements in a module have been executed at least once. Note that in this testing, the entire control structure should be exercised.
- **Error handling paths:** After successful completion of the various tests, error-handling paths are tested.

(b) Unit Test Case Generation: Various unit test cases are generated to perform unit testing. Test cases are designed to uncover errors that occur due to erroneous computations, incorrect comparisons, and improper control flow. A proper unit test case ensures that unit testing is performed efficiently. To develop test cases, the following points should be considered.

- **Expected functionality:** A test case is created for testing all functionalities present in the unit being tested. For example, structured query language (SQL) query is given that creates Table_A and alters Table_B. A test case is developed to make sure that 'Table_A' is created and 'Table_B' is altered.
- **Input values:** Test cases are developed to check various aspects of inputs, which are listed below:
 - *Every input value:* A test case is developed to check every input value, which is accepted by the unit being tested. For example, if a program is developed to print a table of five, then a test case is developed which verifies that only five is entered as input.
 - *Validation of input:* Before executing software, it is important to verify whether all inputs are valid or not. For this purpose, a test case is developed which verifies the validation of all inputs. For example, if a numeric field accepts only positive values, then a test case is developed to verify that the numeric field is accepting only positive values.
 - *Boundary conditions:* Generally, software fails at the boundaries of input domain (maximum and minimum value of the input domain). Thus, a test case is developed, which is capable of detecting errors that caused software to fail at the boundaries of input domain. For example, errors may occur while processing the last element of an array. In this case, a test case is developed to check whether error occurs while processing the last element of the array or not.
 - *Limitation of data types:* Variable that holds data types has certain limitations. For example, if a variable with data type 'long' is executed then a test case is developed to ensure that the input entered for the variable is within the acceptable limit of 'long' data type.
- **Output values:** A test case is developed to check whether the unit is producing the expected output or not. For example, when two numbers, '2' and '3' are entered as input in a program that multiplies two numbers, then a test case is developed to verify that the program produces the correct output value, that is, '6'.
- **Path coverage:** There can be many conditions specified in a unit. For executing all these conditions, many paths have to be traversed. For example, when a unit consists of nested 'if' statements and all of them are to be executed, then a test case is developed to check whether all these paths are traversed or not.
- **Assumptions:** For a unit to execute properly, certain assumptions are made. Test cases are developed by considering these assumptions. For example, a unit may need a database to be open. Then a test case is written to check that the unit reports errors, if such assumptions are not met.

- **Abnormal terminations:** A test case is developed to check the behaviour of a unit in case of abnormal termination. For example, when a power cut results in termination of a program due to shutting down of the computer, a test case is developed to check the behaviour of a unit as a result of abnormal termination of program.
- **Error messages:** Error messages that appear when software is executed should be short, precise, self explanatory, and free from grammatical mistakes. For example, if 'print' command is given when a printer is not installed, error message that appears should be 'Printer not installed' instead of 'Problem has occurred as no printer is installed and hence unable to print'. In this case, a test case is developed to check whether the error message is according to the condition occurring in the software or not.

NOTES

(c) **Unit Testing Procedure:** Unit tests can be designed before coding begins or after the code is developed. Review of this design information guides the creation of test cases, which are used to detect errors in various units. Since a component is not an independent program, two modules, drivers and stubs are used to test the units independently. **Driver** is a module that passes input to the unit to be tested. It accepts test case data and then passes the data to the unit being tested. After this, driver prints the output produced. **Stub** is a module that works as unit referenced by the unit being tested. It uses the interface of the subordinate unit, does minimum data manipulation, and returns control back to the unit being tested.

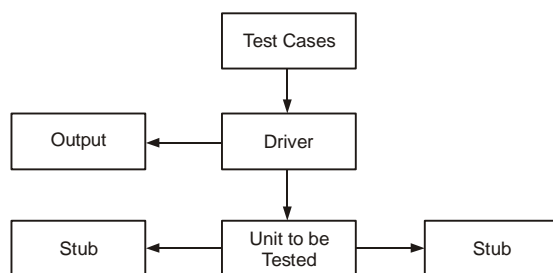


Figure 5.9 Unit Testing Environment

Note: Drivers and stubs are not delivered with the final software product. Thus, they represent an overhead.

5.5.2 Integration Testing

Once unit testing is complete, integration testing begins. In integration testing, the units validated during unit testing are combined to form a sub system. The purpose of integration testing is to ensure that all the modules continue to work in accordance with user/customer requirements even after integration.

The objective of integration testing is to take all the tested individual modules, integrate them, test them again, and develop the software, which is according to design specifications. The other points that are noted about integration testing are listed below:

- Integration testing ensures that all modules work together properly, are called *correctly*, and transfer accurate data across their interfaces.
- Testing is performed with an intention to expose defects in the interfaces and in the interactions between integrated components or systems.
- Integration testing examines the components that are new, changed, affected by a change, or needed to form a complete system.

NOTES

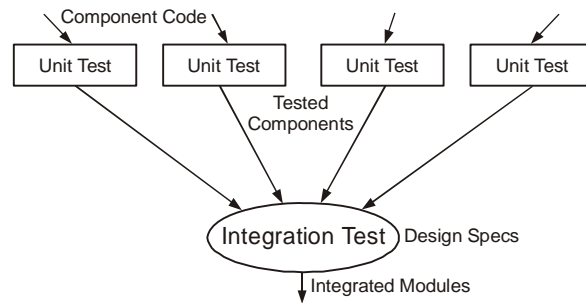


Figure 5.10 Integration of Individual Modules

The *big bang* approach and *incremental integration* approach are used to integrate modules of a program. In *big bang* approach, initially, all modules are integrated and then the entire program is tested. However, when the entire program is tested, it is possible that a set of errors is detected. It is difficult to correct these errors since it is difficult to isolate the exact cause of the errors when program is very large. In addition, when one set of errors is corrected, new sets of errors arise and this process continues indefinitely.

To overcome the above problem, incremental integration is followed. This approach tests program in small increments. It is easier to detect errors in this approach because only a small segment of software code is tested at a given instance of time. Moreover, interfaces can be tested completely if this approach is used. Various kinds of approaches are used for performing incremental integration testing, namely, top-down integration testing, bottom-up integration testing, regression testing, and smoke testing.

(a) Top-down Integration Testing: In this testing, software is developed and tested by integrating the individual modules, moving downwards in the control hierarchy. In top-down integration testing, initially only one module known as the main control module is tested. After this, all the modules called by it are combined with it and tested. This process continues till all the modules in the software are integrated and tested.

It is also possible that a module being tested calls some of its subordinate modules. To simulate the activity of these subordinate modules, a stub is written. Stub replaces modules that are subordinate to the module being tested. Once, the control is passed to the stub, it does minimal data manipulation, provides verification of entry, and returns control back to the module being tested.

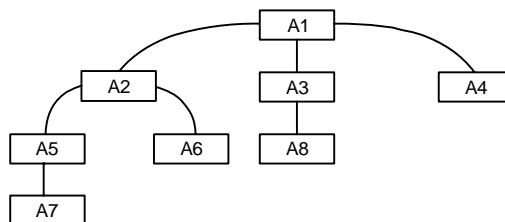


Figure 5.11 Top-down Integration

To perform top-down integration testing, a number of steps are followed, which are listed below:

1. The main control module is used as a test driver and stubs are used to replace all the other modules, which are directly subordinate to the main control module.
2. Subordinate stubs are then replaced one at a time with actual modules. The manner in which the stubs are replaced depends on the approach (depth first or breadth first) used for integration.

3. Every time a new module is integrated, tests are conducted.
4. After tests are complete, another stub is replaced with the actual module.
5. Regression testing is conducted to ensure that no new errors are introduced.

Top-down integration testing uses either depth-first integration or breadth-first integration for integrating the modules. In depth-first integration, the modules are integrated starting from left and then moves down in the control hierarchy. As shown in Figure 5.12(a), initially, modules 'A1', 'A2', 'A5' and 'A7' are integrated. Then, module 'A6' integrates with module 'A2'. After this, control moves to the modules present at the centre of control hierarchy, that is, module 'A3' integrates with module 'A1' and then module 'A8' integrates with module 'A3'. Finally, the control moves towards right, integrating module 'A4' with module 'A1'.

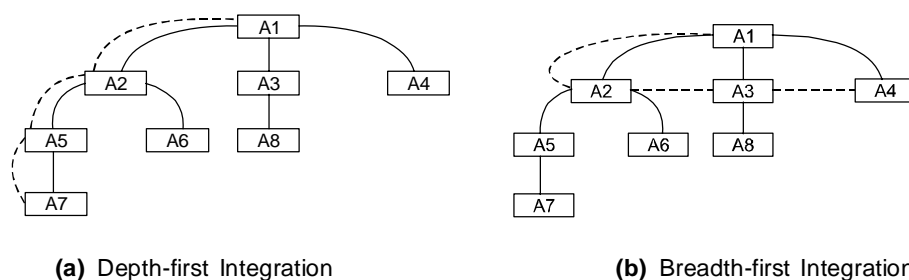


Figure 5.12 Top-down Integration

In breadth-first integration, initially, all modules at the first level are integrated moving downwards, integrating all modules at the next lower levels. As shown in Figure 5.12 (b), initially, modules 'A2', 'A3', and 'A4' are integrated with module 'A1' and then it moves down integrating modules 'A5' and 'A6' with module 'A2' and module 'A8' with module 'A3'. Finally, module 'A7' is integrated with module 'A5'.

The various advantages and disadvantages associated with top-down integration are listed in Table 5.4.

Table 5.4 Advantages and Disadvantages of Top-down Integration

Advantages	Disadvantages
<ul style="list-style-type: none"> Behaviour of modules at high level is verified early. None or only one driver is required. Modules can be added one at a time with each step. Supports both breadth-first method and depth-first method. Modules are tested in isolation from other modules. 	<ul style="list-style-type: none"> Delays the verification of behaviour of modules present at lower levels. Large numbers of stubs are required in case the lowest level of software contains many functions. Since stubs replace modules present at lower levels in the control hierarchy, no data flows upward in program structure. To avoid this, tester has to delay many tests until stubs are replaced with actual modules or has to integrate software from the bottom of the control hierarchy moving upward. Module cannot be tested in isolation from other modules because it has to invoke other modules.

(b) Bottom-up Integration Testing: In this testing, individual modules are integrated starting from the bottom and then moving upwards in the hierarchy. That is, bottom-up integration testing combines and tests the modules present at the lower levels proceeding towards the modules present at higher levels of control hierarchy.

NOTES

NOTES

Some of the low-level modules present in software are integrated to form clusters or builds (collection of modules). After clusters are formed, a driver is developed to co-ordinate test case input and output and then, the clusters are tested. After this, drivers are removed and clusters are combined moving upwards in the control hierarchy.

Figure 5.13 shows modules, drivers, and clusters in bottom-up integration. The low-level modules 'A4', 'A5', 'A6', and 'A7' are combined to form cluster 'C1'. Similarly, modules 'A8', 'A9', 'A10', 'A11', and 'A12' are combined to form cluster 'C2'. Finally, modules 'A13' and 'A14' are combined to form cluster 'C3'. After clusters are formed, drivers are developed to test these clusters. Drivers 'D1', 'D2', and 'D3' test clusters 'C1', 'C2', and 'C3' respectively. Once these clusters are tested, drivers are removed and clusters are integrated with the modules. Cluster 'C1' and cluster 'C2' are integrated with module 'A2'. Similarly, cluster 'C3' is integrated with module 'A3'. Finally, both the modules 'A2' and 'A3' are integrated with module 'A1'.

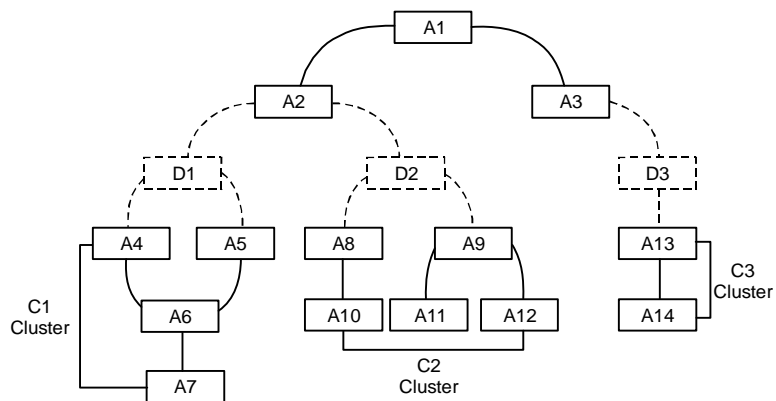


Figure 5.13 Bottom-up Integration

The various advantages and disadvantages associated with bottom-up integration are listed in Table 5.5.

Table 5.5 Advantages and Disadvantages of Bottom-up Integration

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Behaviour of modules at lower levels is verified earlier. ▪ No stubs are required. ▪ Uncovers errors that occur at the lower levels in software. ▪ Modules are tested in isolation from other modules. 	<ul style="list-style-type: none"> ▪ Delays in verification of modules at higher levels. ▪ Large numbers of drivers are required to test clusters.

(c) Regression Testing: Software undergoes changes every time a new module is added as part of integration testing. Changes can occur in the control logic or input/output media, and so on. It is possible that new data flow paths are established as a result of these changes, which may cause problems in the functioning of some parts of the software that was previously working perfectly. In addition, it is also possible that new errors may surface during the process of correcting existing errors. To avoid these problems, regression testing is used.

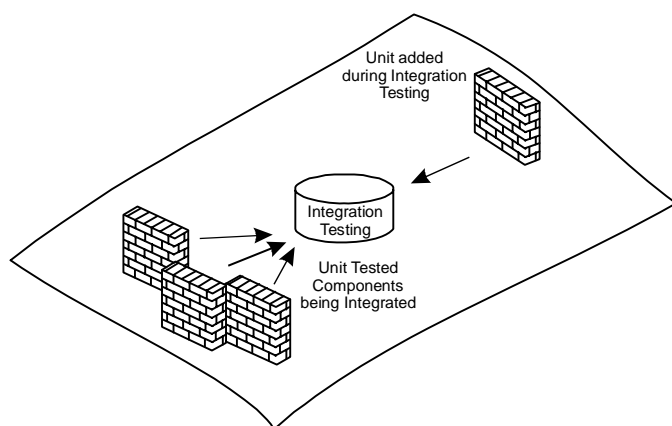


Figure 5.14 Addition of Module in Integration Testing

Regression testing ‘re-tests’ the software or part of it to ensure that no previously working components, functions, or features fail as a result of the error correction process and integration of modules. Regression testing is considered an expensive but a necessary activity since it is performed on modified software to provide knowledge that changes do not adversely affect other system components. Thus, regression testing can be viewed as a quality control tool that ensures that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the change. For instance, suppose a new function is added to the software, or a module is modified to improve its response time. The changes may introduce errors into the software that was previously correct. For example, suppose part of the code written below works properly.

```
x = b + 1 ;
proc (z) ;
b = x + 2 ; x = 3;
```

Now suppose in an attempt to optimise the code it is transformed into the following:

```
proc (z) ;
b = b + 3 ;
x = 3 ;
```

This may result in an error if procedure ‘proc’ accesses variable ‘x’. Thus, testing should be organised with the purpose of verifying possible degradations of correctness or other qualities due to later modifications. During regression testing, existing test cases are executed on the modified software so that errors can be detected. Test cases for regression testing consist of three different types of tests, which are listed below:

- Tests that are used to execute software function.
- Tests that check the function, which is likely to be affected by changes.
- Tests that check software modules that have already been modified.

The various advantages and disadvantages associated with regression testing are listed in Table 5.6.

NOTES

Table 5.6 Advantages and Disadvantages of Regression Testing**NOTES**

Advantages	Disadvantages
<ul style="list-style-type: none"> Ensures that the unchanged parts of software work properly. Ensures that all errors that have occurred in software due to modifications are corrected and are not affecting the working of software. 	<ul style="list-style-type: none"> Time consuming activity. Considered to be expensive.

(d) Smoke Testing Smoke testing is defined as a subset of all defined test cases that cover the main functionality of a component or system, to ascertain that the most crucial functions of a program work properly. It is mainly used for time critical software and allows the development team to assess the software frequently.

Smoke testing is performed when software is under development. As the modules of software are developed, they are integrated to form a 'cluster'. After the cluster is formed, certain tests are designed to detect errors that prevent the cluster to perform its function. Next, the cluster is integrated with other clusters thereby leading to the development of the entire software, which is smoke tested frequently. A smoke test should possess the following characteristics:

- Should run quickly.
- Should try to cover a large part of software and if possible the entire software.
- Should be easy for testers to perform smoke testing on software.
- Should be able to detect all errors present in the cluster being tested.
- Should try to find showstopper errors.

Generally, smoke testing is conducted every time a new cluster is developed and integrated with the existing cluster. Smoke testing takes minimum time to detect errors that occur due to integration of clusters. This reduces the risk associated with the occurrence of problems, such as introduction of new errors in software. A cluster cannot be sent for further testing unless smoke testing is performed on it. Thus, smoke testing determines whether the cluster is suitable to be sent for further testing or not. Other benefits associated with smoke testing are listed below:

- **Minimises the risks, which are caused due to integration of different modules:** Since smoke testing is performed frequently on software, it allows the testers to uncover errors as early as possible, thereby reducing the chance of causing severe impact on the schedule when there is delay in uncovering errors.
- **Improves quality of final software:** Since smoke testing detects both functional and architectural errors as early as possible, they are corrected early, thereby resulting in high quality software.
- **Simplifies detection and correction of errors:** As smoke testing is performed almost every time a new code is added, it becomes clear that the probable cause of errors is the new code.
- **Assesses progress easily:** Since smoke testing is performed frequently, it keeps track of the continuous integration of modules, that is, the progress of software development. This boosts the morale of software developers.

Integration Test Documentation: To understand the overall procedure of software integration, a document known as test specification is prepared. This document provides information in the form of test plan, a test procedure, and actual test results.

1.0	Scope of Testing
2.0	Test Plan
2.1	Test Phases and Builds
2.2	Schedule
2.3	Overhead Software
2.4	Environment and Resources
3.0	Test Procedure 'n'
3.1	Order of Integration
3.1.1	Purpose
3.1.2	Modules to be Tested
3.2	Unit Tests for Modules in Build
3.3	Description of Test for Module 'm'
3.4	Overhead Software Description
3.5	Expected Results
3.6	Test Environment
3.7	Special Tools or Techniques
3.8	Test Case Data
3.9	Expected Results for Build 'n'
4.0	Actual Test Results
5.0	References
6.0	Appendices

Figure 5.15 Integration Test Specification

Figure 5.15 shows integration test documentation. This template comprises of various sections, which are listed below:

- **Scope of testing:** Provides overview of the specific functional, performance, and design characteristics that are to be tested. In addition, scope describes the completion criteria for each test phase and keeps record of the constraints that occur in the schedule.
- **Test plan:** Describes the strategy for integration of software. Testing is divided into phases and builds. Phases describe distinct tasks that involve various sub-tasks. On the other hand, builds are group of modules that correspond to each phase. Both phases and builds address specific functional and behavioural characteristics of the software. Some of the common test phases that require integration testing include user interaction, data manipulation and analysis, display outputs, database management, and so on. Every test phase consists of a functional category within the software. Generally, these phases can be related to a specific domain within the architecture of software. The criteria commonly considered for all test phases include *interface integrity*, *functional validity*, *information content*, and *performance*.

Note that a test plan should be customised to local requirements, however it should contain an integration strategy (in the Test Plan) and testing details (in Test Procedure). Test plan should also include the following:

- A schedule for integration, which should include the start and end dates given for each phase.
- A description of overhead software, concentrating on those that may require special effort.
- A description of the testing environment.
- **Test procedure 'n':** Describes the order of integration and unit tests for modules. Order of integration provides information about the purpose and the modules to be tested. Unit tests are conducted for the modules that are built along with the description of tests for these modules. In addition, test procedure describes the development of overhead software, expected results during integration testing, and description of test case data. The test environment and tools or techniques used for testing are also mentioned in test procedure.

NOTES

NOTES

- **Actual test results:** Provides information about actual test results and problems that are recorded in the test report. With the help of this information, it is easy to carry out software maintenance.
- **References:** Describes the list of references that are used for preparing user documentation. Generally, references include books and websites.
- **Appendices:** Provides information about integration test document. Appendices are in the form of supplementary material that is provided at the end of the document.

5.5.3 System Testing

Software is integrated with other elements, such as hardware, people, and database to form a computer-based system. This system is then checked for errors using system testing. IEEE defines system testing as “a testing conducted on a complete, integrated system to evaluate the system’s compliance with its specified requirements”.

System testing compares the system with the non-functional system requirements, such as *security, speed, accuracy, and reliability*. The emphasis is on validating and verifying the functional design specifications and examining how modules work together. This testing also evaluates external interfaces to other applications and utilities or the operating environment.

During system testing, associations between objects (like fields), control and infrastructure (like time management, error handling), feature interactions or problems that occur when multiple features are used simultaneously and compatibility between previously working software releases and new releases are tested. System testing also tests some properties of the developed software, which are essential for users. These properties are listed below:

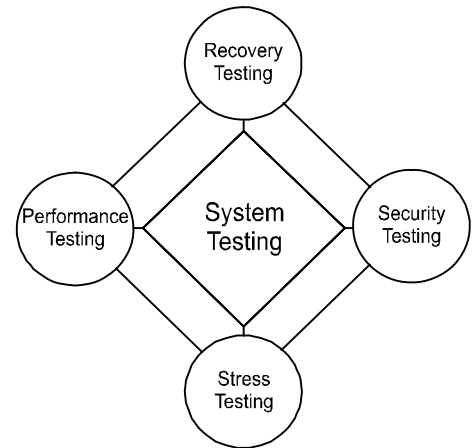


Figure 5.16 Types of System Testing

- **Usable:** Verifies that developed software is easy to use and is understandable.
- **Secure:** Verifies that access to important or sensitive data is restricted even for those individuals who have authority to use software.
- **Compatible:** Verifies that developed software works correctly in conjunction with existing data, software and procedures.
- **Documented:** Verifies that manuals that give information about developed software are complete, accurate and understandable.
- **Recoverable:** Verifies that there are adequate methods for recovery in case of failure.

System testing requires many test runs because it entails feature-by-feature validation of behaviour using a wide range of both normal and erroneous test inputs and data. Test plan plays an important role in system testing because it contains descriptions of the test cases, the sequence in which the tests must be executed, and the documentation needed to be collected in each run. When an error or defect is discovered, previously executed system tests must be rerun after the repair is made to make sure that the modifications do not lead to other problems.

As part of system testing, conformance tests and reviews can be run to verify that the application conforms to corporate or industry standards in terms of portability, interoperability, and compliance. For example, to enhance software portability, a corporate standard may be that SQL queries must be written so that they work for both Oracle and DB2 databases.

System testing is deemed to be complete when the actual results and expected results are either in line or in difference with the inputs specified by the user. Various kinds of testing performed as a part of system testing are recovery testing, security testing, stress testing and performance testing.

(a) Recovery Testing: Recovery testing is a system test, which forces the system to fail in different ways and verifies that the software recovers from expected or unexpected events without loss of data or functionality. Events, which lead to failure, include system crashes, hardware failures, unexpected loss of communication, and other catastrophic problems.

To recover from any type of failure, system should be fault tolerant. Fault tolerant system can be defined as a system which continues to perform the intended functions even when errors are present in it. In case the system is not fault tolerant, it needs to be corrected within a specified time limit after failure has occurred so that the software performs its functions in a desired manner.

Test cases generated for recovery testing not only show the presence of errors in system, but also provide information about the data lost due to problems, such as power failure and improper shutting down of computer system. Recovery testing also ensures that appropriate methods are used to restore the lost data. Other advantages of recovery testing are listed below:

- Checks whether the backup data is saved properly or not.
- Ensures that the backup data is stored in a secure location.
- Ensures that proper detail of recovery procedures is maintained.

(b) Security Testing: Systems with sensitive information are generally the target of improper or illegal use. Therefore, protection mechanisms are required to restrict unauthorised access to the system. To avoid any improper usage, security testing is performed which identifies and removes software flaws that may potentially lead to security violations. In security testing, the tester plays the role of the individual trying to penetrate the system. For this, tester performs tasks, such as cracking the password, attacking the system with custom software, which is used to break-down any protection mechanisms built to protect the system, and intentionally produces errors in the system. This testing focuses on the two areas of security listed below:

- **Application security:** Verifies that user can access only those data and functions for which system developer or user of system has given permission. This security is referred to as *authorisation*.
- **System security:** Verifies that only the users, who have permission to access the system, are accessing it. This security is referred to as *authentication*.

Unauthorised access can be made by 'outside' individuals for fun or personal gain or by disgruntled/dishonest employees. And if people are able to gain access to the system, then, there is a possibility that a large amount of important data can be lost resulting in huge loss to the organisation or individuals.

Security testing verifies that system accomplishes all the security requirements and validates the effectiveness of these security measures. Other advantages associated with security testing are listed below:

- Determines whether proper techniques are used to identify security risks or not.
- Verifies that appropriate protection techniques are followed to secure the system.
- Ensures that the system is able to protect its data and maintain its functionality.
- Conducts tests to ensure that the implemented security measures are working properly.

NOTES

NOTES

(c) Stress Testing: Stress testing is designed to test the software with abnormal situations. These abnormal situations arise when resources are required in abnormal quantity, frequency, or volume. For example, the abnormal conditions may arise when:

- There are higher rates of interrupts when the average rate is low.
- Test cases are developed, which cause ‘thrashing’ in a virtual operating system.
- There are test cases that cause excessive ‘hunting’ for data on disk systems.

IEEE defines stress testing as “*testing conducted to evaluate a system or component at or beyond the limits of its specified requirements*”. For example, if a software system is developed to execute 100 statements at a time, then stress testing may generate 110 statements to be executed. This load may increase until the software fails. Thus, stress testing specifies the way in which a system reacts when it is made to operate beyond its performance and capacity limits. The other advantages associated with stress testing are listed below:

- Indicates the expected behaviour of a system when it reaches the extreme level of its capacity.
- Executes a system till it fails. This enables the testers to determine the difference between the expected operating conditions and the failure conditions.
- Determines the part of a system that leads to errors.
- Determines the amount of load that causes a system to fail.
- Evaluates a system at or beyond the limits of its performance capacity.

(d) Performance Testing: Performance testing checks the run-time performance of the software (especially real-time and embedded systems) in the context of the entire computer based system. This testing is used to verify the load, volume, and response times as defined by requirements. Performance testing also determines and informs the software developer about the current performance of the software under various parameters (like condition to complete software within a specified time limit).

Often performance tests and stress tests are used together and require both software and hardware instrumentation of the system. By instrumenting a system, the tester can reveal situations that lead to conditions of system degradation and system failure. While performance tests evaluate response time, memory usage, throughput, device utilisation, and execution time, stress testing pushes the system to or beyond its specified limits to evaluate its robustness and error handling capabilities. Performance testing is used to test several factors that play an important role in improving the overall performance of the system. Some of these factors are listed below:

- **Speed:** Refers to the capability of a system to respond to users as quickly as possible. Performance testing verifies whether the response is quick enough or not.
- **Scalability:** Refers to the capability of a system to handle the load given to it. Performance testing verifies whether the system is able to handle the load expected by users or not.
- **Stability:** Refers to the capability of a system to prevent itself from failure as long as possible. Performance testing verifies whether the system remains stable under expected and unexpected loads.

The outputs produced during performance testing are provided to the system developer. Based on these outputs, the developer makes changes to the system in order to remove the errors. This testing also checks the system characteristics such as its reliability. Other advantages associated with performance testing are listed below:

- Evaluates the compliance of a system or component with specified performance requirements.
- Compares different systems to determine which system performs better.

5.5.4 Validation Testing

Validation testing, also known as **acceptance testing** is performed to determine whether software meets all the functional, behavioural, and performance requirements or not. IEEE defines acceptance testing as a “*formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorised entity to determine whether or not to accept the system*”.

During validation testing, software is tested and evaluated by a group of users either at the developer’s site or user’s site. This enables the users to test the software themselves and analyse whether it is meeting their requirements or not. To perform validation testing, a predetermined set of data is given to software as input. It is important to know the expected output before performing validation testing so that outputs produced by software as a result of testing can be compared with them. Based on the results of tests, users decide whether to accept or reject the software. That is, if both outputs (expected and produced) match, then software is considered to be correct and is accepted, otherwise, it is rejected.

The various advantages and disadvantages associated with validation testing are listed in Table 5.7.

Table 5.7 Advantages and Disadvantages of Acceptance Testing

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Gives user an opportunity to ensure that software meets user requirements, before actually accepting it from the developer. ▪ Enables both users and software developers to identify and resolve problems in software. ▪ Determines the readiness (state of being ready to operate) of software to perform operations. ▪ Decreases the possibility of software failure to a large extent. 	<ul style="list-style-type: none"> ▪ Although, users provide a valuable feedback, they do not have a detailed knowledge of software code. ▪ Since testing is not users’ primary occupation so they may fail to observe or accurately report some software failures.

Since the software is intended for large number of users, it is not possible to perform acceptance testing with all the users. Therefore, organisations engaged in software development use *alpha* and *beta* testing as a process to detect errors by allowing a limited number of users to test the software.

(a) Alpha Testing: Alpha testing is conducted by the users at the developer’s site. In other words, this testing assesses the performance of software in the environment in which it is developed. On completion of alpha testing, users report the errors to software developers so that they can correct them. Note that alpha testing is often employed as a form of internal acceptance testing.

NOTES

NOTES

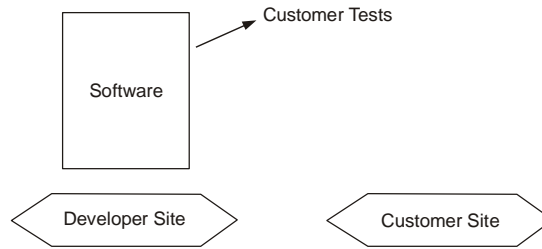


Figure 5.17 Alpha Testing

The advantages of alpha testing are listed below:

- Identifies all the errors present in the software.
- Checks whether all the functions mentioned in the requirements are implemented properly in software or not.

(b) Beta Testing: Beta testing assesses performance of software at *user's* site. This testing is 'live' testing and is conducted in an environment, which is not controlled by the developer. That is, this testing is performed without any interference from the developer. Beta testing is performed to know whether the developed software satisfies the user requirements and fits within the business processes or not.

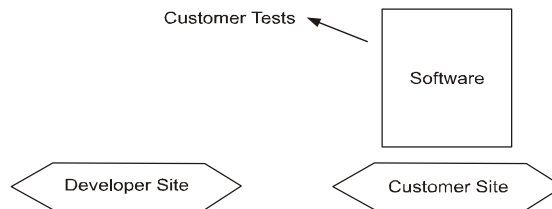


Figure 5.18 Beta Testing

Note that beta testing is often employed as a form of *external* acceptance testing in order to acquire feedback from the 'market'. Often limited public tests known as **beta-versions** are released to groups of people so that further testing can ensure that the end product has few faults or bugs. Sometimes, beta-versions are made available to the open public to increase the feedback.

The advantages of beta testing are listed below:

- Evaluates the entire documentation of software. For example, it examines the detailed description of software code, which forms a part of documentation of software.
- Checks whether software is operating successfully in user environment or not.

Check Your Progress

9. What is unit testing?
10. Explain top-down and bottom-up integration testing.
11. Why is integration test document maintained?
12. Define system testing and its various types.
13. Define validation testing and its various types.

5.6 TESTING TECHNIQUES

Once the software is developed it should be tested in a proper manner before the system is delivered to the user. For this, two techniques that provide systematic guidance for designing tests are used. These techniques are listed below:

- Once the internal working of software is known, tests are performed to ensure that all internal operations of software are performed according to specifications. This is referred to as white box testing.

- Once the specified function for which software has been designed is known, tests are performed to ensure that each function is working properly. This is referred to as black box testing.

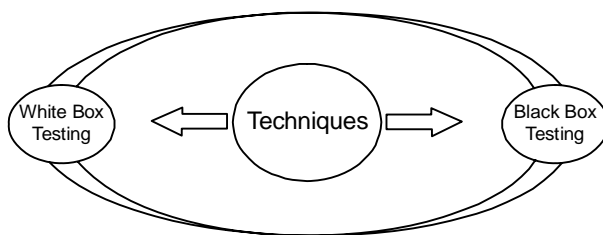


Figure 5.19 Testing Techniques

5.6.1 White Box Testing

White box testing, also known as **structural testing** is performed to check the internal structure of a program. To perform white box testing, tester should have a thorough knowledge of the program code and the purpose for which it is developed. The basic strength of this testing is that the entire software implementation is included while testing is performed. This facilitates error detection even when the software specification is vague or incomplete.

The objective of white box testing is to ensure that the test cases (developed by software testers by using white box testing) exercise each path through a program. That is, test cases ensure that all internal structures in the program are developed according to design specifications. The test cases also ensure that:

- All independent paths within the program have been executed at least once.
- All internal data structures are exercised to ensure validity.
- All loops (simple loops, concatenated loops, and nested loops) are executed at their boundaries and within operational bounds.
- All the segments present between the control structures (like 'switch' statement) are executed at least once.
- Each branch (like 'case' statement) is exercised at least once.
- All the branches of the conditions and the combinations of these conditions are executed at least once. Note that for testing all the possible combinations, a '*truth table*' is used where all logical decisions are exercised for both true and false paths.

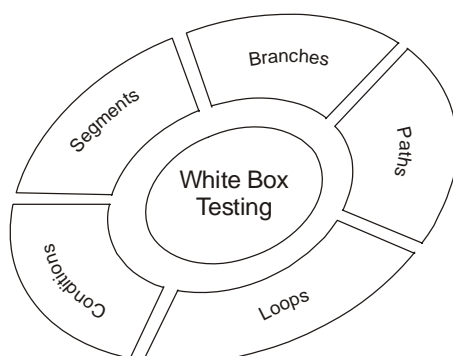


Figure 5.20 White Box Testing

The various advantages and disadvantages associated with white box testing are listed in Table 5.8.

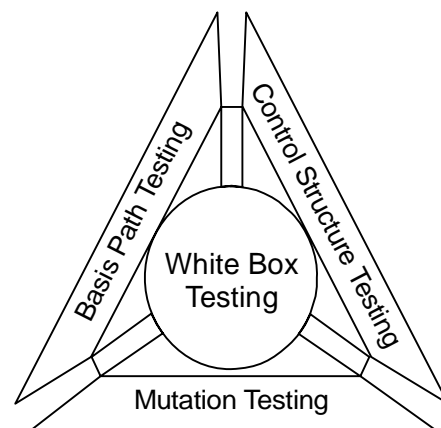
NOTES

Table 5.8 Advantages and Disadvantages of White Box Testing**NOTES**

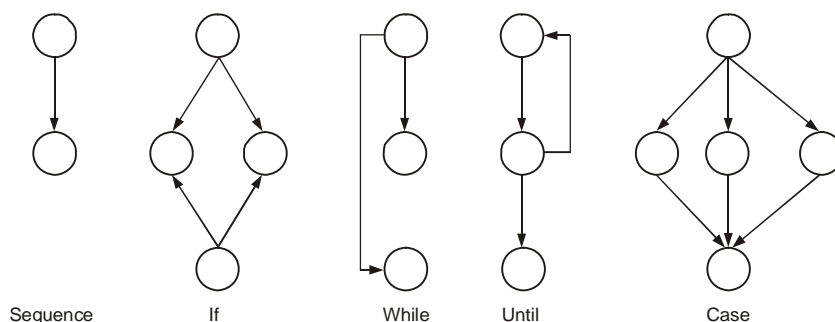
Advantages	Disadvantages
<ul style="list-style-type: none"> Covers the larger part of the program code while testing. Uncovers typographical errors. Detects design errors that occur when incorrect assumptions are made about execution paths. 	<ul style="list-style-type: none"> Tests that cover most of the program code may not be good for assessing the functionality of surprise (unexpected) behaviours and other testing goals. Tests based on design may miss other system problems. Tests cases need to be changed if implementation changes.

The effectiveness of white box testing is commonly expressed in terms of test or code coverage metrics, which measure the fraction of code exercised by test cases. The various types of testing, which occur as part of white box testing are *basis path testing*, *control structure testing*, and *mutation testing*.

(a) Basis Path Testing: Basis path testing enables the software tester to generate test cases in order to develop a logical complexity measure of a component-based design (procedural design). This measure is used to specify the basis set of execution paths. Here, logical complexity refers to the set of paths required to execute all statements present in the program. Note that test cases are generated to make sure that every statement in a program has been executed at least once.

**Figure 5.21** Types of White Box Testing

Creating Flow Graph Flow graph is used to show the logical control flow within a program. To represent the control flow, flow graph uses a notation which is shown in Figure 5.22.

**Figure 5.22** Flow Graph Notation

Flow graph uses different symbols, namely, circles and arrows to represent various statements and flow of control within the program. Circles represent **nodes**, which are used to depict the procedural statements present in the program. A series of process boxes and a decision diamond in a flow chart can be easily mapped into a single node. Arrows represent **edges** or **links**, which are used to depict the flow of control within the program. It is necessary for every edge to end in a node irrespective of whether it represents a procedural statement or not. In a flow graph, area bounded by edges and nodes is known as a **region**. While counting regions, the area outside the graph is also considered as a region. Flow graph can be easily understood with the help of a diagram. For example, in Figure 5.23(a) a flow chart has been depicted, which has been represented as a flow graph in Figure 5.23(b).

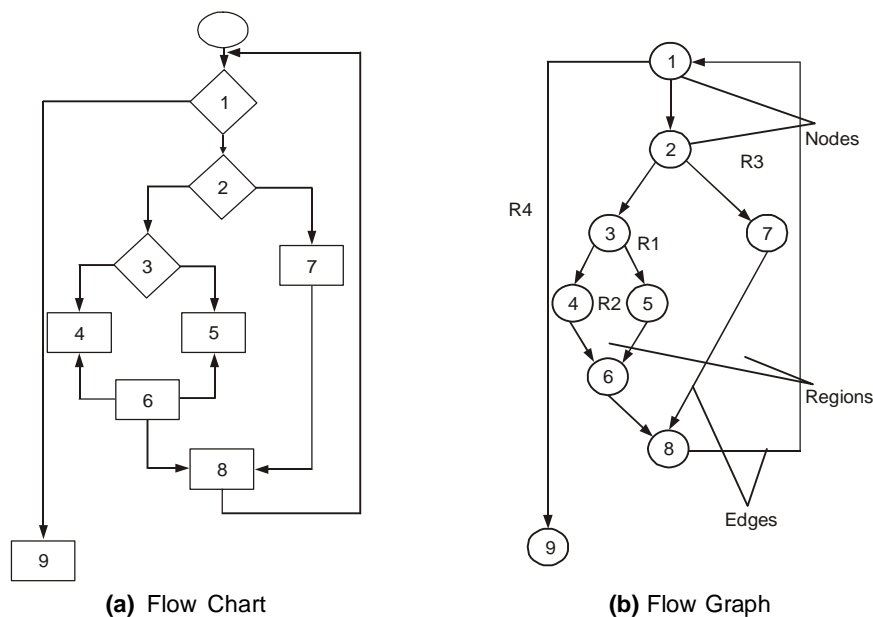


Figure 5.23 Creating Flow Graph

Note that a node that contains a condition is known as **predicated node**, which contains one or more edges emerging out of it. For example, in Figure 5.23(b), node 2 and node 3 represent the predicated nodes.

Finding Independent Paths: A path through the program, which specifies a new condition or a minimum of one new set of processing statements, is known as an **independent path**. For example, in nested 'if' statements there are several conditions that represent independent paths. Note that a set of all independent paths present in the program is known as **basis set**.

A test case is developed to ensure that all the statements present in the program are executed at least once during testing. For example, all the independent paths in Figure 5.23(b) are listed below:

P1: 1 – 9

P2: 1 – 2 – 7 – 8 – 1 – 9

P3: 1 – 2 – 3 – 4 – 6 – 8 – 1 – 9

P4: 1 – 2 – 3 – 5 – 6 – 8 – 1 – 9

where 'P1', 'P2', 'P3', and 'P4' represents different independent paths present in the program.

The number of independent paths present in the program is calculated using cyclomatic complexity, which is defined as the software metric that provides quantitative measure of the logical complexity of a program. This software metric also provides information about the number of tests required to ensure that all statements in the program are executed at least once.

Cyclomatic complexity can be calculated by using any of the three methods listed below:

1. The total number of regions present in the flow graph of a program represents the cyclomatic complexity of the program. For example, in Figure 5.23(b), there are four regions represented by 'R1', 'R2', 'R3', and 'R4', hence, the cyclomatic complexity is four.
2. Cyclomatic complexity can be calculated according to the formula given below:

$$CC = E - N + 2$$

NOTES

where, 'CC' represents the cyclomatic complexity of the program, 'E' represents the number of edges in the flow graph, and 'N' represents the number of nodes in the flow graph. For example, in Figure 5.23(b), 'E' = '11', 'N' = '9'. Therefore, $CC = 11 - 9 + 2 = 4$.

NOTES

3. Cyclomatic complexity can be also calculated according to the formula given below:

$$CC = P + 1$$

where 'P' is the number of predicate nodes in the flow graph. For example, in Figure 5.23(b), $P = 3$. Therefore, $CC = 3 + 1 = 4$.

Note: Cyclomatic complexity can be calculated manually for small program suites, but automated tools are preferred for most operational environments.

Deriving Test Cases: In this, basis path testing is presented as a series of steps and test cases are developed to ensure that all statements present in the program are executed during testing. While performing basis path testing, initially the basis set (independent paths in the program) is derived. The basis set can be derived using the steps given below:

1. *Draw the flow graph of the program:* A flow graph is constructed using symbols previously discussed. For example, a program to find the greater of two numbers is listed below:

```

procedure greater;
integer: a, b, c = 0;
1  enter the value of a;
2  enter the value of b;
3  if a > b then
4    c = a;
   else
5    c = b;
6  end greater

```

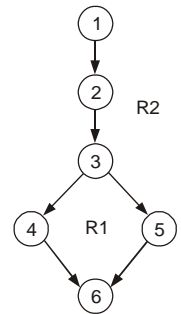


Figure 5.24 Flow Graph to Find the Greater Between Two Numbers

Flow graph for the above program is shown in Figure 5.24.

2. *Determine the cyclomatic complexity of the program using flow graph:* The cyclomatic complexity for flow graph depicted in 6.26 can be calculated as follows:

$$CC = 2 \text{ regions}$$

Or

$$CC = 6 \text{ edges} - 6 \text{ nodes} + 2 = 2$$

Or

$$CC = 1 \text{ predicate node} + 1 = 2$$

3. *Determine all the independent paths present in the program using flow graph:* For the flow graph shown in Figure 5.24, the independent paths are listed below:

$$P1 = 1 - 2 - 3 - 4 - 6$$

$$P2 = 1 - 2 - 3 - 5 - 6$$

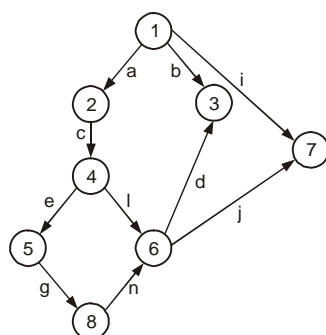
4. *Prepare test cases:* Test cases are prepared to implement the execution of all the independent paths in the basis set. Each test case is executed and compared with the desired results.

Generating Graph Matrix: Graph matrix is used to develop a software tool that in turn helps in carrying out basis path testing. Graph matrix can be defined as a data structure,

which represents the flow graph of a program in a tabular form. This matrix is also used to evaluate the control structures present in the program during testing.

Graph matrix consists of rows and columns that represent nodes present in the flow graph. Note that the size of graph matrix is equal to the number of nodes present in the flow graph. Every entry in the graph matrix is assigned some value known as **link weight**. Adding link weights to each entry makes graph matrix a useful tool for evaluating the control structure of the program during testing.

Flow graph shown in the Figure 5.25(a) is depicted as a graph matrix in Figure 5.25(b). In Figure 5.25(a), numbers are used to identify each node in a flow graph, while letters are used to identify edges in a flow graph. In Figure 5.25(b), a letter entry is made when there exists a connection between two nodes in the flow graph. For example, node 3 is connected to the node 6 by edge 'd' and node 4 is connected to node 2 by edge 'c', and so on.



(a) Flow Graph

	1	2	3	4	5	6	7	8
1		a	b				i	
2				c				
3								
4					e	f		
5								g
6			d				j	
7								
8						h		

(b) Graph Matrix

Figure 5.25 Generating Graph Matrix

(b) Control Structure Testing: Control structure testing is used to enhance the coverage area by testing various control structures (which include logical structures and loops) present in the program. Note that basis path testing is used as one of the techniques for control structure testing. The various types of testing performed under control structure testing are *condition testing*, *data flow testing*, and *loop testing*.

Condition Testing: Condition testing is a test case design method, which ensures that the logical conditions and decision statements are free from errors. The errors present in logical conditions can be incorrect Boolean operators, missing parenthesis in a Boolean expression, error in relational operators, arithmetic expressions, and so on.

The common types of logical conditions that are tested using condition testing are listed below:

- A relational expression, such as 'E1 op E2', where 'E1' and 'E2' are arithmetic expressions and 'op' is an operator.
- A simple condition, such as any relational expression preceded by a 'NOT' (~) operator. For example, (~ E1), where 'E1' is an arithmetic expression and '~' represents 'NOT' operator.
- A compound condition, which is composed of two or more simple conditions, Boolean operators, and parenthesis. For example, (E1 & E2) | (E2 & E3), where 'E1', 'E2', and 'E3' are arithmetic expressions and '&' and '|' represents 'AND' and 'OR' operators.
- A Boolean expression consisting of operands and a Boolean operator, such as 'AND', 'OR', 'NOT'. For example, 'A | B' is a Boolean expression, where 'A' and 'B' are operands and '|' represents 'OR' operator.

NOTES

NOTES

Condition testing is performed using different strategies, namely, branch testing, domain testing, and branch and relational operator testing. **Branch testing** executes each branch (like 'if' statement) present in the module of a program at least once to detect all the errors present in the branch. **Domain testing** tests relational expressions present in a program. For this, domain testing executes all statements of the program that contain relational expressions. **Branch and relational operator testing** tests the branches present in the module of a program using condition constraints. For example,

```
if a > 10
then
print big
```

In this case, branch and relational operator testing verifies that the output produced by the execution of the above code is 'big' only if the value of variable 'a' is greater than '10'.

Data Flow Testing: Data flow testing is a test design technique in which test cases are designed to execute definition and uses of variables in the program. This testing ensures that all variables are used properly in a program. To specify test cases, data flow based testing uses information, such as location at which the variables are defined and used in the program.

To perform data flow based testing, a definition-use graph is constructed by associating variables with nodes and edges in the control flow graph. Once these variables are attached with nodes and edges of control flow graph, test cases can easily determine which variable is used in which part of a program and how data is flowing in the program. Thus, data flow of a program can be tested easily using specified test cases.

Loop Testing: Loop testing is used to check the validity of loops present in the program modules. Generally, there exist four types of loops, which are listed below:

- *Simple loops:* Refers to a loop that has no other loops in it. Consider a simple loop of size 'n'. Size 'n' of the loop indicates that the loop can be traversed 'n' times, that is, 'n' passes are made through the loop. To test simple loops, a number of steps are followed, which are listed below:
 1. Skip the entire loop.
 2. Traverse the loop only once.
 3. Traverse the loop two times.
 4. Make 'a' passes through the loop, where 'a' is a number less than the size of loop 'n'.
 5. Traverse the loop $n - 1$, n , $n + 1$ times.
- *Nested loops:* Loops within loops are known as nested loops. While testing nested loops, number of tests increases as the level of nesting increases. The steps followed for testing nested loops are listed below:
 1. Start with the inner loop and set values of all the outer loops to minimum.
 2. Test the inner loop using the steps followed for testing simple loops while holding the outer loops at their minimum parameter values. Add other tests for values that are either out-of-range or are eliminated.
 3. Move outwards, conducting tests for the next loop. However, keep the nested loops to 'typical' values and outer loops at their minimum values.
 4. Continue testing until all loops are tested.
- *Concatenated loops:* Refers to the loops which contain several loops that may or may not depend on each other. If the loops are independent from each other, then steps in

simple loops are followed. Otherwise, if the loops are dependent on each other, then steps in nested loops are followed.

- **Unstructured loops:** This type of loop should be redesigned so that the use of structured programming constructs can be reflected.

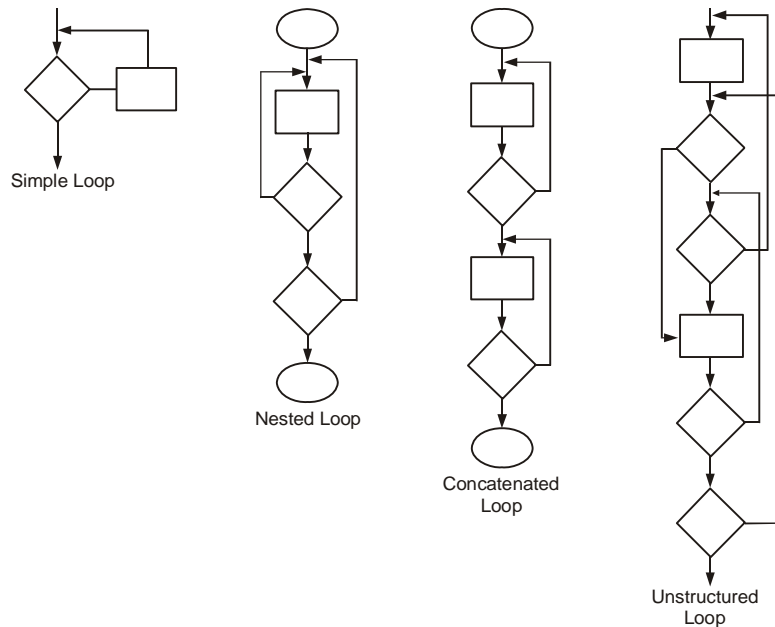


Figure 5.26 Types of Loops

(c) Mutation Testing Mutation testing is a white box method where errors are ‘purposely’ inserted into a program (under test) to verify whether the existing test case is able to detect the error or not. In this testing, mutants of the program are created by making some changes in the original program. The objective is to check whether each mutant produces an output that is different from the output produced by the original program.

In mutation testing, test cases that are able to ‘kill’ all the mutants should be developed. This is accomplished by testing mutants with the developed set of test cases. There can be two possible outcomes when the test cases test the program, either the test case detects the faults or fails to detect faults. If faults are detected, then necessary measures are taken to correct them.

When no faults are detected, it implies that either the program is absolutely correct or the test case is inefficient to detect the faults. Therefore it can be said that mutation testing is performed to check the effectiveness of a test case. That is, if a test case is able to detect these ‘small’ faults (minor changes) in a program, then it is likely that the same test case will be equally effective in finding real faults.

To perform mutation testing, a number of steps are followed, which are listed below:

1. Create mutants of a program.
2. Check both program and its mutants using test cases.
3. Find the mutants that are different from the main program. A mutant is said to be different from the main program if it produces an output, which is different from the output produced by the main program.
4. Find mutants that are equivalent to the program, that is, the mutants that produce same outputs as produced by the program.

NOTES

NOTES

5. Calculate the mutation score using the formula given below:

$$(M = D/N - E)$$

where, M = Mutation score

N = Total number of mutants of the program.

D = Number of mutants different from the main program.

E = Total number of mutants that are equivalent to the main program.

6. Repeat steps 1 to 5 till the mutation score is '1'.

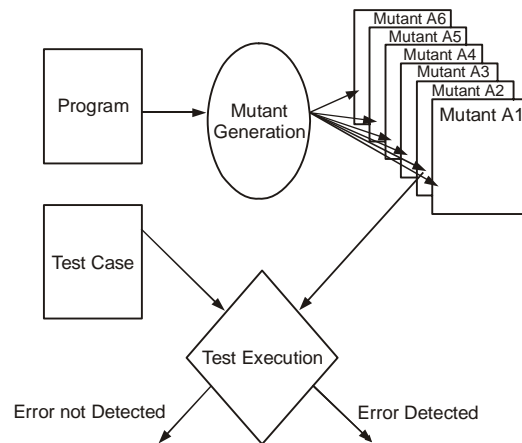


Figure 5.27 Mutation Testing

However, mutation testing is very expensive to run on large programs. Thus, certain tools are used to run mutation tests on large programs. For example, 'Jester' is used to run mutation tests on java code. This tool targets the specific areas of program code, such as changing constants and Boolean values.

5.6.2 Black Box Testing

Black box testing, also known as **functional testing**, checks the functional requirements and examines the input and output data of these requirements. The functionality is determined by observing the outputs to corresponding inputs. For example, when black box testing is used, the tester should only know the 'legal' inputs and what the expected outputs should be, but not how the program actually arrives at those outputs.

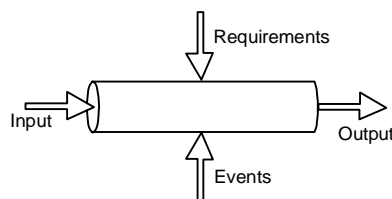


Figure 5.28 Black Box Testing

The black box testing is used to find errors listed below:

- Interface errors, such as functions, which are unable to send or receive data to/from other software.
- Incorrect functions that lead to undesired output when executed.
- Missing functions and erroneous data structures.
- Erroneous databases, which lead to incorrect outputs when software uses the data present in these databases for processing.

- Incorrect conditions due to which the functions produce incorrect outputs when they are executed.
- Termination errors, such as certain conditions due to which function enters a loop that forces it to execute indefinitely.

In this testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. Note that test cases are derived from these specifications without considering implementation details of the code. The outputs are compared with user requirements and if they are as specified by the user, then the software is considered to be correct, else the software is tested for the presence of errors in it.

The various advantages and disadvantages associated with black box testing are listed in Table 5.9.

Table 5.9 Advantages and Disadvantages of Black Box Testing.

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Tester requires no knowledge of implementation and programming language used. ▪ Reveals any ambiguities and inconsistencies in the functional specifications. ▪ Efficient when used on larger systems. ▪ Non-technical person can also perform black box testing. 	<ul style="list-style-type: none"> ▪ Only small number of possible inputs can be tested as testing every possible input consumes a lot of time. ▪ There can be unnecessary repetition of test inputs if the tester is not informed about the test cases that software developer has already tried. ▪ Leaves many program paths untested. ▪ Cannot be directed towards specific segments of code, hence is more error prone.

The various methods used in black box testing are equivalence class partitioning, boundary value analysis, orthogonal array testing, and cause effect graphing. In **equivalence class partitioning** the test inputs are classified into equivalence classes such that one input checks (validates) all the input values in that class. In **boundary value analysis** the boundary values of the equivalence classes are considered and tested. In **orthogonal array testing** faults in the logic of the software component are considered and tested. In **cause-effect graphing**, cause-effect graphs are used to design test cases, which provides all the possible combinations of inputs to the program.

(a) Equivalence Class Partitioning : Equivalence class partitioning method tests the validity of outputs by dividing the input domain into different classes of data (known as equivalence classes) using which test cases can be easily generated. Test cases are designed with the purpose of covering each partition at least once. If a test case is able to detect all the errors in the specified partition, then the test case is said to be an ideal test case.

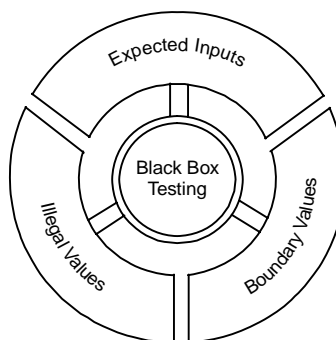


Figure 5.29 Types of Error Detection in Black Box Testing

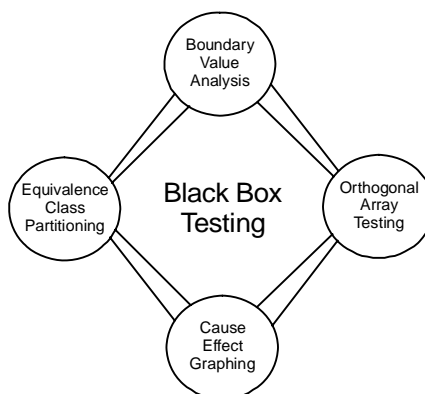


Figure 5.30 Types of Black Box Testing

NOTES

NOTES

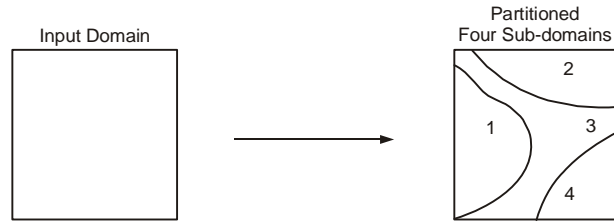


Figure 5.31 Input Domain and Equivalence Classes

An equivalence class depicts valid or invalid states for the input condition. An input condition can be either a specific numeric value, a range of values, a Boolean condition, or a set of values. Generally, guidelines that are followed for generating the equivalence classes are listed below:

- If an input condition is Boolean, then there will be two equivalence classes: one valid and one invalid class.
- If input consists of a specific numeric value, then there will be three equivalence classes: one valid and two invalid classes.
- If input consists of a range, then there will be three equivalence classes: one valid and two invalid classes.
- If an input condition specifies a member of a set, then there will be one valid and one invalid equivalence class.

To understand equivalence class partitioning properly, let us consider an example. This example is explained in series of steps listed below:

1. Suppose that a program 'P' takes an integer 'X' as input.
2. Now for this input we have ' $X < 0$ ' and ' $X > 0$ '.
3. If ' $X < 0$ ' then program is required to perform task T1 and if $X > 0$ then task T2 is performed.
4. The input domain is as large as 'X' and it can assume a large number of values. Therefore the input domain (P) is partitioned into two equivalence classes and all test inputs in the $X < 0$ and $X > 0$ equivalence classes are considered to be equivalent.
5. Now, as shown in Figure 5.32 *independent* test cases are developed for $X < 0$ and $X > 0$.

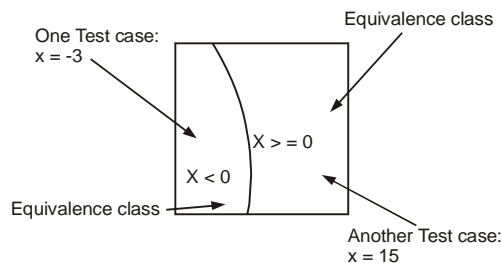


Figure 5.32 Test Case and Equivalence Class

(b) Boundary Value Analysis: Boundary value analysis (BVA) is a black box test design technique where test cases are designed based on boundary values (that is, test cases are designed at the edge of the class). Boundary value can be defined as an input value or output value, which is at the edge of an equivalence partition or at the smallest incremental distance on either side of an edge, for example the minimum or maximum value of a range.

BVA is used since it has been observed that a large number of errors occur at the boundary of the given input domain rather than at the middle of the input domain. Note that boundary value analysis complements the equivalence partitioning method. The only difference is that in BVA, test cases are derived for both input domain and output domain while in equivalence partitioning, test cases are derived only for input domain.

Generally, the test cases are developed in boundary value analysis using certain guidelines, which are listed below:

- If input consists of a *range* of certain values, then test cases should be able to exercise both the values at the boundaries of the range and the values that are just above and below boundary values. For example, for the range $-0.5 \leq X \leq 0.5$, the input values for a test case can be -0.4 , -0.5 , 0.5 , 0.6 .
- If an input condition specifies a *number* of values, then test cases are generated to exercise the minimum and maximum numbers and values just above and below these limits.
- If input consists of a list of numbers, then the test case should be able to exercise the first and the last elements of the list.
- If input consists of certain data structures (like arrays), then the test case should be able to execute all the values present at the boundaries of the data structures, such as the maximum and minimum value of an array.

(c) Orthogonal Array Testing: Orthogonal array testing can be defined as a mathematical technique that determines the variations of parameters that need to be tested. This testing is performed when limited data is to be given as input. Orthogonal array testing is useful in finding errors in the software where incorrect logic is applied. Orthogonal array testing provides a way to select tests that:

- Guarantee testing of pair wise combination of all selected variables.
- Create an efficient way to test all combinations of variables using fewer test cases as compared to other black box testing methods, such as boundary value analysis, equivalence class partitioning, and cause effect graphing.
- Create test cases that have even distribution of all pair wise combinations of variables in orthogonal array.
- Execute complex combinations of all the variables.

To understand orthogonal array testing, it is important to understand orthogonal arrays, which are two-dimensional arrays of numbers. In these arrays, if any two columns are chosen then the complete distribution of pair-wise combination of values present in the array can be obtained. To perform orthogonal array testing, follow the steps listed below:

1. Find all the independent variables that need to be tested for interaction. This gives the factors present in the array.
2. Decide the maximum number of values that each independent variable follows. This gives the number of levels present in the array.
3. Find an orthogonal array that has minimum number of runs. An orthogonal array with the minimum number of runs is one that has maximum factors and at least as many levels as decided for each factor.
4. Map factors and values on to the array.
5. Choose values for 'left over' levels, that is, the levels for which there is no value mapped in the array.
6. Convert runs into test cases.

NOTES

NOTES

In the above steps, **runs** refer to the number of rows in the array. This directly translates into the number of test cases that will be generated by the orthogonal analysis testing technique. **Factors** refer to the number of columns in an array. This directly translates to the maximum number of variables that can be handled by this array. **Levels** refer to the maximum number of values that can be taken on by any single factor.

To understand orthogonal array testing properly, let us consider an example of a web page. This web page consists of three sections, namely, *top*, *middle*, and *bottom*, these sections can be individually shown or hidden from the users. According to the procedure of orthogonal array testing, the interactions among different sections can be tested as follows:

- Factors = 3, as there are three sections in the web page.
- Levels = 2, as variables can have either hidden or visible state.
- Draw orthogonal array = 2^3 , as there are two levels and three factors.

Table 5.10 Orthogonal Array

	Orthogonal array before mapping factors		
	Factor 1	Factor 2	Factor 3
Run 1	0	0	0
Run 2	0	1	1
Run 3	1	0	1
Run 4	1	1	0
	Orthogonal array after mapping factors		
Test 1	Hidden	Hidden	Hidden
Test 2	Hidden	Visible	Visible
Test 3	Visible	Hidden	Visible
Test 4	Visible	Visible	Hidden

The left over levels = 0. Now generate test cases from each run. Four test cases are generated to check the conditions listed below:

- Home page is displayed and all other sections are hidden.
- Home page and all other sections rather than top section are displayed.
- Home page and all other sections rather than middle section are displayed.
- Home page and all other sections rather than bottom section are displayed.

(d) Cause-Effect Graphing: Cause-effect graphing is a test design technique where test cases are designed using cause-effect graphs. A cause-effect graph is a graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases. Test cases are generated to test all the possible combinations of inputs provided to the program being tested.

One of the major drawbacks of using equivalence partitioning and boundary value analysis is that both these methods test every input given to a program independently. This drawback is avoided in cause effect graphing where combinations of inputs are used instead of individual inputs. To use cause effect graphing method, a number of steps are followed, which are listed below:

1. List the cause (input conditions) and effects (outputs) of the program.
2. Create a cause-effect graph.
3. Convert graph into decision table.
4. Modify decision table rules to test cases.

For generating test cases, initially, all causes and effects are allocated unique numbers, which are used to identify them. After allocating numbers, the cause due to which a particular effect occurred is determined. Next, the combinations of various conditions that make the effect 'true' are recognised. A condition has two states, 'true' and 'false'. A condition is 'true' if it causes the effect to occur, otherwise it is 'false'. The conditions are combined using Boolean operators, such as 'AND' (&), 'OR' (|) and 'NOT' (~). Finally, a test case is generated for all possible combinations of conditions.

The various symbols used in cause-effect graph are shown in Figure 5.33. The left side in the figure depicts the various *logical* associations among causes c_i and effects e_i and the dashed notation in the right side indicates the various *constraint* associations that can be applied to either causes or effects.

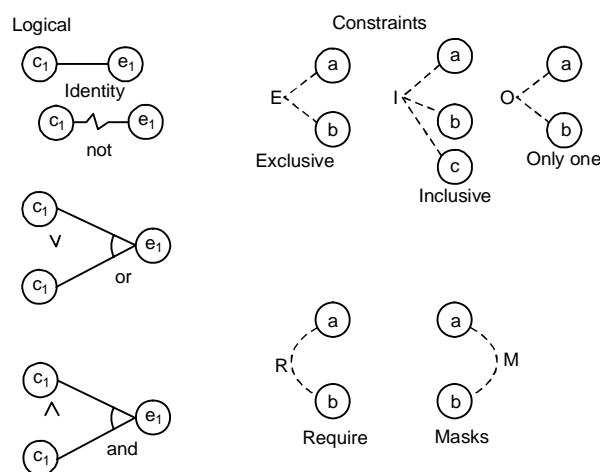


Figure 5.33 Logical and Constraints Associations

To understand cause-effect graphing properly, let us consider an example. Suppose a triangle is drawn with inputs (x, y, z). The values of these inputs are given between '0' and '100'. Using these inputs, three outputs are produced, namely, isosceles triangle, equilateral triangle or no triangle is made (if values of x, y, z are less than 60°).

1. Using the steps of cause-effect graphing, initially the causes and effects of the problem are recognised, which are listed in Table 5.11.

Table 5.11 Causes and Effects

Cause	Effect
C1: side x is less than the sum of sides y and z.	E1: no triangle is formed.
C2: sides x, y, z are equal.	E2: equilateral triangle is formed.
C3: side x is equal to side y.	E3: isosceles triangle is formed.
C4: side y is equal to side z.	
C5: side x is equal to side z.	

NOTES

NOTES

2. The cause effect graph is generated as shown in Figure 5.34.

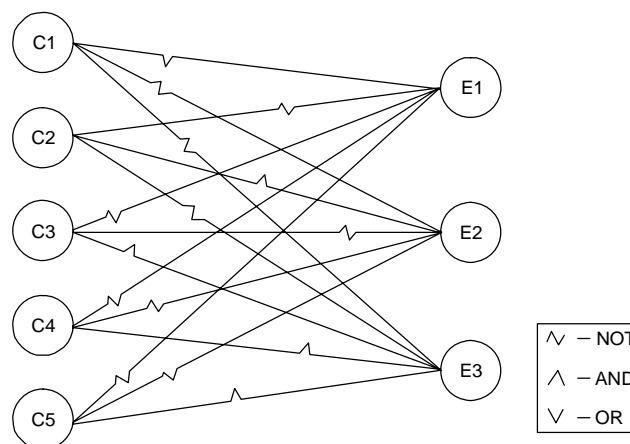


Figure 5.34 Cause-Effect Graph

3. A decision table (A table that shows a set of conditions and the actions resulting from them) is drawn as shown in Table 5.12.

Table 5.12 Decision Table

Conditions					
C1: $x < y + z$	0	X	X	X	X
C2: $x = y = z$	X	1	X	X	X
C3: $x = y$	X	X	1	X	X
C4: $y = z$	X	X	X	1	X
C5: $x = z$	X	X	X	X	1
E1: not a triangle	1				
E2: equilateral triangle		1			
E3: isosceles triangle			1	1	1

4. Each combination of conditions for an effect in Table 5.12 is a test case.

5.6.3 Difference between White Box and Black Box Testing

Although white box testing and black box testing are used together for testing many programs, there are several considerations that make them different from each other. Black box testing detects errors of *omission*, which are errors occurring due to non-accomplishment of user requirements. On the other hand, white box testing detects errors of *commission* which are errors occurring due to non-implementation of some part of software code. The other differences between white box testing and black box testing are listed in Table 5.13.

Table 5.13 Difference between White Box and Black Box Testing

Basis	White Box Testing	Black Box Testing
Purpose	<ul style="list-style-type: none"> It is used to test the internal structure of software. 	<ul style="list-style-type: none"> It is used to test the functionality of software.
	<ul style="list-style-type: none"> It is concerned only with testing software and does not guarantee the complete implementation of all the specifications mentioned in user requirements. 	<ul style="list-style-type: none"> It is concerned only with testing specifications and does not guarantee that all the components of software that are implemented are tested.
	<ul style="list-style-type: none"> It addresses flow and control structure of a program. 	<ul style="list-style-type: none"> It addresses validity, behaviour and performance of software.
Stage	<ul style="list-style-type: none"> It is performed in the early stages of testing. 	<ul style="list-style-type: none"> It is performed in the later stages of testing.
Requirement	<ul style="list-style-type: none"> Knowledge of the internal structure of a program is required for generating test case. 	<ul style="list-style-type: none"> No knowledge of the internal structure of a program is required to generate test case.
Test Cases	<ul style="list-style-type: none"> Here test cases are generated based on the actual code of the module to be tested. 	<ul style="list-style-type: none"> Here the internal structure of modules or programs is not considered for selecting test cases.
Example	<ul style="list-style-type: none"> The inner software present inside the calculator (which is known by the developer only) is checked by giving inputs to the code. 	<ul style="list-style-type: none"> In this testing, it is checked whether the calculator is working properly or not by giving inputs by pressing the buttons in the calculator.

NOTES**5.6.4 Gray Box Testing**

Gray box testing does not require full knowledge of the internals of the software that is to be tested instead it is a test strategy, which is based partly on the internals. This testing technique is often defined as a mixture of black box testing and white box testing techniques. Gray box testing is especially used in web applications, because these applications are built around loosely integrated components that connect through relatively well-defined interfaces.

Testing in this methodology is done from the outside of the software similar to black box testing. However, testing choices are developed through the knowledge of how the underlying components operate and interact. Some points noted in gray box testing are listed below:

- Gray box testing is platform and language independent.
- The current implementation of gray box testing is heavily dependent on the use of a host platform debugger(s) to execute and validate the software under test.
- Gray box testing can be applied in real-time systems.
- Gray box testing utilises automated software-testing tools to facilitate the generation of test cases.
- Module drivers and stubs are created by automation means thus, saving time of testers.

5.7 OBJECT-ORIENTED TESTING

The shift from traditional to object-oriented environment involves looking at and reconsidering old strategies and methods for testing software. The traditional programming consists of procedures operating on data, while the object-oriented paradigm focuses on objects that

Check Your Progress

14. Define white box testing.
15. How is basis path testing different from control structure testing?
16. When is black box testing useful?
17. Differentiate between various methods used in black box testing.

NOTES

are instances of classes. The object-oriented (OO) paradigm provides a better understanding of requirements in terms of identifying and specifying the objects, their behaviours, the services provided by objects, object interactions, and their constraints. It is observed that the OO paradigm significantly increases software reusability, extendibility, interoperability, and reliability.

With the adoption of object-oriented paradigm, the various life cycle activities also have acquired new perspectives. For example, waterfall methodology has been replaced by iterative-incremental approach. This has been done to meet tighter delivery schedules and dynamically changing requirements. Similarly requirement analysis not only identifies the functional specifications, but also the business model, actors, objects, and interactions between them. Analysis and design has changed from making a low level pseudo-code to creating class and state-chart diagrams.

Testing too has undergone a major transformation in its approach, environments and tools. OO software testing deals with new problems introduced by the powerful new features of OO languages. These features (such as encapsulation, inheritance, polymorphism, and dynamic binding) provide visible benefits in software designing and programming. Object-oriented testing can be used in number of ways, which are listed below:

- Testing object-oriented software.
- Using object-oriented tools to test object-oriented as well as non object-oriented software.
- Using object-oriented techniques to test object-oriented software.

Object-oriented programs can be tested at four levels the algorithmic level, class level, cluster level, and system level. At the **algorithmic level**, individual methods are tested in isolation. Here conventional testing techniques can be applied without much change. At the **class level**, the objective is to verify the integrity of a class by testing it as an individual entity. The **cluster level** is concerned about the integration of classes. The focus is on the synchronisation of different concurrent components as well as interclass method invocations. At the **system level**, interactions among clusters are tested.

5.7.1 Testing of Classes

It is now accepted that class forms the basic unit of testing in object-oriented programs. Class testing in OO software is driven by the operations encapsulated by the class and the state behaviour of the class. This is unlike unit testing done in conventional software, which focuses on the algorithmic detail of the module.

- **Testing individual classes:** Programmers who are involved in the development of the class conduct testing at the object level. Test cases for individual objects can be drawn from requirements specifications, models, and the language used. In addition, structural testing methods, such as boundary analysis are extensively used.
- **Testing groups of classes:** The next unit is aggregation of classes (also referred to as cluster) or a small subsystem. A cluster/component is a set of classes, which are related to each other through association, aggregation or dependency. The methods of a class are tested in isolation, and then in parallel with other collaborating classes. This can be viewed as integration testing among the classes. System testing is initiated when all cluster/component tests are completed.

Usually there is a misconception that if individual classes are well designed and have proved to work in isolation, then there is no need to test the interactions between two or more classes when they are integrated. However, this is not true because sometimes there can be errors, which can be detected only through integration of classes. Also, it is possible that if a class does not contain a bug, it may still be used in a wrong way by another class, leading to system failure.

5.7.2 Developing Test Cases in Object-Oriented Testing

Test case design in object-oriented testing is based on the conventional methods, however, these test cases should encompass special features so that they can be used in the object-oriented environment. The points that should be noted while developing test cases in object-oriented environment are listed below:

- Each test case should be uniquely identified and explicitly associated with the class to be tested.
- The purpose of the test should be stated clearly.
- A list of testing steps should be developed for each test and should contain the following:
 - A list of specified states for the object that is to be tested.
 - A list of messages and operations, which will be exercised as a consequence of the test.
 - A list of exceptions, which may occur as the object is tested.
 - A list of external conditions (changes in the environment external to the software) that must exist in order to properly conduct the test.
 - Supplementary information that aids in understanding or implementing the test.

5.7.3 Object-Oriented Testing Methods

Currently, most software development organisations are still in the process of observing and/or switching over to the OO paradigm. It is anticipated that OO software testing will receive much attention in the software development process as the importance of this paradigm increases. The methods used for performing object-oriented testing include **state-based testing**, **fault-based testing**, **scenario-based testing**, and **Unified modelling language-based testing**.

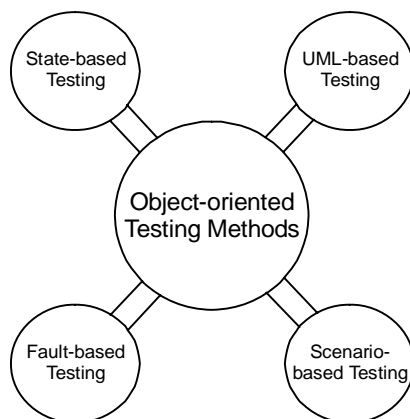


Figure 5.35 Object-Oriented Testing Methods

(a) State-based Testing: State-based testing is used to verify whether the methods (a procedure that is executed by an object) of a class are interacting properly with each other or not. This testing seeks to exercise the transitions among the states based upon the identified inputs. For this, finite-state machine (FSM) or state-transition diagram is constructed to represent the change of states that occur in the program under test.

For testing the methods, state-based testing generates test cases, which check whether the method is able to change the state of object as expected or not. If any method of the class

NOTES

NOTES

is not able to change the state of object as expected, then the method is said to contain errors.

To perform state-based testing, a number of steps are followed, which are listed below:

1. Derive a new class from an existing class with some additional features, which are used to examine and set the state of the object.
2. Next, test driver is written. This test driver contains a main program to create an object, send messages to set the state of object, send messages to invoke methods of the class that is being tested and send messages to check the final state of the object.
3. Finally, stubs are written. These stubs call the untested methods.

(b) Fault-based Testing: In fault-based testing, test cases are developed to determine a set of plausible faults. Here, the focus is on falsification. In this testing, tester does not focus on a particular coverage of a program or its specification, but on concrete faults that should be detected. The focus on possible faults enables testers to incorporate their expertise in both the application domain and the particular system under test. Since testing can only prove the existence of errors and not their absence, this testing approach is considered to be an effective testing method and is hence often used when security or safety of a system is to be tested.

Fault-based testing starts by examining the analysis and design model of object-oriented software. These models provide an overview of the problems that can occur during implementation of software. The faults occur in both operation calls and various types of messages (like a message sent to invoke an object). These faults are unexpected outputs, incorrect messages or operations, and incorrect invocation. The faults can be recognised by determining the behaviour of all operations performed to invoke the methods of a class.

(c) Scenario-based Testing: Scenario-based testing is used to detect errors that are caused due to incorrect specifications and improper interactions among various segments of the software. Incorrect interactions often lead to incorrect outputs that can cause malfunctioning of some segments of software. The use of scenarios in testing is a common way of describing how a representative user might execute a task or achieve a goal within a specific context or environment. Note that these scenarios are more context and user specific instead of being product specific. Generally, the structure of a scenario includes the following:

- A condition under which the scenario runs.
- A goal to achieve, which can also be a name of the scenario.
- A set of steps of actions.
- An end condition at which the goal is achieved.
- A possible set of extensions written as scenario fragments.

Scenario-based testing combines all the classes that support a use case (scenarios are subset of use cases) and executes a test case to test them. Execution of all the test cases ensures that all methods in all the classes are executed at least once during testing. However, it is difficult to test all the objects (present in the classes combined together) collectively. Thus, rather than testing all objects collectively, they are tested using either top-down or bottom-up integration approach.

This testing is considered to be the most effective method as in this method, scenarios can be organised in such a manner that the most likely scenarios are tested first with unusual or exceptional scenarios considered later in the testing process. This satisfies a fundamental principle of testing that most testing effort should be devoted to those paths of the system that are mostly used.

Note: A use case collects all the scenarios together, specifying the manner in which the goal can succeed or fail.

(d) Unified Modelling Language-based Testing: Unified modelling language (UML) is a semi-formal modelling language that is commonly used in object-oriented software development. It includes class diagrams, activity diagrams, sequence diagrams, collaboration diagrams, and state diagrams. Class diagrams describe general relationships amongst classes. Activity, sequence, collaboration, and state diagrams describe the interaction of objects at different levels of abstraction. For example, activity diagrams may illustrate a use case from the users' point of view while sequence diagrams express interactions among objects in greater detail.

UML techniques have been proposed to test object-oriented systems based on UML specifications. In UML-based technique, following points are noted:

- The main test plan consists of use case sequences.
- Each use case is associated with a sequence diagram. The diagram is translated formally into a regular expression.
- Every term in regular expression represents either a use case scenario or a set of scenarios in the presence of iteration symbols. A guard condition expressed in an object constraint language is associated with each path in the sequence diagram.
- The exact operation sequences to be executed for each term, including inter-dependencies, are also identified.
- The testing 'oracles' are specified in terms of the post-conditions of the sequences of operations, specified also in the object constraint language.

5.8 LET US SUMMARIZE

1. Through effective software testing, the software can be examined for correctness, comprehensiveness, consistency, and adherence to standards. This helps in delivering high quality software product, lowering of maintenance costs, and leads to more contented users. Software testing is often used in association with the terms verification and validation.
2. Verification refers to checking or testing of items, including software, for conformance and consistency with an associated specification. Validation refers to the process of checking that the developed software is according to the requirements specified by the user.
3. The main reasons due to which errors occur in the software are unclear requirements, software complexity, programming errors, changing requirements, time pressure, and poorly documented code.
4. Testing is performed either by software developers or by independent test group.
5. The ease with which a program is tested is known as testability. Testability can be defined as the degree to which a program facilitates the establishment of test criteria and execution of tests to determine whether the criteria have been met or not.
6. Test plan is a document, which is developed to specify the objectives, scope, method, and purpose of software testing. A complete test plan helps people outside the test group to understand the 'why' and 'how' of product validation. While an incomplete test plan can result in a failure to check how the software works on different hardware and operating systems or when software is used with other software.

NOTES

Check Your Progress

18. Object-oriented programs can be tested at four levels. Explain all the levels.
19. List the points that should be noted while developing test cases in object-oriented environment.
20. Define the methods used for performing object-oriented testing.

NOTES

7. Test case is defined as a set of input values, execution preconditions, expected results, and execution post conditions developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.
8. There are four levels of testing unit testing, integration testing, system testing, and acceptance testing. The lowest level of testing is unit testing, which is used to test the individual units of software.
9. The various tests that are performed as a part of unit testing are module interface, local data structure, boundary conditions, all independent paths, and error handling paths.
10. After unit testing, integration testing is performed to ensure that all the modules continue to work in accordance with customer requirements even after integration.
11. Various approaches used to perform incremental integration testing are top-down integration testing, bottom-up integration testing, regression testing, and smoke testing.
12. During top-down integration testing, software is developed and tested by integrating the individual modules, moving downwards in the control hierarchy.
13. Bottom-up integration testing combines and tests modules present at the lower levels proceeding towards the modules present at higher levels of control hierarchy.
14. Regression testing is used to re-test the software or part of it to ensure that no previously working components, functions, or features fail as a result of error correction and due to integration of modules.
15. Smoke testing ensures that the most crucial functions of a program work correctly.
16. Software is integrated with other elements, such as hardware, people, and database to form a computer-based system. This system is then checked for errors using system testing. Various kinds of system testing are recovery testing, security testing, stress testing, and performance testing.
17. Recovery testing is a system test, which forces system to fail in different ways and verifies that the software recovers from expected or unexpected events without loss of data or functionality.
18. Systems with sensitive information are generally the target for improper or illegal use. Therefore, protection mechanisms are required to restrict unauthorised access to the system. To avoid any improper usage, security testing is performed, which identifies and removes software flaws that may potentially lead to security violations.
19. Stress testing is designed to test the software with abnormal situations. These abnormal situations arise when resources are required in abnormal quantity, frequency, or volume.
20. Performance testing checks the run-time performance of the software (especially real-time and embedded systems) in the context of the entire computer based system. This testing is used to verify the load, volume, and response times as defined by requirements.
21. Validation testing/acceptance testing is performed to determine whether software meets all the functional, behavioural, and performance requirements or not. During acceptance testing, software is tested and evaluated by a group of users either at the developer's site or user's site. This enables the users to test the software themselves and analyse whether it is meeting their requirements or not.
22. Alpha testing is conducted by the users at the developer's site. On completion of alpha testing, users report the errors to software developers so that they can correct them.
23. Beta testing assesses performance of software at user's site. This testing is 'live' testing and is conducted in an environment, which is not controlled by the developer.

24. Once the software is developed it should be tested in a proper manner before the system is delivered to the user. For this white box testing and black box testing technique are used.
25. White box testing, also known as structural testing, is performed to check the internal structure of a program. To perform white box testing, tester should have a thorough knowledge of the program code and the purpose for which it is developed. Various types of white box testing are basis path testing, control structure testing, and mutation testing.
26. Basis path testing enables the software tester to generate test cases in order to develop a logical complexity measure of a component-based design (procedural design). This measure is used to specify the basis set of execution paths.
27. Set of all the independent paths within the program is known as basis set.
28. Control structure testing is used to enhance the coverage area by testing various control structures (which include logical structures and loops) present in the program. The various types of testing performed under control structure testing are condition testing, data flow testing, and loop testing.
29. Condition testing is a test case design method, which ensures that the logical conditions and decision statements are free from errors.
30. Data flow testing is a test design technique in which test cases are designed to execute definition and uses of variables in the program. This testing ensures that all variables are used properly in a program.
31. Loop testing is used to test simple loops, nested loops, concatenated loops, and unstructured loops.
32. Mutation testing is a white box method where errors are 'purposely' inserted into a program (under test) to verify that whether the existing test case is able to detect the error or not. In this testing, mutants of the program are created by making some changes in the original program.
33. Black box testing, also known as functional testing, checks the functional requirements and examines the input and output data of these requirements. The functionality is determined by observing the outputs to corresponding inputs. The various methods used in black box testing are equivalence class partitioning, boundary value analysis, orthogonal array testing, and cause effect graphing.
34. In equivalence class partitioning, the test inputs are classified into equivalence classes such that one input checks all the input values in that class. This method tests the validity of outputs by dividing the input domain into different classes of data (known as equivalence classes) using which test cases can be easily generated. Test cases are designed with the purpose of covering each partition at least once.
35. In boundary value analysis, the boundary values of the equivalence classes are considered and tested. BVA is used since it has been observed that a large number of errors occur at the boundary of the given input domain rather than at the middle of the input domain.
36. Orthogonal array testing can be defined as a mathematical technique that determines the variations of parameters that need to be tested. This testing is performed when limited data is to be given as input. Orthogonal array testing is useful in finding errors in the software where incorrect logic is applied.
37. Cause-effect graphing is a test design technique where test cases are designed using cause-effect graphs. A cause-effect graph is a graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases.

NOTES

NOTES

38. Gray box testing does not require full knowledge of the internals of the software that is to be tested, instead it is a test strategy, which is based partly on the internals. This testing technique is often defined as a mixture of black box testing and white box testing techniques.
39. Object-oriented software testing deals with new problems introduced by the powerful new features of OO languages. These features (such as encapsulation, inheritance, polymorphism, and dynamic binding) provide visible benefits in software designing and programming. The various methods to perform object-oriented testing are state-based testing, fault-based testing, scenario-based testing, and UML-based testing.

5.9 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. **Verification** refers to checking or testing of items, including software, for conformance and consistency with an associated specification. For verification, techniques like reviews, analysis, inspections and walkthroughs are commonly used. While **validation** refers to the process of checking that the developed software is according the requirements specified by the user.
2. **Bug** is defined as a logical mistake, which is caused by a software developer while writing the software code. **Error** is defined as the difference between the outputs produced by the software and the output desired by the user (expected output). **Fault** is defined as the condition that leads to malfunctioning of the software. Malfunctioning of software is caused due to several reasons, such as change in the design, architecture, or software code. Defect that causes error in operation or negative impact is called failure. **Failure** is defined as the state in which software is unable to perform a function according to user requirements. Bugs, errors, faults, and failures prevent software from performing efficiently and hence, cause the software to produce unexpected outputs.
3. Independent test group (ITG) is responsible to detect errors that may have been neglected by the software developers. ITG tests the software without any discrimination since the group is not directly involved in the development process. However, the testing group does not completely take over the testing process, instead it works with the software developers in the software project to ensure that testing is performed in an efficient manner.
4. A test plan describes how testing would be accomplished. A test plan is defined as a document that describes the objectives, scope, method, and purpose of software testing. This plan identifies test items, features to be tested, testing tasks and the persons involved in performing these tasks.
5. The various components of the test plan are listed below:

Component	Purpose
Responsibilities	Assigns responsibilities and keeps people on track and focused.
Assumptions	Avoids misunderstandings about schedules.
Test	Outlines the entire process and maps specific tests. The testing scope, schedule, and duration are also outlined.
Communication	Communication plan (who, what, when, how about the people) is developed.
Risk Analysis	Identifies areas that are critical for success.
Defect Reporting	Specifies how to document a defect so that it can be reproduced, fixed, and retested.
Environment	Specifies the technical environment, data, work area, and interfaces used in testing. This reduces or eliminates misunderstandings and sources of potential delay.

6. A test case is a document that describes an input, action, or event and its expected result, in order to determine whether the software or a part of the software is working correctly or not.
7. Incomplete and incorrect test cases lead to incorrect and erroneous test outputs. To avoid this, a test case should be developed in such a manner that it checks software with all possible inputs. This process is known as **exhaustive testing** and the test case, which is able to perform exhaustive testing, is known as **ideal test case**.
8. The test plan is not concerned with the details of testing a unit. Moreover, it does not specify the test cases to be used for testing units. Thus, test case specification is done in order to test each unit separately. Depending on the testing method specified in test plan, features of unit that need to be tested are ascertained.
9. Unit testing is performed to test the individual units of software. Since software is made of a number of units/modules, detecting errors in these units is simple and consumes less time, as they are small in size.
10. **Top-down integration testing:** In this testing, software is developed and tested by integrating the individual modules, moving downwards in the control hierarchy. In top-down integration testing, initially only one module known as the main control module is tested. After this, all the modules called by it are combined with it and tested. This process continues till all the modules in the software are integrated and tested.

Bottom-up integration testing: In this testing, individual modules are integrated starting from the bottom and then moving upwards in the hierarchy. That is, bottom-up integration testing combines and tests the modules present at the lower levels proceeding towards the modules present at higher levels of control hierarchy.

11. To understand the overall procedure of software integration, a document known as test specification is prepared. This document provides information in the form of test plan, a test procedure, and actual test results.
12. System testing can be defined as a testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. The various types of system testing are defined below:
 - **Recovery testing:** Recovery testing is a system test, which forces the system to fail in different ways and verifies that the software recovers from expected or unexpected events without loss of data or functionality.
 - **Security testing:** To avoid any improper usage, security testing is performed which identifies and removes software flaws that may potentially lead to security violations.
 - **Stress testing:** Stress testing is designed to test the software with abnormal situations. These abnormal situations arise when resources are required in abnormal quantity, frequency, or volume.
 - **Performance testing:** Performance testing checks the run-time performance of the software (especially real-time and embedded systems) in the context of the entire computer based system. This testing is used to verify the load, volume, and response times as defined by requirements.
13. Validation testing is performed to determine whether software meets all the functional, behavioural, and performance requirements or not. The various types of validation testing defined below:
 - **Alpha testing:** Alpha testing is conducted by the users at the *developer's* site. In other words, this testing assesses the performance of software in the environment in which it is developed.

NOTES

NOTES

- **Beta testing:** Beta testing assesses performance of software at *user's* site. This testing is 'live' testing and is conducted in an environment, which is not controlled by the developer. That is, this testing is performed without any interference from the developer.
14. White box testing is performed to check the internal structure of a program. To perform white box testing, tester should have a thorough knowledge of the program code and the purpose for which it is developed.
 15. Basis path testing enables the software tester to generate test cases in order to develop a logical complexity measure of a component-based design (procedural design). While, control structure testing is used to enhance the coverage area by testing various control structures (which include logical structures and loops) present in the program. Note that basis path testing is used as one of the techniques for control structure testing.
 16. The black box testing is used to find errors listed below:
 - Interface errors, such as functions, which are unable to send or receive data to/from other software.
 - Incorrect functions that lead to undesired output when executed.
 - Missing functions and erroneous data structures.
 - Erroneous databases, which lead to incorrect outputs when software uses the data present in these databases for processing.
 - Incorrect conditions due to which the functions produce incorrect outputs when they are executed.
 - Termination errors, such as certain conditions due to which function enters a loop that forces it to execute indefinitely.
 17. The various methods of black box testing are equivalence class partitioning, boundary value analysis, orthogonal array testing, and cause-effect graphing. In **equivalence class partitioning** the test inputs are classified into equivalence classes such that one input checks (validates) all the input values in that class. In **boundary value analysis** the boundary values of the equivalence classes are considered and tested. In **orthogonal array testing** faults in the logic of the software component are considered and tested. In **cause-effect graphing**, cause-effect graphs are used to design test cases, which provides all the possible combinations of inputs to the program.
 18. Object-oriented programs can be tested at four levels the algorithmic level, class level, cluster level, and system level. At the **algorithmic level**, individual methods are tested in isolation. Here conventional testing techniques can be applied without much change. At the **class level**, the objective is to verify the integrity of a class by testing it as an individual entity. The **cluster level** is concerned about the integration of classes. The focus is on the synchronisation of different concurrent components as well as interclass method invocations. At the **system level**, interactions among clusters are tested.
 19.
 - Each test case should be uniquely identified and explicitly associated with the class to be tested.
 - The purpose of the test should be stated clearly.
 - A list of testing steps should be developed for each test and should contain the following:
 - A list of specified states for the object that is to be tested.
 - A list of messages and operations, which will be exercised as a consequence of the test.

- A list of exceptions, which may occur as the object is tested.
 - A list of external conditions (changes in the environment external to the software) that must exist in order to properly conduct the test.
 - Supplementary information that aids in understanding or implementing the test.
20. The methods used for performing object-oriented testing include state-based testing, fault-based testing, scenario-based testing, and unified modelling-based testing.
- **State-based testing:** State-based testing is used to verify whether the methods (a procedure that is executed by an object) of a class are interacting properly with each other or not. This testing seeks to exercise the transitions among the states based upon the identified inputs.
 - **Fault-based testing:** In fault-based testing, test cases are developed to determine a set of plausible faults. Here, the focus is on falsification. In this testing, tester does not focus on a particular coverage of a program or its specification, but on concrete faults that should be detected.
 - **Scenario-based testing:** Scenario-based testing is used to detect errors that are caused due to incorrect specifications and improper interactions among various segments of the software.
 - **Unified modelling language-based testing:** Unified modelling language (UML) is a semi-formal modelling language that is commonly used in object-oriented software development. It includes class diagrams, activity diagrams, sequence diagrams, collaboration diagrams, and state diagrams.

NOTES

5.10 QUESTIONS AND EXERCISES

I. Fill in the Blanks

1. The purpose of software testing is to find bugs, _____, _____, and _____ in the software.
2. A document that describes the objectives, scope, method, and purpose of software testing is known as _____.
3. Two methods that enable users to test the software are _____ and _____.
4. _____ testing is used to check the internal structure of the program.

II. Multiple Choice Questions

1. Which of the following is a type of software testing strategy?
(a) Model based (b) Process oriented (c) Dynamic (d) All
2. _____ is defined as the difference between the outputs produced by the software and the output desired by the user (expected output).
(a) Error (b) Failure (c) Faults (d) None of these
3. Which of the following is a type of control structure testing?
(a) Loop testing (b) Data flow testing
(c) Both (a) and (b) (d) None
4. _____ is an object-oriented testing method.
(a) Scenario-based testing (b) Acceptance testing
(c) Test plan (d) All

NOTES

III. State Whether True or False

1. Software testing activities need not be planned before testing commences.
2. While performing testing, there is no need to include test cases for invalid and unexpected conditions.
3. Testability is the ease with which the program is tested.
4. Set of all independent paths present in the program is known as basis set.

IV. Descriptive Questions

1. Can software execute properly if it is not tested? Explain.
2. Explain the guidelines that are followed to make testing effective and efficient.
3. Describe unit testing along with the procedure of performing it.
4. Is it beneficial to allow users to test the software before finally accepting it? If yes, why and explain the testing through which the user test the software.
5. What is white box testing and black box testing? Explain the differences between them.
6. What is the difference between equivalence class partitioning and boundary value analysis?
7. Write a short note on
 - (a) Basis path testing
 - (b) Control structure testing
 - (c) Mutation testing

5.11 FURTHER READING

1. Software Engineering – A Practitioner’s Approach– *Roger Pressman*
2. An Integrated Approach to Software Engineering– *Pankaj Jalote*
3. Software Engineering – *Ian Sommerville*

PUNJAB TECHNICAL UNIVERSITY

LADOWALI ROAD, JALANDHAR



INTERNAL ASSIGNMENT

TOTAL MARKS: 25

NOTE: Attempt any 5 questions
All questions carry 5 Marks.

- Q. 1. What are the different categories software can be classified into?
- Q. 2. Explain the waterfall process model.
- Q. 3. Explain the following components of software cost estimation:
(a) problem-based estimation; (b) process-based estimation
- Q. 4. Explain the project planning process.
- Q. 5. What is the difference between structured analysis and object-oriented analysis? Describe the concepts used in both of them.
- Q. 6. Write short notes on the following:
(a) SADT (b) Data dictionary
(c) Technical feasibility (d) Functional requirements
- Q. 7. Enumerate the various sections of Software Design Documentation (SDD).
- Q. 8. Outline the User Interface Design Process.
- Q. 9. What are the essential principles to be followed for software testing?
- Q.10. Briefly state the advantages and disadvantages of Acceptance Testing.



PUNJAB TECHNICAL UNIVERSITY

LADOWALI ROAD, JALANDHAR

ASSIGNMENT SHEET

(To be attached with each Assignment)

Full Name of Student: _____
(First Name) (Last Name)

Registration Number:

--	--	--	--	--	--	--	--	--	--	--	--

Course: _____ Sem.: _____ Subject of Assignment: _____

Date of Submission of Assignment:

--	--	--

(Question Response Record-To be completed by student)

S.No.	Question Number Responded	Pages ____ - ____ of Assignment	Marks
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

Total Marks: _____/25

Remarks by Evaluator: _____

Note: Please ensure that your Correct Registration Number is mentioned on the Assignment Sheet.

Name of the Evaluator

Signature of the Student

Signature of the Evaluator

Date: _____

Date: _____

