



Software Engineering

Er Pooja Yadav
[Email address]

Course Design Committee

Dr. Ashutosh Gupta **Chairman**
Director (In-charge)
School of Computer and Information Science, UPRTOU Allahabad

Prof. R. S. Yadav **Member**
Department of Computer Science and Engineering
MNNIT Allahabad

Ms. Marisha **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU Allahabad

Mr. Manoj Kumar Balwant **Member**
Assistant Professor (computer science)
School of Sciences, UPRTOU Allahabad

Course Preparation Committee

Dr. Pooja Yadav **Author**
Assistant Professor, Dept. of CS & IT
MJP Rohilkhand University, Bareilly (UP)

Dr. Ashutosh Gupta **Editor**
Associate Professor, Dept. of CS & IT
MJP Rohilkhand University, Bareilly (UP)

Mr. Manoj Kumar Balwant **Coordinator**
Assistant Professor (computer science)
School of Sciences, UPRTOU Allahabad

Software Engineering

FIRST - BLOCK

BLOCK

1

UNIT 1 SOFTWARE ENGINEERING FUNDAMENTALS

UNIT 2 SOFTWARE PROCESS

UNIT 3 PROJECT MANAGEMENT CONCEPT

BLOCK INTRODUCTION

In this section we discuss the overview of this block's content. This block consists of the following units:

Unit 1 Software Engineering Fundamentals

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Unit 2 Software Process

Software development life cycle (SDLC) models describe phases of the software cycle and the order in which these phases are executed. Each phase produces deliverables required by the next phase in the life cycle. The software development models are the various processes or methodologies that are being selected for the development of the project depending on the project's aims and goals. There are many development life cycle models that have been developed in order to achieve different required objectives. The models specify the various stages of the process and the order in which they are carried out. The SDLC aims to produce high quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

Unit 3 Project Management Concept

A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.

UNIT-1 SOFTWARE ENGINEERING FUNDAMENTALS

1.0 Introduction

1.1 Objective

1.2 Definition Of Software Engineering

1.3 Components Of Computer

1.4 Software Characteristics

1.5 Software Applications

1.6 Summary

1.7 Exercise

1.0 INTRODUCTION

Software Engineering is the discipline that aims to provide methods and procedures for developing software systems. Software engineering focuses on the process with the aim that the quality of product developed using a process is influenced mainly by the process. It is the application of science and mathematics by which the capabilities of computer equipment are made useful to humans via computer programs, procedure and associated documentation

1.1 OBJECTIVE

Objectives of this unit are:

- a) to apply knowledge of modules ,tools, procedure ,methods, paradigms.
- b) to design and conduct experiments, as well as to analyse and interpret data.
- c) to identify, formulate, and solve engineering problems.
- d) to communicate effectively.
- e) the broad education necessary to understand the impact of engineering solutions in a global, economic, environmental, and societal context.
- f) to use the techniques, skills, and modern engineering tools necessary for engineering practice.

1.2 DEFINITION OF SOFTWARE ENGINEERING

Software engineering is essentially a set of steps that comprises of process, methods and tools. It is defined not as a branch of engineering but rather a discipline whose aim is the production of quality software that satisfies the user's need and is delivered on time and within budget.

It can also be defined as –

“A systemization of the process of the software development in order to ensure the best solution in the most economical way”.

1.2.1 Elements of Software Engineering: The following are the major elements of software engineering:

- a. Methods
- b. Procedures

- c. Tools
- d. Paradigms

Methods: A method is a procedure for producing some result. It is sometimes also referred to as a technique. Methods generally demand some formal notation and processes.

Procedure: A procedure is a mechanism that combines tools and/or methods to produce a particular product. Algorithms are example of procedures.

Tools: Tools are the automated system that increases accuracy, efficiency, productivity, or equality of the end product.

Paradigm: It refers to a particular approach for building software such as the object oriented paradigm.

1.2.2 Evolution of software engineering

From its beginnings in the 1940s, writing software has evolved into a profession concerned with how best to maximize the quality of software and of how to create it. Quality refers to its stability, speed, usability, testability, readability, size, cost, security, and number of flaws or "bugs", as well as to less measurable qualities like elegance, conciseness, and customer satisfaction, among many other attributes. How best to create high quality software is a separate and controversial problem covering software design principles, so-called "best practices" for writing code, as well as broader management issues such as optimal team size, process, how best to deliver software on time and as quickly as possible, work-place "culture," hiring practices, and so forth. All this falls under the broad rubric of software engineering.

1945 to 1965: The Origins

The term *software engineering* first appeared in the late 1950s and early 1960s. Programmers have always known about civil, electrical, and computer engineering and debated what engineering might mean for software.

The NATO Science Committee sponsored two conferences on software engineering in 1968 and 1969, which gave the field its initial boost. Many believe these conferences marked the official start of the profession of *software engineering*.

1965 to 1985: The Software Crisis

Software engineering was spurred by the so-called *software crisis* of the 1960s, 1970s, and 1980s, which identified many of the problems of software development. Many software projects ran over budget and schedule. Some projects caused property damage.

A few projects caused loss of life. The software crisis was originally defined in terms of productivity, but evolved to emphasize quality. Some used the term *software crisis* to refer to their inability to hire enough qualified programmers.

- **Cost and Budget Overruns:** The OS/360 operating system was a classic example. This decade-long project from the 1960s eventually produced one of the most complex software systems at the time. OS/360 was one of the first large (1000 programmer's software projects). Fred Brooks claims in *The Mythical Man Month* that he made a multi-million dollar mistake of not developing a coherent architecture before starting development.
- **Property Damage:** Software defects can cause property damage. Poor software security allows hackers to steal identities, costing time, money, and reputations.
- **Life and Death:** Software defects can kill. Some embedded systems used in radiotherapy machines failed so catastrophically that they administered lethal doses of radiation to patients. The most famous of these failures is the *Therac-25* incident.

Peter G. Neumann has kept a contemporary list of software problems and disasters. The software crisis has been fading from view, because it is psychologically extremely difficult to remain in crisis mode for a protracted period (more than 20 years). Nevertheless, software - especially real-time embedded software - remains risky and is pervasive, and it is crucial not to give in to complacency. Over the last 10–15 years Michael A. Jackson has written extensively about the nature of software engineering, has identified the main source of its difficulties as lack of specialization, and has suggested that his problem frames provide the basis for a "normal practice" of software engineering, a prerequisite if software engineering is to become an engineering science.

1985 to 1989: No Silver Bullet

For decades, solving the software crisis was paramount to researchers and companies producing software tools. The cost of owning and maintaining software in the 1980s was twice as expensive as developing the software.

- During the 1990s, the cost of ownership and maintenance increased by 30% over the 1980s.
- In 1995, statistics showed that half of surveyed development projects were operational, but were not considered successful.
- The average software project overshoots its schedule by half.

- Three-quarters of all large software products delivered to the customer are failures that are either not used at all, or do not meet the customer's requirements.

1990 to 1999: Prominence of the Internet

The rise of the Internet led to very rapid growth in the demand for international information display/e-mail systems on the World Wide Web.

Programmers were required to handle illustrations, maps, photographs, and other images, plus simple animation, at a rate never before seen, with few well-known methods to optimize image display/storage (such as the use of thumbnail images).

The growth of browser usage, running on the HTML language, changed the way in which information-display and retrieval was organized. The widespread network connections led to the growth and prevention of international computer viruses on MS Windows computers, and the vast proliferation of spam e-mail became a major design issue in e-mail systems, flooding communication channels and requiring semi-automated pre-screening. Keyword-search systems evolved into web-based search engines, and many software systems had to be re-designed, for international searching, depending on search engine optimization (SEO) techniques. Human natural-language translation systems were needed to attempt to translate the information flow in multiple foreign languages, with many software systems being designed for multi-language usage, based on design concepts from human translators. Typical computer-user bases went from hundreds, or thousands of users, to, often, many-millions of international users.

2000 to Present: Lightweight Methodologies

With the expanding demand for software in many smaller organizations, the need for inexpensive software solutions led to the growth of simpler, faster methodologies that developed running software, from requirements to deployment, quicker & easier. The use of rapid-prototyping evolved to entire *lightweight methodologies*, such as Extreme Programming (XP), which attempted to simplify many areas of software engineering, including requirements gathering and reliability testing for the growing, vast number of small software systems. Very large software systems still used heavily-documented methodologies, with many volumes in the documentation set; however, smaller systems had a simpler, faster alternative approach to managing the development and maintenance of software calculations and algorithms, information storage/retrieval and display.

1.2.3 Current Trends in Software Engineering

Software engineering is a young discipline, and is still developing. The directions in which software engineering is developing include:

Aspects

Aspects help software engineer's deal with quality attributes by providing tools to add or remove boilerplate code [**boilerplate code** refers to sections of **code** that have to be included in many places with little or no alteration.] from many areas in the source code. Aspects describe how all objects or functions should behave in particular circumstances. For example, aspects can add debugging, logging, or locking control into all objects of particular types. Researchers are currently working to understand how to use aspects to design general-purpose code. Related concepts include generative programming and templates.

Agile

Agile software development guides software development projects that evolve rapidly with changing expectations and competitive markets. Groups of this method believe that heavy, document-driven processes (like TickIT, CMM and ISO 9000) are fading in importance. Some people believe that companies and agencies export many of the jobs that can be guided by heavy-weight processes. Related concepts include extreme programming, scrum, and lean software development.

Experimental

Experimental software engineering is a branch interested in devising experiments on software, in collecting data from the experiments, and in devising laws and theories from this data. Groups of this method advocate that the nature of software is such that we can advance the knowledge on software through experiments only.

Model-driven

Model driven design develops textual and graphical models as primary design artifacts. Development tools are available that use model transformation and code generation to generate well-organized code fragments that serve as a basis for producing complete applications.

Software product lines

Software product lines are a systematic way to produce *families* of software systems, instead of creating a succession of completely individual products. This method emphasizes extensive, systematic, formal code reuse, to try to industrialize the software development process.

Check Your Progress 1:

When you know programming, what is the need to learn software engineering concepts?

1.3 COMPONENTS OF COMPUTER

There are three basic components of Computer System. These are hardware, software and user.

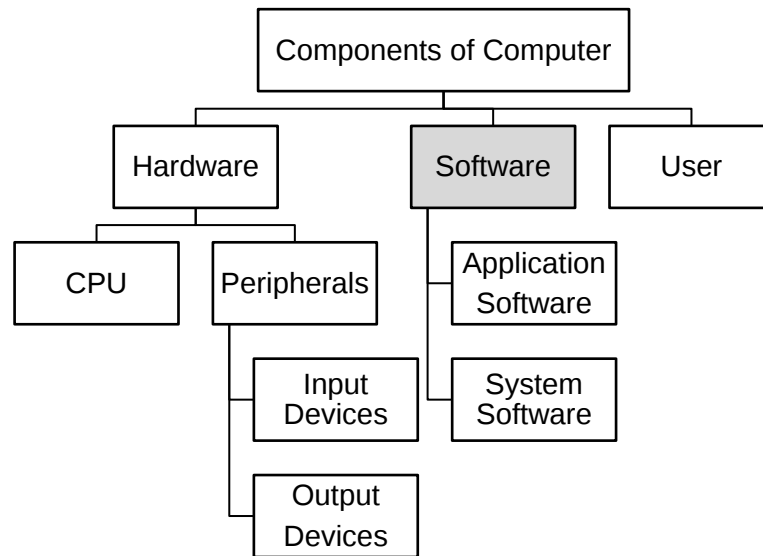


Figure 1.1 Components of computer

1.3.1 Hardware: Hardware is the tangible unit of the computer system. Hardware refers to all the physical components of a computer. It includes all input devices, processing devices, storage devices, and output devices. The keyboard, mouse, motherboard, monitor, hard disk, cables, and printer are all examples of hardware. You use hardware to provide input to a computer and also to get the desired output. Hardware needs software for the controlling itself. There are several parts of the hardware.

1.3.2 Software: Software is

- (1) Instructions (computer programs) that when executed provide desired function and performance,
- (2) Data structures that enable the programs to adequately manipulate information, and
- (3) Documents that describe the operation and use of the programs.

It is the essential component of computer system. Software gives "intelligence" to the computer. Software is a collection of program which drives the hardware in solving a problem. Software is kept on a secondary storage. Software's are classified into 2 categories: -

(a) *System Software*: - It includes the computer programs that run a computer system itself or that assist the computer in running application program. They control and support computer system. End user never uses system software directly.

(b) *Application Software*: - Application software is the main program for various applications written by programmers under an organization to solve any particular problem. All the software, which users actually use, is application software. For example- MS Word, excel, PowerPoint, Tally etc. Application software can be product based or project based:

a. *Products*: - These software are developed by software developers then launched in the market for the end users e.g. MS office, Tally, Photoshop etc.

b. *Project*: - (Custom software) these software are developed by software development companies on the demand of any client. They cannot be purchased from open market. They are the property of the specific organization e.g. banking software of bank, railway reservation software, billing software of any agency etc.

1.3.3 User: - User is the essential component of the computer. Anyone who is using the computer is computer user. Computer user can be manager, student, teacher, administrator or any lay person.

Check Your Progress 2.

Can you differentiate computer software and computer program?

1.4 SOFTWARE CHARACTERISTICS

To gain an understanding of, it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and bread boarded prototype evolve into a physical product.

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are non-existent for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product" but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. Software doesn't "wear out."

The relationship indicates that hardware exhibits relatively high failure rates early in its life; defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3. Most software continues to be custom built.

In the software world, it is something that has only begun to be achieved on a broad scale. A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. Today, we have extended our view of reuse to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts.

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

Operational

This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

Transitional

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

This aspect briefs about how well software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

1.5 SOFTWARE APPLICATION

Software engineering is the application of engineering to the design, development, implementation and maintenance of software in a systematic method. Software's are being developed in almost every area of life for automation. Software may be applied in any situation for which a pre-specified set of procedural steps has been defined.

The following software areas indicate the breadth of potential applications:

■ System software.

System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other systems applications (e.g., operating system

components, drivers, telecommunications processors) process largely indeterminate data. In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Real-time software.

Software that monitors/analyses/controls real-world events as they occur is called real time. Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

Business software.

Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision making. In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., point-of-sale transaction processing).

Engineering and scientific software.

Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software.

Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide

significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

■ **Personal computer software.**

The personal computer software market has burgeoned over the past two decades. Word processing, spread-sheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

■ **Web-based software.**

The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g. hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.

■ **Artificial intelligence software.**

Artificial intelligence (AI) software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

Here is the list of some application of software's in some areas:

- Business Process Re-Engineering
- Communication Networks
- Computer Graphics
- Cooperative Work Support
- e-Commerce
- Education
- Training
- Embedded Systems Programming
- m-Commerce
- Medical Informatics
- Mobile and Wireless Computing
- Multimedia Systems
- Parallel and Distributed Systems
- Real Time Systems
- Web-based Simulation

- Workflow Modelling
- others

Check Your Progress 3.

What are the challenges in software?

1.6 SUMMARY

This section covered about software, its types, characteristics and application area of software. And also explain software engineering, its evaluation, and current trends of this field.

1.7 EXERCISE

- 1) Define the term software and software engineering. What are the objectives of software engineering?
- 2) What is software engineering? How is it different from other traditional engineering branches?
- 3) Define the types of software. What are the characteristics of software?
- 4) What are the attributes of good software?
- 5) Write some applications of software.

UNIT-2 SOFTWARE PROCESS

2.0 Introduction

2.1 Objective

2.2 Definition of Software Process

2.3 Software Process Models

2.3.1 Waterfall Model

2.3.2 Prototype Model

2.3.3 Spiral Model

2.3.4 Incremental Model

2.4 Concurrent Development Model

2.5 Summary

2.6 Exercise

2.0 INTRODUCTION

The development lifecycle of software comprises of four major stages namely Requirement Elicitation, Designing, Coding and Testing. A software process model is the basic framework which gives a workflow from one stage to the next. This workflow is a guideline for successful planning, organization and final execution of the software project. Generally we have many different techniques and methods used to software development life cycle. Project and most real world models are customized adaptations of the generic models while each is designed for a specific purpose or reason, most have similar goals and share many common tasks.

Software processes performed during software Development and evolution are becoming rather complex and resource intensive. They involve people who execute actions with the primary goal to create quality software in accordance with the previously set user requirements. Only structured, carefully guided and documented software processes can lead to the stated goal. Constant monitoring and improvement of Software processes is therefore of a significant interest for organizational performing software development and maintenance. In order to improve the process an objective description and evolution of the existing process is needed.

In contrast to software life cycle models, software process models often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing. Software process networks can be viewed as representing multiple interconnected task chains. Task chains represent a non-linear sequence of actions that structure and transform available computational objects (resources) into intermediate or finished products. Non-linearity implies that the sequence of actions may be non-deterministic, iterative, accommodate multiple/parallel alternatives, as well as partially ordered to account for incremental progress. Task actions in turn can be viewed a non-linear sequences of primitive actions which denote atomic units of computing work, such as a user's selection of a command or menu entry using a mouse or keyboard.

2.1 OBJECTIVE

Objectives of this unit are:

- a) To introduce the concept of software process and software process models.
- b) To describe a number of different process models and when they may be used.
- c) To describe outline process models for requirements engineering, software development, testing and evolution.
- d) To describe the pros and cons of each model

2.2 SOFTWARE DEVELOPMENT LIFECYCLE

SDLC, Software Development Life Cycle, is a process used by software industry to design, develop and test high quality software. The SDLC aims to produce high quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates. The software development life cycle (SDLC) is a framework defining tasks performed at each step in the software development process. ISO/IEC 12207 is an international standard for software life-cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software. SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

A typical Software Development life cycle consists of the following stages as shown in figure 2.1:

Stage 1: *Planning and Requirement Analysis*: Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational, and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

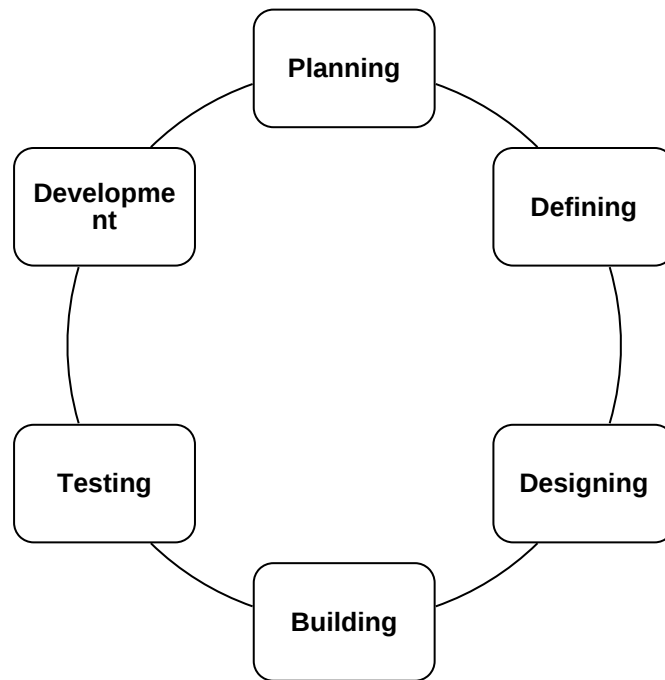


Figure 2.1 : Stages of SDLC

Stage 2: *Defining Requirements*: Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through ‘SRS’ – Software Requirement Specification document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: *Designing the product architecture*: SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification. This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity , budget and time constraints , the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutes of the details in DDS.

Stage 4: *Building or Developing the Product*: In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this

stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers have to follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers etc. are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java, and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product: This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However this stage refers to the testing only stage of the product where products defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance: Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometime product deployment happens in stages as per the organizations' business strategy. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

Check Your Progress 1.

What are the fundamental activities of a software process?

2.3 SOFTWARE PROCESS MODELS

There is various software process models defined and designed which are followed during software development process. These models are also referred as "Software Development Life Cycle Models". Each process model follows a Series of steps unique to its type, in order to ensure success in process of software development.

Following are the most important and popular SDLC models followed in the industry:

- i. **Waterfall Process Model:** The Classical Life Cycle or the Waterfall Process Model was the first process model to present a sequential framework, describing basic stages that are mandatory for a successful software development model. It formed the basis for

most software development standards and consists of the following phases: Requirements elicitation, Designing, Implementation and Testing.

- ii. **Prototype Model:** In Prototype Model, the user is given a “look and feel” of the system using a prototype. The prototype for the system to be developed is built, tested and reworked as necessary. Prototype process model is suitable for dynamic environment where requirements change rapidly. The process begins with gathering main functional requirements; this is followed by a quick design leading to the development of a prototype. The prototype is then evaluated by users and customers. Developers rework on the prototype until the customer and users are satisfied.
- iii. **Incremental Development Model:** In incremental development process, customers identify, in outlined the services to be provided by the system. They identify which of the services are most important and which are least important to them. A number of delivery increments are then defined which each increment providing a subset of functional requirements. The highest priority functional requirements are delivered first.
- iv. **Spiral Model:** In Spiral model, instead of presenting a sequence of activities with some backtracking from one activity to the other, the process model followed a spiral organization of activities. It combines characteristics of both prototype and waterfall process model. The model is divided into some task regions, which are as follows: Customer Communication, Planning, Risk Analysis, and Engineering, Construction and release and Customer evaluation. The distinctive feature of this model is that each stage is controlled by a specific risk management criteria ensuring decision making using critical factors.
- v. **Rapid Application Development Model:** The RAD model is an adaptation of the classical model for achieving rapid development using component based construction. If requirements are well understood with a well constrained project scope, the RAD process enables delivery of the fully function system. The model is considered to be incremental development model and that have emphasis on short development cycle.
- vi. **Rational Unified Process Model (RUP):** The RUP provides dynamic, static and practice perspectives of a product. The RUP provides each team member with the guidelines, templates and tool mentors necessary for the entire team to take full advantage of the best practices. The software lifecycle is broken into cycles, each cycle working on a new generation of the product.

- vii. **The V-Model:** The V-Model is an extension to the Waterfall Model in that it does not follow a sequential mode of execution rather it bends upward after the coding phase to form V shape.
- viii. **Concurrent Engineering Model:** The concurrent development model sometimes called concurrent engineering model can be represented schematically as a series of frame work activities, software engineering action and task, and their associated status. Provide a schematic representation of one software engineering task with in the modelling activities for the concurrent process model. The activity-modelling may be in any one of the states noted at any given time. Similarly, other activities or task can be represented in an analogous manner. All activities exist concurrently but reside in different states .its first iteration and exist in the waiting changes state. The modelling activities which existed in none state while initial communication was completed, now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the modeling activities moves from the under development states into the awaiting changes states. The concurrent process model defines a series of events that will trigger transition from state to state for each of the software engineering activities, actions, or tasks.
- ix. **Confident Software Development Process Model:** The Confident process model which we have proposed has seven phases, namely; Feasibility study/Requirement, Requirement Based Analysis, Logical Design, Confident Code, Logical Testing, Implementation & Deployment, and Maintenance. It is a flexible model not restricting the developers enabling them to move both Front and back from any given stage to any other stage during its lifecycle. Each phase is further divided into sub phases, each specifying a criterion which has to be met to move to the next phase.
- x. **The Formal model:** The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous. When formal methods are used during development, they provide a mechanism for eliminating many of the problem that are difficult to overcome using other software engineering cardiogram

2.3.1 WATERFALL MODEL

The Waterfall Model was first Process Model to be introduced. It is also referred to as a linear-sequential life cycle model. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

Waterfall model is the earliest SDLC approach that was used for software development .The waterfall Model illustrates the software development process in a linear sequential flow; hence it is also referred to as a linear-sequential life cycle model. This means that any phase in the development process begins only if the previous phase is complete. In waterfall model phases do not overlap.

Waterfall Model design

Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially as shown in figure 2.2.

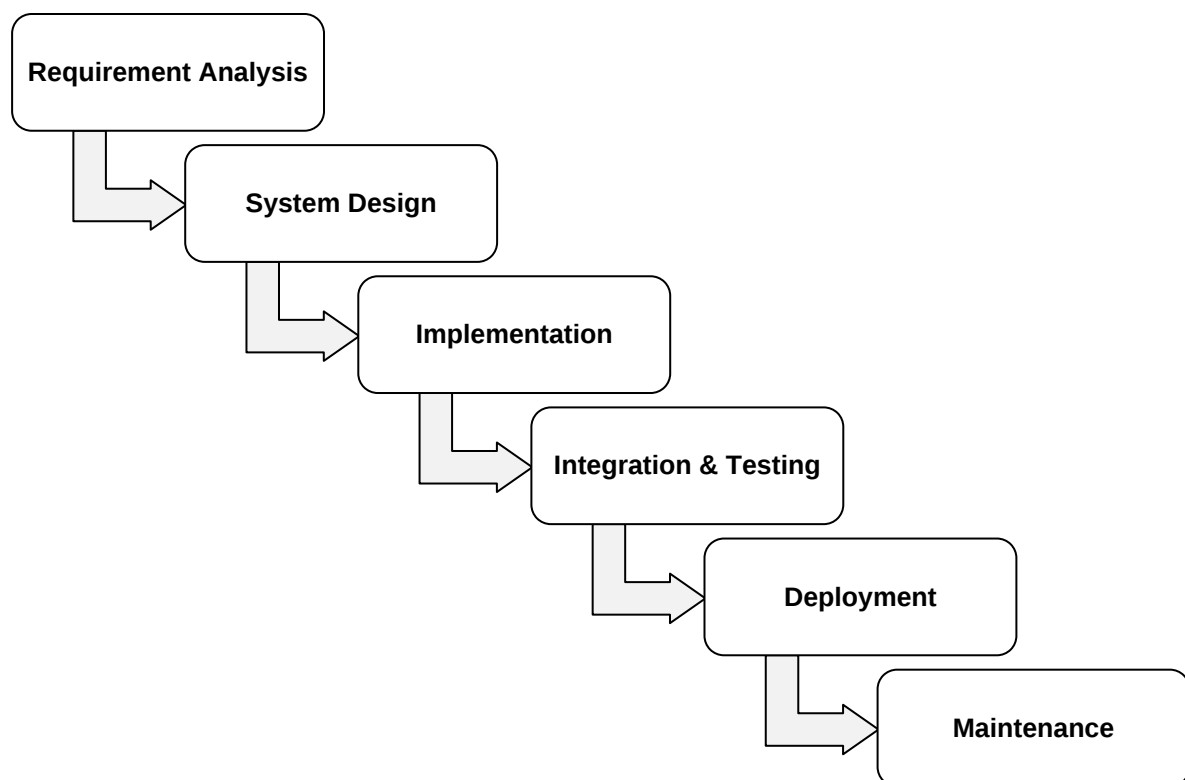


Figure 2.2: Stages of Waterfall Model

The sequential phases in Waterfall model are:

- **Requirement Gathering and analysis:** All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification doc.
- **System Design:** The requirement specifications from first phase are studied in this phase and system design is prepared. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture.
- **Implementation:** With inputs from system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality which is referred to as Unit Testing.
- **Integration and Testing:** All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system:** Once the functional and non-functional testing is done, the product is deployed in the customer environment or released into the market.
- **Maintenance:** There are some issues which come up in the client environment. To fix those issues patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model". In this model phases do not overlap.

Waterfall Model Application

Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors. Some situations where the use of Waterfall model is most appropriate are:

- Requirements are very well documented, clear and fixed.
- Product definition is stable.
- Technology is understood and is not dynamic.
- There are no ambiguous requirements.
- Ample resources with required expertise are available to support the product.
- The project is short.

Advantage of Waterfall Model:

- Simple and easy to understand.

- 🎬 Work well for smaller projects where requirements are very well understood.
- 🎬 Easy to manage due to rigidity of model. Each phase has its specific deliverables and review process.
- 🎬 Easy to arrange task.
- 🎬 Clearly defined stages.
- 🎬 Phases are processed and completed one at a time.
- 🎬 Process and results are well documented.

Disadvantage of Waterfall Model:

- 🎬 High amount of risk and uncertainty.
- 🎬 Poor model for long and on-going projects.
- 🎬 Not a good model for object-oriented projects and complex projects.
- 🎬 It is very difficult to measure progress within stages.
- 🎬 Cannot accommodate changing requirements. i.e. it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.
- 🎬 No working software is produced until late in the life cycle.
- 🎬 Not suitable for the projects where requirements are at a moderate to high risk of changing.

Check Your Progress 2.

Write out the reasons for the Failure of Water Fall Model?

2.3.2 PROTOTYPE MODEL

The Software Prototyping refers to building software application prototypes which display the functionality of the product under development but may not actually hold the exact logic of the original software.

Software prototyping is becoming very popular as a software development model, as it enables to understand customer requirements at an early stage of development. It helps get valuable feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.

What is Software Prototyping?

- 🎬 Prototype is a working model of software with some limited functionality.

- The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.
- Prototyping is used to allow the users evaluate developer proposals and try them out before implementation.
- It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Stepwise approach of Software prototype

Following is the stepwise approach to design a software prototype as depicted in figure 2.3:

- **Basic Requirement Identification:** This step involves understanding the very basics product requirements especially in terms of user interface. The more intricate details of the internal design and external aspects like performance and security can be ignored at this stage.
- **Developing the initial Prototype:** The initial Prototype is developed in this stage, where the very basic requirements are showcased and user interfaces are provided. These features may not exactly work in the same manner internally in the actual software developed and the workarounds are used to give the same look and feel to the customer in the prototype developed.
- **Review of the Prototype:** The prototype developed is then presented to the customer and the other important stakeholders in the project. The feedback is collected in an organized manner and used for further enhancements in the product under development.
- **Revise and enhance the Prototype:** The feedback and the review comments are discussed during this stage and some negotiations happen with the customer based on factors like, time and budget constraints and technical feasibility of actual implementation. The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until customer expectations are met.

Prototypes can have horizontal or vertical dimensions. Horizontal prototype displays the user interface for the product and gives a broader view of the entire system, without concentrating on internal functions. A vertical prototype on the other side is a detailed elaboration of a specific function or a sub system in the product.

The purpose of both horizontal and vertical prototype is different. Horizontal prototypes are used to get more information on the user interface level and the business requirements. It can even be presented in the sales demos to get business in the market. Vertical prototypes are

technical in nature and are used to get details of the exact functioning of the sub systems. For example, database requirements, interaction and data processing loads in a given sub system.

Software Prototyping Types

There are different types of software prototypes used in the industry. Following are the major software prototyping types used widely:

- **Throwaway/Rapid Prototyping:** Throwaway prototyping is also called as rapid or close ended prototyping. This type of prototyping uses very little efforts with minimum requirement analysis to build a prototype. Once the actual requirements are understood, the prototype is discarded and the actual system is developed with a much clear understanding of user requirements.
- **Evolutionary Prototyping:** Evolutionary prototyping also called as breadboard prototyping is based on building actual functional prototypes with minimal functionality in the beginning. The prototype developed forms the heart of the future prototypes on top of which the entire system is built. Using evolutionary prototyping only well understood requirements are included in the prototype and the requirements are added as and when they are understood.
- **Incremental Prototyping:** Incremental prototyping refers to building multiple functional prototypes of the various sub systems and then integrating all the available prototypes to form a complete system.
- **Extreme Prototyping:** Extreme prototyping is used in the web development domain. It consists of three sequential phases. First, a basic prototype with all the existing pages is presented in the html format. Then the data processing is simulated using a prototype services layer. Finally the services are implemented and integrated to the final prototype. This process is called Extreme Prototyping used to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the actual services.

Software Prototyping Application

Software Prototyping is most useful in development of systems having high level of user interactions such as online systems. Systems which need users to fill out forms or go through various screens before data is processed can use prototyping very effectively to give the exact look and feel even before the actual software is developed.

Software that involves too much of data processing and most of the functionality is internal with very little user interface does not usually benefit from prototyping. Prototype development could be an extra overhead in such projects and may need lot of extra efforts.

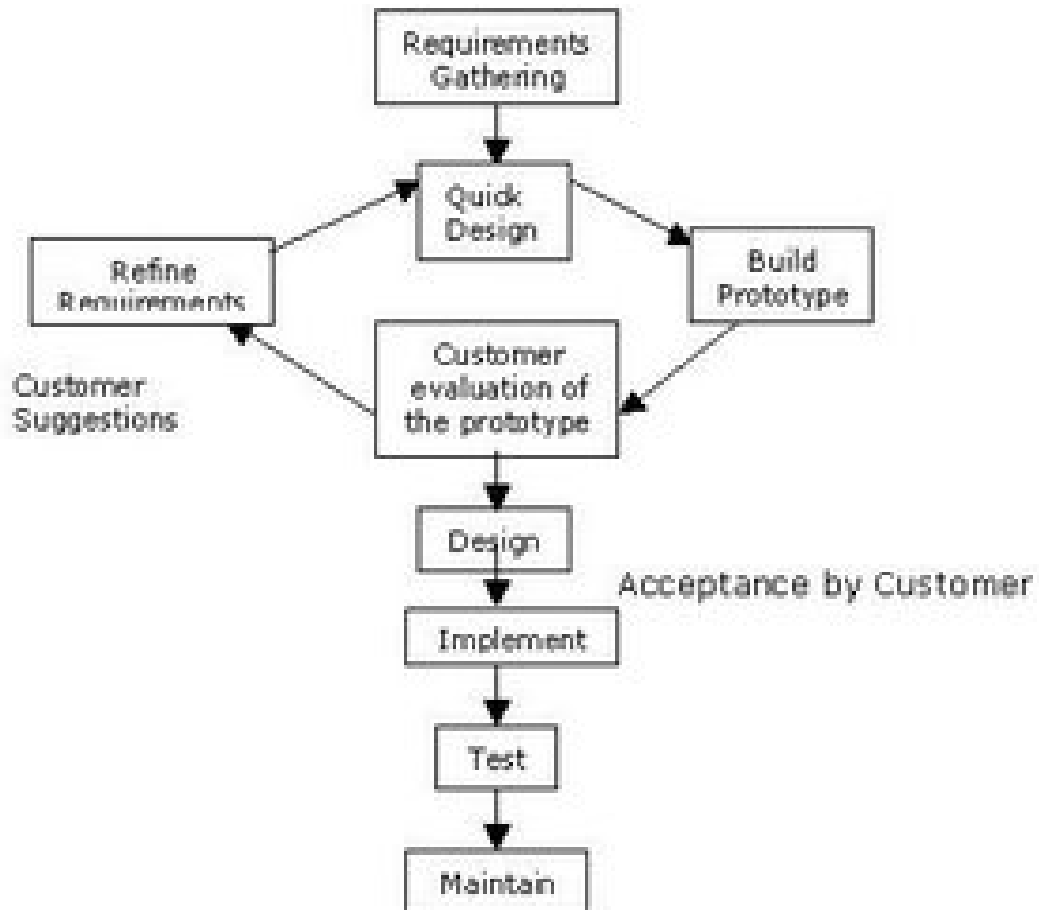


Figure 2.3: Prototype Model

Software prototyping is used in typical cases and the decision should be taken very carefully so that the efforts spent in building the prototype add considerable value to the final software developed. The model has its own pros and cons discussed as below.

Following table lists out the pros and cons of the Model:

Advantages of Prototype Model

- Increased user involvement in the product even before implementation
- Since a working model of the system is displayed, the users get a better understanding of the system being developed.
- Reduces time and cost as the defects can be detected much earlier.

- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily. Confusing or difficult functions can be identified

Disadvantages of Prototype Model

- Risk of insufficient requirement analysis owing to too much dependency on prototype
- Users may get confused in the prototypes and actual systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Developers may try to reuse the existing prototypes to build the actual system, even when it's not technically feasible
- The effort invested in building prototypes may be too much if not monitored properly

Check Your Progress 3.

What are the benefits of prototyping?

2.3.3 SPIRAL MODEL

The spiral model combines the idea of iterative development with the systematic, controlled aspects of the waterfall model.

Spiral model is a combination of iterative development process model and sequential linear development model i.e. waterfall model with very high emphasis on risk analysis. It allows for incremental releases of the product, or incremental refinement through each iteration around the spiral.

Spiral Model design

The spiral model has four phases. A software project repeatedly passes through these phases in iterations called Spirals.

- **Identification:** This phase starts with gathering the business requirements in the baseline spiral. In the subsequent spirals as the product matures, identification of system requirements, subsystem requirements and unit requirements are all done in this phase.
- This also includes understanding the system requirements by continuous communication between the customer and the system analyst. At the end of the spiral the product is deployed in the identified market.

- **Design:** Design phase starts with the conceptual design in the baseline spiral and involves architectural design, logical design of modules, physical product design and final design in the subsequent spirals.

- **Construct or Build:** Construct phase refers to production of the actual software product at every spiral. In the baseline spiral when the product is just thought of and the design is being developed a POC (Proof of Concept) is developed in this phase to get customer feedback.

Then in the subsequent spirals with higher clarity on requirements and design details a working model of the software called build is produced with a version number. These builds are sent to customer for feedback.

- **Evaluation and Risk Analysis:** Risk Analysis includes identifying, estimating, and monitoring technical feasibility and management risks, such as schedule slippage and cost overrun. After testing the build, at the end of first iteration, the customer evaluates the software and provides feedback.

Based on the customer evaluation, software development process enters into the next iteration and subsequently follows the linear approach to implement the feedback suggested by the customer. The process of iterations along the spiral continues throughout the life of the software.

Following, figure 2.4, is a diagrammatic representation of spiral model listing the activities in each phase:

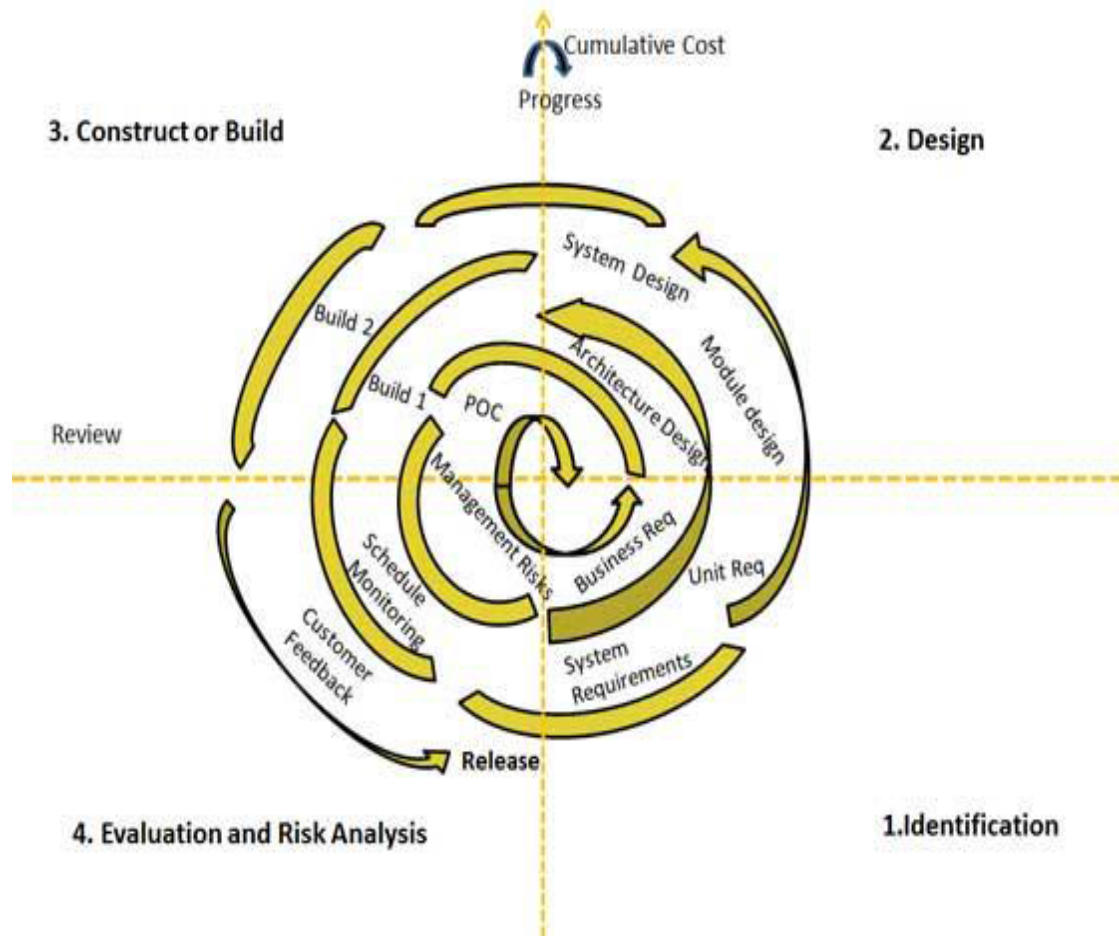


Figure 2.4: Stages of Spiral Model

Spiral Model Application

Spiral Model is very widely used in the software industry as it is in synch with the natural development process of any product i.e. learning with maturity and also involves minimum risk for the customer as well as the development firms. Following are the typical uses of Spiral model:

- For medium to high-risk projects.
- Long-term project commitment because of potential changes to economic priorities as the requirements change with time.
- Customer is not sure of their requirements.
- Requirements are complex and need evaluation to get clarity.
- New product line which should be released in phases to get enough customer feedback.
- Significant changes are expected in the product during the development cycle.

Advantages of Spiral Model

- Development can be divided into smaller parts and more risky parts can be developed earlier which helps better risk management.
- Users see the system early.
- Requirement can be captured more accurately
- Allows for extensive use of prototypes.
- Changing requirements can be accommodated

Disadvantages of Spiral Model

- Management is more complex.
- End of projects may not be known early.
- Not suitable for small and low risk projects and could be extensive for small projects.
- Process is complex.
- Spiral may go indefinitely.
- Large number of intermediate stages requires excessive documentation.

Check Your Progress 4.

When to use Spiral model?

2.3.4 INCREMENTAL MODEL

Incremental model are broken down into multiple standalone modules of software development cycle. These cycles are further divided into smaller and more manageable iterations as depicted in figure 2.5.

The incremental build model is a method of software development where the model is designed, implemented and tested incrementally (a little more is added each time) until the product is finished. It involves both development and maintenance. The product is defined as finished when it satisfies all of its requirements. This model combines the elements of the waterfall model with the iterative philosophy of prototyping.

The product is decomposed into a number of components, each of which are designed and built separately (termed as builds). Each component is delivered to the client when it is complete. This allows partial utilisation of product and avoids a long development time. It also creates a large initial capital outlay with the subsequent long wait avoided. This model of development also helps ease the traumatic effect of introducing completely new system all at once.

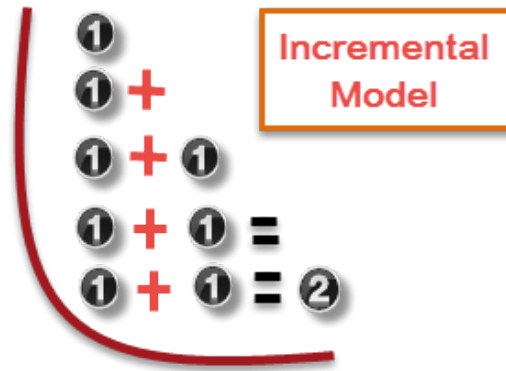


Figure 2.5: Basic Concept of Incremental model

Each iteration passes through the **requirements, design, coding and testing phases** as depicted in figure 2.6.

. And each subsequent release of the system adds function to the previous release until all designed functionality has been implemented.

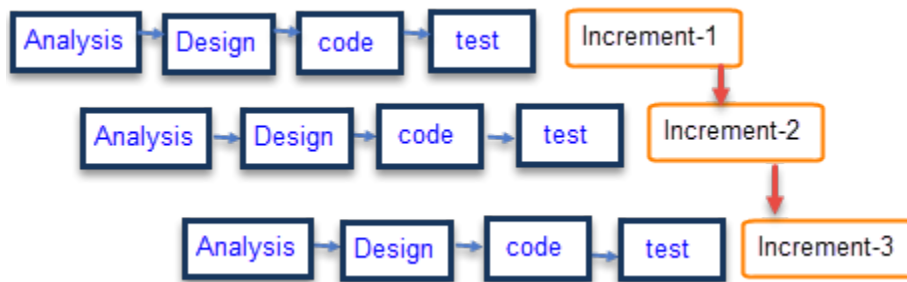


Figure 2.6: Stages of Incremental Model

The system is put into production when the first increment is delivered. The first increment is often a core product where the basic requirements are addressed, and supplementary features are added in the next increments. Once the core product is analyzed by the client, there is plan development for the next increment.

Characteristics of Incremental module includes

- System development is broken down into many mini development projects
- Partial systems are successively built to produce a final total system
- Highest priority requirement is tackled first
- Once the incremented portion is developed, requirements for that increment are frozen

Incremental Phases	Activities performed in incremental phases
--------------------	--

Requirement Analysis	Requirement and specification of the software are collected
Design	Some high-end function are designed during this stage
Code	Coding of software is done during this stage
Test	Once the system is deployed, it goes through the testing phase

When to use Incremental models?

- Requirements of the system are clearly understood
- When demand for early release of product arises
- When team resources are not very well skilled or trained
- When high-risk features and goals are involved
- Such model is more in use for web application and product based companies

Advantages of Incremental Model

- Software will be generated quickly during the software life cycle
- It is flexible and less expensive to change requirements and scope
- Thought the development stages changes can be done
- This model is less costly compared to others
- Customer can respond to each built
- Errors are easy to be identified

Disadvantages of Incremental Model

- It requires a good planning designing
- Problems might cause due to system architecture as such not all requirements collected up front for the entire software life cycle
- Each iteration phase is rigid and does not overlap each other
- Rectifying a problem in one unit requires correction in all the units and consumes a lot of time

Check Your Progress 5.

Explain in your own words a distinction between the iterative and incremental life cycle models.

2.4 CONCURRENT DEVELOPMENT MODEL

The concurrent development model is also called concurrent engineering. Project managers who track project status in terms of the major phases have no idea of the status of their projects. These are examples of trying to track extremely complex sets of activities using overly simple models. Note that although project is in the coding phase, there are personnel on the project involved in activities typically associated with many phases of development simultaneously. For example, personnel are writing requirements, designing, coding, testing, and integration testing. Software engineering process models by Humphrey and Kellner have shown the concurrency that exists for activities occurring during any one phase. Kellner's more recent work uses state charts to represent the concurrent relationship existent among activities associated with a specific event (e.g., a requirements change during late development), but fails to capture the richness of concurrency that exists across all software development and management activities in the project. Most software development process models are driven by time; the later it is, the later in the development process you are. A concurrent process model is driven by user needs, management decisions, and review results.

The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states. For example, the engineering activity defined for the spiral model is accomplished by invoking the following tasks: prototyping and/or analysis modelling, requirements specification, and design.

The activity—analysis—may be in any one of the states noted at any given time. Similarly, other activities can be represented in an analogous manner. All activities exist concurrently but reside in different states. For example, early in a project the customer communication activity has completed its first iteration and exists in the awaiting changes state. The analysis activity now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the analysis activity moves from the under development state into the awaiting changes state.

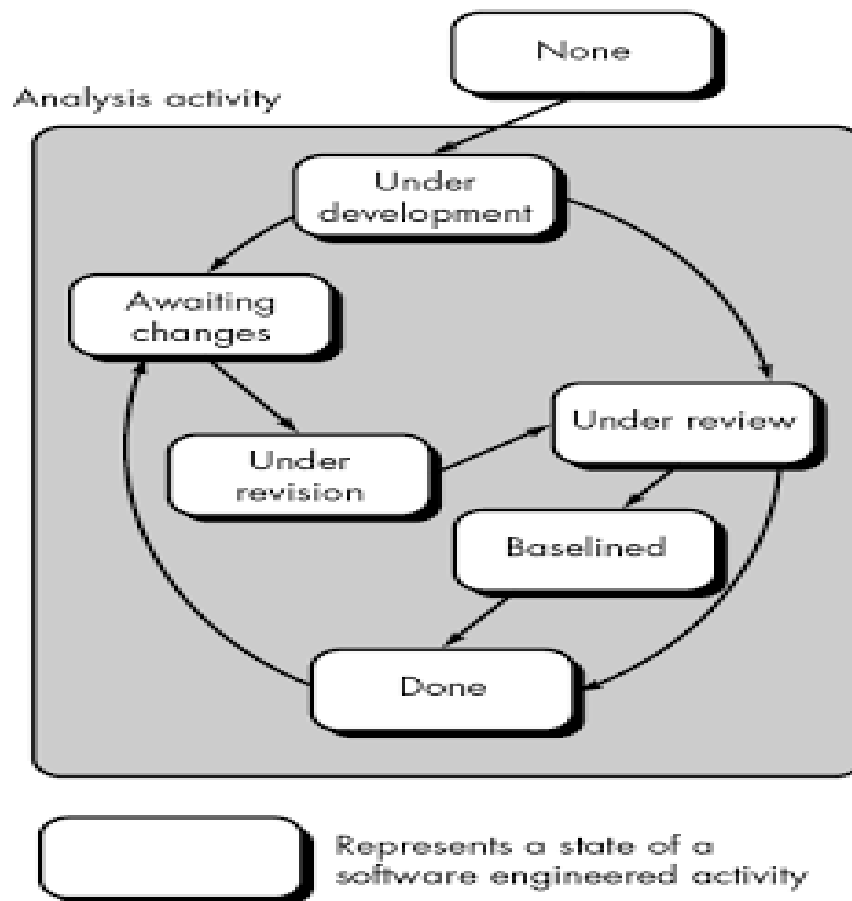


Figure2.7: A Concurrent Process Model

As shown in figure 2.7, the concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities. For example, during early stages of design, an inconsistency in the analysis model is uncovered. This generates the event analysis model correction which will trigger the analysis activity from the done state into the awaiting changes state.

The concurrent process model is often used as the paradigm for the development of client/server applications. A client/server system is composed of a set of functional components. When applied to client/server, the concurrent process model defines activities in two dimensions: a system dimension and a component dimension. System level issues are addressed using three activities: design, assembly, and use. The component dimension is addressed with two activities: design and realization.

Concurrency is achieved in two ways:

1. System and component activities occur simultaneously and can be modelled using the state-oriented approach described previously;

2. A typical client/server application is implemented with many components, each of which can be designed and realized concurrently.

In reality, the concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities to a sequence of events, it defines a network of activities. Each activity on the network exists simultaneously with other activities. Events generated within a given activity or at some other place in the activity network trigger transitions among the states of an activity.

Check Your Progress 6.

What is concurrency and how it is achieved in software?

2.5 SUMMARY

This was about the various SDLC models available and the scenarios in which these SDLC models are used. We have discussed all the popular SDLC models is used in the industry, Waterfall model is sequential type. Sequential means that the next phase can start only after the completion of first phase. Such models are suitable for projects with very clear product requirements and where the requirements will not change dynamically during the project completion.

Spiral models are more accommodative in terms of change and are suitable for projects where the requirements are not so well defined, or the market requirements change quite frequently. Software Prototyping is most useful in development of systems having high level of user interactions and Incremental model is more in use for web application.

2.6 EXERCISE

- 1) Explain the waterfall model. Explain its application area and drawback of waterfall model.
- 2) List four reasons why it is difficult to improve software process.
- 3) What is the advantage of using prototype software development model instead of waterfall model?
- 4) How does the risk factor affect the spiral model of software development?
- 5) Write some application of spiral model.
- 6) What is incremental model? Write some characteristics and advantages of it.
- 7) What are the different phases of traditional system development life cycle?

UNIT-3 PROJECT MANAGEMENT CONCEPTS

3.0 Introduction

3.1 Objective

3.2 Need Of Project Management

3.3 The Management Spectrum

3.3.1 The People

3.3.2 The Product

3.3.3 The Process

3.3.4 The Project

3.4 Summary

3.5 Exercise

3.0 INTRODUCTION

Software development is not a new stream in world business but there's very little experience in building software products. A project is well-defined task, which is a collection of several operations done in order to achieve a goal. The most important is that the technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one. All such business and environmental constraints bring risk in software development hence it is essential to manage software projects efficiently.

3.1 OBJECTIVE

The main objectives and principles behind good project management are as follows:

- a) Agree exactly what a project is meant to do and what it is meant to deliver.
- b) Agree the scope, timescales, cost and quality of a project.
- c) Maintain a schedule and project plan.
- d) Deliver the agreed outcomes of the project to the right scope, timescales, cost and quality.
- e) Provide communications, reports and progress updates throughout the lifecycle of the project.
- f) Manage risks, issues and dependencies.
- g) Manage policies, processes, tools, frameworks, techniques, people and relationships to a successful project outcome.

3.2 NEED OF PROJECT MANAGEMENT

Software project management is the important task of planning, directing, motivating, and coordinating a group of professionals to accomplish software development. Software project management uses many concepts from management in general, but it also has some concerns unique to software development. One such concern is project visibility. The lack of visibility of the software product during software development makes it hard to manage. In many other fields, it is easy to see progress or lack of progress. Many software projects get stalled at 90 percent complete. Ask any programmer if that bug that he or she found is the last bug in the software, and the answer will almost always be an emphatic yes. Many of the techniques in software management are aimed at overcoming this lack of visibility.

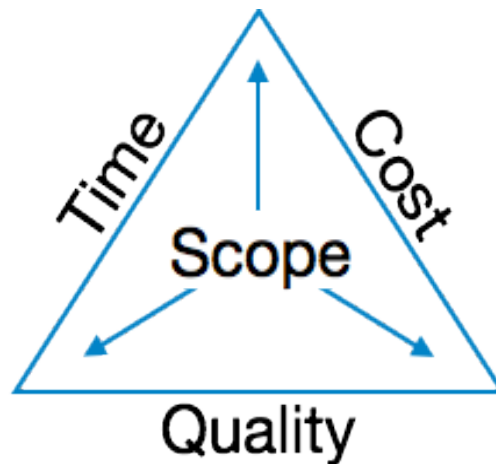


Figure 3.1: Triple constraints for software projects

It is an essential part of software organization to deliver quality product, keeping the cost within client's budget constrain and deliver the project as per scheduled. There are several factors, both internal and external, which may impact this triple constrain triangle as shown in figure 3.1. Any of three factors can severely impact the other two.

Therefore, software project management is essential to incorporate user requirements along with budget and time constraints.

Who does it? Everyone “manages” to some extent, but the scope of management activities varies with the person doing it. A software engineer manages her day-to-day activities, planning, monitoring, and controlling technical tasks. Project managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between the business and the software professionals.

Importance of Project Management: Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed.

A project plan is produced as management activities commence. The plan defines the process and tasks to be conducted, the people who will do the work, and the mechanisms for assessing risks, controlling change, and evaluating quality.

Steps of Project Management: Understand the four P's—people, product, process, and project. People must be organized to perform software work effectively. Communication with the customer must occur so that product scope and requirements are understood. A process must be selected that is appropriate for the people and the product. The project must be planned by estimating effort and calendar time to accomplish work tasks: defining work products,

establishing quality checkpoints, and establishing mechanisms to monitor and control work defined by the plan.

3.3 THE MANAGEMENT SPECTRUM

There are four P's of project management as shown in figure 3.2.

- The People
- The Product
- The Process
- The Project

The point to emphasize is that each of the P's is important and it is the synergy of all four working together that yields the successful management of software products. This also the time to remind students that it is customer for whom the product is being developed. Process framework activities are populated with tasks, milestones, work products, and quality assurance checkpoints regardless of the project size. To avoid project failure developers need react to warning signs and focus their attention on practices that are associated with good project management.

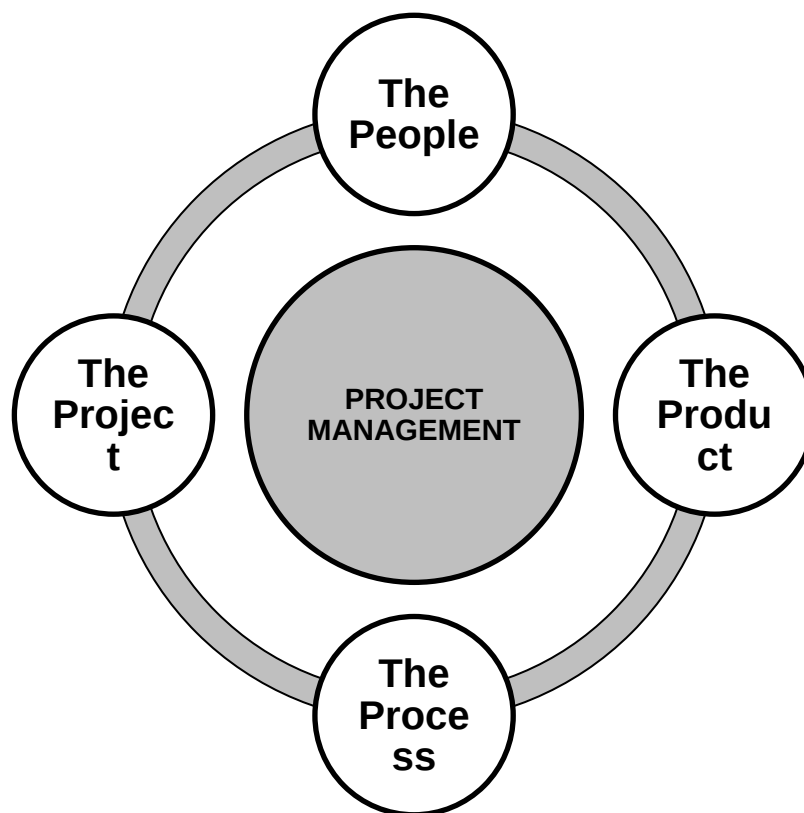


Figure 3.2: Management Spectrum

3.3.1 THE PEOPLE

Companies that manage their people wisely prosper in the long run. To be effective the project team must be organized in a way that maximizes each person's skills and abilities. Effective managers focus on problem solving and insist on high product quality. Software teams may be organized in many different ways. Two factors in selecting a team organizational model are desired level of communication among its members and difficulty level of the problems to be solved. Hierarchically organized teams can develop routine software applications without much communication among the team members. Teams having a more democratic style organization often develop novel applications more efficiently. It is important for students to understand that the larger the team, the greater the effort required to ensure effective communication and coordination of team member efforts.

Five categories of The People:

i. Stakeholders / Players

The software process (and every software project) is populated by players who can be categorized into one of five constituencies:

- *Senior managers* – define business issues that often have significant influence on the project
- *Project (technical) managers* – plan, motivate, organize, and control the practitioners who do the work
- *Practitioners* – deliver the technical skills that are necessary to engineer a product or application
- *Customers* – specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome
- *End users* – interact with the software once it is released for production use.

ii. Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills.

There is a *simple model of leadership* which includes-

- **Motivation.** The ability to encourage (by “push or pull”) technical people to produce to their best ability.
- **Organization.** The ability to mould existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.

- 🎬 **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

The **characteristics** that define an effective project manager emphasize four key traits:

- 🎬 Problem solving
- 🎬 Managerial identity
- 🎬 Achievement
- 🎬 Influence and team building

iii. **The Software Team**

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. However, the organization of the people directly involved in a new software project is within the project manager's purview.

There are **three types of generic team organizations**:

- Democratic decentralized (DD).** This software engineering team has no permanent leader. Rather, "task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks." Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.
- Controlled decentralized (CD).** This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.
- Controlled Centralized (CC).** Top-level problem solving and internal team coordination are managed by a team leader. Communication between the leader and team members is vertical.

There are **seven project factors** that should be considered when planning the structure of software engineering teams:

- The difficulty of the problem to be solved.
- The size of the resultant program(s) in lines of code or function points.
- The time that the team will stay together (team lifetime).
- The degree to which the problem can be modularized.

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

iv. **Coordination and Communication Issues**

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

Check Your Progress 1.

What does software project manager do?

3.3.2 THE PRODUCT

The first project management activity is the determination of software scope. This is essential to ensure the product developed is the product requested by the customer. It is sometimes helpful to remind students that unless developers and customers agree on the scope of the project there is no way to determine when it ends (or when they will get paid). Regardless of the process model followed, a problem must be decomposed along functional lines into smaller, more easily managed sub-problems.

The scope of the software development must be established and bounded:

- ▣ **Context** – How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?
- ▣ **Information objectives** – What customer-visible data objects are produced as output from the software? What data objects are required for input?
- ▣ **Function and performance** – What functions does the software perform to transform input data into output? Are there any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at both the managerial and technical levels.

Problem decomposition is also referred to as partitioning or problem elaboration. It sits at the core of software requirements analysis.

There are two major areas of problem decomposition: first, the functionality that must be delivered; second; the process that will be used to deliver it.

3.3.3 THE PROCESS

The generic phases that characterize the software process—definition, development, and support—are applicable to all software. The problem is to select the process model that is appropriate for the software to be engineered by a project team.

Once a process model is chosen, it needs to be populated with the minimum set of work tasks and work products. Avoid process overkill. It is important to remind students that framework activities are applied on every project, no matter how small. Work tasks may vary, but not the common process framework. Process decomposition can occur simultaneously with product decomposition as the project plan evolves.

The project manager must decide which process model is most appropriate for

- i. The customers who have requested the product and the people who will do the work,
- ii. The characteristics of the product itself, and
- iii. The project environment in which the software team works.

When a process model has been selected, the team then defines a preliminary project plan based on the set of common process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities must be created. The result is a complete plan reflecting the work tasks required to populate the framework activities. Project planning begins as a melding of the product and the process based on the various framework activities.

The Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by the software team must pass through the set of framework activities that have been defined for a software organization. Assume that the organization has adopted the following set of framework activities:

- i. *Customer communication*—tasks required to establish effective requirements elicitation between developer and customer.
- ii. *Planning*—tasks required to define resources, timelines, and other project related information.
- iii. *Risk analysis*—tasks required to assess both technical and management risks.

- iv. Engineering—tasks required to build one or more representations of the application.
- v. *Construction and release*—tasks required to construct, test, install, and provide user support (e.g., documentation and training).
- vi. *Customer evaluation*—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering activity and implemented during the construction activity.

If there is a more complex project, which has a broader scope and more significant business impact, such a project might require the following work tasks for the customer communication activity:

- i. Review the customer request.
- ii. Plan and schedule a formal, facilitated meeting with the customer.
- iii. Conduct research to specify the proposed solution and existing approaches.
- iv. Prepare a “working document” and an agenda for the formal meeting.
- v. Conduct the meeting.
- vi. Jointly develop mini-specs that reflect data, function, and behavioral features of the software.
- vii. Review each mini-spec for correctness, consistency, and lack of ambiguity.
- viii. Assemble the mini-specs into a scoping document.
- ix. Review the scoping document with all concerned.
- x. Modify the scoping document as required.

Check Your Progress 2.

List the criteria to evaluate a process.

3.3.4 THE PROJECT

In order to manage a successful software project, we must understand what can go wrong so that problems can be avoided and how to do it right.

There are ten signs that indicate that an information systems project is in risk:

- i. Software people don't understand their customer's needs.
- ii. The product scope is poorly defined.
- iii. Changes are managed poorly.

- iv. The chosen technology changes.
- v. Business needs change [or ill-defined].
- vi. Deadlines are unrealistic.
- vii. Users are resistant.
- viii. Sponsorship is lost [or was never properly obtained].
- ix. The project team lacks people with appropriate skills.
- x. Managers [and practitioners] avoid best practices and lessons learned.

There are few approaches to avoid the above problems:

- i. *Start on the right foot.* This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objects and expectations for everyone who will be involved in the project. It is reinforced by building the right team and giving the team the autonomy, authority, and technology needed to do the job.
- ii. *Maintain momentum.* Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.
- iii. *Track progress.* For a software project, progress is tracked as work products (e.g., specifications, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity. In addition, software process and project measures can be collected and used to assess progress against averages developed for the software development organization.
- iv. *Make smart decisions.* In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks.
- v. *Conduct a post-mortem analysis.* Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyse software project metrics, get feedback from team members and customers, and record findings in written form.

Check Your Progress 3.

What are project management objectives?

3.4 SUMMARY

This chapter covers the basic concept of project management, its need and its spectrum. And the role of people, process, product and project. And also explain the role of team, team leader in any project, the organisation of a team and the characteristics of a leader.

3.5 EXERCISE

- 1) What is the need to manage the software project?
 - 2) Define the concept of Project management and its spectrum.
 - 3) How People play a vital role in the management of people?
 - 4) Which type of risk encounter during information system project and also explain how they can be overcome?
 - 5) What is the role of team leader in a team? Which type of qualities a team leader hold?
 - 6) What is the role of team and team work? How a team can be organised during the project?
- Which type of points considered at the time of making team.

Software Engineering

SECOND - BLOCK

BLOCK

2

UNIT 1 Software Process and Project Metrics

UNIT 2 Software Project Planning

UNIT 3 Risk Analysis And Management

Overview

In this section we discuss the overview of this block's content. This block consists of the following units:

Unit 1 Software Process and Project Metrics

Measurement is fundamental to any engineering discipline, and software engineering is no exception. Measurement enables us to gain insight by providing a mechanism for objective evaluation. Measures are often collected by software engineers.

Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved. Software metrics are analysed and assessed by software managers.

Unit 2 Software Project Planning

Software managers do the planning using information solicited from customers and software engineers and software metrics data collected from past projects. Software project planning actually encompasses all of the estimation activities like—your attempt to determine how much money, how much effort, how many resources, and how much time it will take to build a specific software-based system or product.

Unit 3 Risk Analysis And Management

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

UNIT-1 SOFTWARE PROCESS AND PROJECT METRICS

- 1.0** Introduction
- 1.1** Objective
- 1.2** Reasons to Measure
- 1.3** Measures, Metric and Indicators
- 1.4** Software Measurement
- 1.5** Size-Oriented Metric
- 1.6** Function-Oriented Metric
- 1.7** Extended Function Point Metric
- 1.8** Summary
- 1.9** Exercise

1.0 INTRODUCTION

Software process and product metrics are quantitative measures that enable software people to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analysed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred.

If you don't measure, judgement can be based only on subjective evaluation. With measurement, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished over time. Begin by defining a limited set of process, project, and product measures that are easy to collect. These measures are often normalized using either size- or function-oriented metrics. The result is analysed and compared to past averages for similar projects performed within the organization. Trends are assessed and conclusions are generated. A set of software metrics that provide insight into the process and understanding of the project.

Within the context of software project management, we are concerned primarily with productivity and quality metrics—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For planning and estimating purposes, our interest is historical.

1.1 OBJECTIVE

Objectives of this unit are:

- a) to improve product quality and development-team productivity.
- b) Concerned with *productivity* and *quality* measures
 - measures of SW development output as function of effort and time
 - measures of usability
- c) Identify quantifiable questions and the related indicators that will use to help to achieve the measurement goals.
- d) to identify the data elements that will collect to construct the indicators that help answer to the questions raise in the mind.
- e) Define the measures to be used, and make these definitions operational.
- f) to Identify the actions that will take to implement the measures.
- g) to prepare a plan for implementing the measures.

1.2 REASONS TO MEASURE

There are four reasons for measuring software processes, products, and resources:

- i. To characterize
- ii. To evaluate
- iii. To predict
- iv. To improve

We characterize to gain understanding of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments. We evaluate to determine status with respect to plans. Measures are the sensors that let us know when our projects and processes are drifting off track, so that we can bring them back under control. We also evaluate to assess achievement of quality goals and to assess the impacts of technology and process improvements on products and processes.

We predict so that we can plan. Measuring for prediction involves gaining understandings of relationships among processes and products and building models of these relationships, so that the values we observe for some attributes can be used to predict others. We do this because we want to establish achievable goals for cost, schedule, and quality-so that appropriate resources can be applied. Predictive measures are also the basis for extrapolating trends, so estimates for cost, time, and quality can be updated based on current evidence. Projections and estimates based on historical data also help us analyse risks and make design/cost trade-offs. We measure to improve when we gather quantitative information to help us identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance.

Software Process

In order for software to be consistently well engineered, its development must be conducted in an orderly process. It is sometimes possible for a small software product to be developed without a well-defined process. However, for a software project of any substantial size, involving more than a few people, a good process are essential. The process can be viewed as a road map by which the project participants understand where they are going and how they are going to get there.

Thus, as depicted in figure 1.1, the software process is the set of activities and associated results that produce a software project.

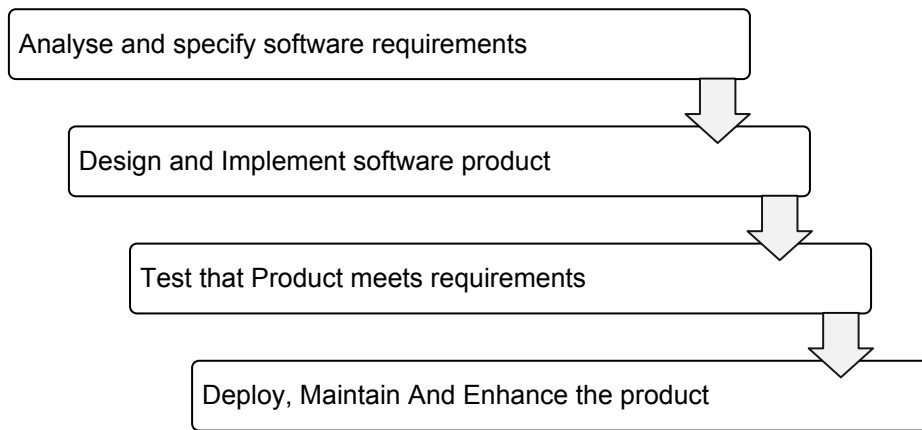


Figure 1.1: Software Process

The first two steps of the process are often referred to, respectively, as the "what and how" of software development. The "Analyse and Specify" step defines what the problem is to be solved; the "Design and Implement" step entails how the problem is solved.

Software Process Characteristics

The following are the software process characteristics:

- a) Understand ability
- b) Visibility
- c) Robustness
- d) Reliability
- e) Acceptability
- f) Maintainability
- g) Rapidity
- h) Supportability

Project Metrics

Software process metrics are used for strategic purposes. Software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates. The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of pages of documentation, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from specification into design, technical metrics are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold.

First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks.

Second, project metrics are used to assess product quality on an on-going basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

Another model of software project metrics suggests that every project should measure:

- *Inputs*- measures of the resources (e.g., people, environment) required to do the work.
- *Outputs*- measures of the deliverables or work products created during the software engineering process.
- *Results*- measures that indicate the effectiveness of the deliverables.

In actuality, this model can be applied to both process and project. In the project context, the model can be applied recursively as each framework activity occurs. Therefore the output from one activity becomes input to the next. Results metrics can be used to provide an indication of the usefulness of work products as they flow from one framework activity to the next.

Check Your Progress 1.

What is the difference between Process Metric and Product Metric?

1.3 MEASURES, METRIC AND INDICATORS

Although the terms measure, measurement, and metrics are often used interchangeably, it is important to note the subtle differences between them. Because measure can be used either a noun or a verb, definitions of the term can become confusing.

Within the software engineering context, a **measure** provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure.

As per the IEEE Standard Glossary of Software Engineering Terms [IEE93]:

“A quantitative measure of the degree to which a system, component, or process possesses a given attribute.”

When a single data point has been collected, a measure has been established. Measurement occurs as the result of the collection of one or more data points.

Software **metric** relates the individual measures in some way e.g., the average number of errors found per review or the average number of errors found per person-hour expended on reviews.

A software engineer collects measures and develops metrics so that indicators will be obtained. An **indicator** is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself. An indicator provides insight that enables the project manager or software engineers to adjust the process, the project, or the process to make things better. For example, four software teams are working on a large software project.

Each team must conduct design reviews but is allowed to select the type of review that it will use. Upon examination of the metric, errors found per person-hour expended, the project manager notices that the two teams using more formal review methods exhibit an errors found per person-hour expended that is 40 percent higher than the other teams. Assuming all other parameters equal, this provides the project manager with an indicator that formal review methods may provide a higher return on time investment than another, less formal review approaches. She may decide to suggest that all teams use the more formal approach. The metric provides the manager with insight. And insight leads to informed decision making.

Metrics in the process and project domains:

Measurement is commonplace in the engineering world. We measure power consumption, weight, physical dimensions, temperature, voltage, signal-to-noise ratio, etc. Unfortunately, measurement is far less common in the software engineering world. We have trouble agreeing on what to measure and trouble evaluating measures that are collected.

Metrics should be collected so that process and product indicators can be ascertained. Process indicators enable a software engineering organization to gain insight into the efficacy of an existing process. They enable managers and practitioners to assess what works and what

doesn't. Process metrics are collected across all projects and over long periods of time. Their intent is to provide indicators that lead to long-term software process improvement.

Project indicators enable a software project manager to

- (1) Assess the status of an on-going project,
- (2) Track potential risks,
- (3) Uncover problem areas before they go "critical,"
- (4) Adjust work flow or tasks, and
- (5) Evaluate the project team's ability to control quality of software work products.

Process Metrics and Software Process Improvement:

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement.

We measure the efficacy of a software process indirectly. That is, we derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end-users, work products delivered, human effort expended, calendar time expended, schedule conformance, and other measures. We also derive process metrics by measuring the characteristics of specific software engineering tasks.

Grady argues that there are "private and public" uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis; these data should be private or the individual and serve as an indicator for the individual only. Public metrics generally assimilate information that originally was private to individuals and teams.

Check Your Progress 2.

What is the difference between a measure and an indicators?

1.4 SOFTWARE MEASUREMENT

Measurements in the physical world can be categorized in two ways:

Direct measures (e.g., the length of a bolt) and

Indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects).

Category: Software metrics can be categorized similarly

Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.

Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "-abilities".

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

We have already partitioned the software metrics domain into process, project, and product metrics. We have also noted that product metrics that are private to an individual are often combined to develop project metrics that are public to a software team. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects?

To illustrate, we consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because we do not know the size or complexity of the projects, we cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

Check Your Progress 3.

What is meant by measurement and metrics?

1.5 SIZE-ORIENTED METRICS

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in figure 1.2, can be created. The table lists each software development project that

has been completed over the past few years and corresponding measures for that project. Referring to the table entry for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of \$168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects per KLOC.
- \$ per LOC.
- Page of documentation per KLOC.

Project	LOC	Effort	\$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

Figure 1.2: Size oriented Matrix

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- \$ per page of documentation.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development. Most of the controversy swirls around the use of lines of code as a key

measure. Proponents of the LOC measure claim that LOC is an "artefact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve.

Check Your Progress 4.

Write the any two advantages of LOC.

1.6 FUNCTION-ORIENTED METRICS

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the function point. Function points are derived using an empirical relationship based on countable measures of software's information domain and assessments of software complexity.

Function points are computed by completing the table shown in figure 1.3.

Five information domain characteristics are determined and counts are provided in the appropriate table location. Information domain values are defined in the following manner:

1. *Number of user inputs.* Each user input that provides distinct application oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.
2. *Number of user outputs.* Each user output that provides application oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.
3. *Number of user inquiries.* An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.
4. *Number of files.* Each logical master file is counted.
5. *Number of external interfaces.* All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

Measurement parameter	Count	Weighting factor			=	
		Simple	Average	Complex		
Number of user inputs	<input type="text"/>	× 3	4	6	=	<input type="text"/>
Number of user outputs	<input type="text"/>	× 4	5	7	=	<input type="text"/>
Number of user inquiries	<input type="text"/>	× 3	4	6	=	<input type="text"/>
Number of files	<input type="text"/>	× 7	10	15	=	<input type="text"/>
Number of external interfaces	<input type="text"/>	× 5	7	10	=	<input type="text"/>
Count total	→					<input type="text"/>

Figure 1.3: Computing Function Point

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

$$FP = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)] \quad (4-1)$$

Where count total is the sum of all FP entries obtained from Figure

The F_i ($i = 1$ to 14) are "complexity adjustment values" based on responses to following questions:

1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?

12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (4-1) and the weighting factors that are applied to information domain counts are determined empirically.

Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- \$ per FP.
- Pages of documentation per FP.
- FP per person-month.

Check Your Progress 5.

What are the measuring parameters of function oriented metrics?

1.7 EXTENDED FUNCTION POINT METRICS

The function point measure was originally designed to be applied to business information systems applications. To accommodate these applications, the data dimension was emphasized to the exclusion of the functional and behavioural dimensions. For this reason, the function point measure was inadequate for many engineering and embedded systems. A number of extensions to the basic function point measure have been proposed to remedy this situation. A function point extension called feature points, is a superset of the function point measure that can be applied to systems and engineering software applications.

The feature point measure accommodates applications in which algorithmic complexity is high. Real-time, process control and embedded software applications tend to have high algorithmic complexity and are therefore amenable to the feature point. To compute the feature point, information domain values are again counted and weighted. In addition, the feature point metric counts anew software characteristic—algorithms. An algorithm is defined as "a bounded computational problem that is included within a specific computer

program”. Inverting a matrix, decoding a bit string, or handling an interrupt are all examples of algorithms.

Another function point extension for real-time systems and engineered products has been developed by Boeing. The Boeing approach integrates the data dimension of software with the functional and control dimensions to provide a function-oriented measure amenable to applications that emphasize function and control capabilities. Called the 3D function point, characteristics of all three software dimensions are “counted, quantified, and transformed” into a measure that provides an indication of the functionality delivered by the software. The data dimension is evaluated in much the same way. Counts of retained data and external data are used along with measures of complexity to derive a data dimension count. The functional dimension is measured by considering “the number of internal operations required to transform input to output data”. For the purposes of 3D function point computation, a “transformation” is viewed as a series of processing steps that are constrained by a set of semantic statements. The control dimension is measured by counting the number of transitions between states. A state represents some externally observable mode of behaviour, and a transition occurs as a result of some event that causes the software or system to change its mode of behaviour. For example, a wireless phone contains software that supports auto dial functions. To enter the auto-dial state from a resting state, the user presses an Auto key on the keypad. This event causes an LCD display to prompt for a code that will indicate the party to be called. Upon entry of the code and hitting the Dial key (another event), the wireless phone software makes a transition to the dialling state. When computing 3D function points, transitions are not assigned a complexity value.

To compute 3D function points, the following relationship is used:

$$index = I + O + Q + F + E + T + R$$

where I, O, Q, F, E, T, and R represent complexity weighted values for the elements discussed already: inputs, outputs, inquiries, internal data structures, external files, transformation, and transitions, respectively.

Semantic statements Processing steps	1-5	6-10	11+
1-10	Low	Low	Average
11-20	Low	Average	High
21+	Average	High	High

Figure 1.4: Determining the complexity of a transformation for 3D function points

Each complexity weighted value is computed using the following relationship:

$$\text{Complexity weighted value} = N_{il}W_{il} + N_{ia}W_{ia} + N_{ih}W_{ih}$$

Where N_{il} , N_{ia} , and N_{ih} represent the number of occurrences of element i (e.g., outputs) for each level of complexity (low, medium, high); and W_{il} , W_{ia} , and W_{ih} are the corresponding weights. The overall complexity of a transformation for 3D function points is shown in above figure 1.4.

It should be noted that function points, feature points, and 3D function points represent the same thing—"functionality" or "utility" delivered by software. In fact, each of these measures results in the same value if only the data dimension of an application is considered. For more complex real-time systems, the feature point count is often between 20 and 35 percent higher than the count determined using function points alone.

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages; that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data; that counts of the information domain can be difficult to collect after the fact; and that FP has no direct physical meaning—it's just a number.

1.8 SUMMARY

This section covers the discussion about Software Process and Project Metrics, its characteristics, and also discuss Measures, reasons behind software measure, its pros and cons, Software Metric and Indicators, types of Metric like Size-Oriented Metric, Function-Oriented Metric, Extended Function Point.

1.9 EXERCISE

- 1) Define the term metrics. What are the types of metrics?
- 2) What are the advantages and disadvantages of size measure?
- 3) What is LOC? How it is used for project estimation?
- 4) What is Software Process? Write the characteristics of software Process.
- 5) Define Software Measures in detail.
- 6) What is FP? How to compute Function Point? How it is used for project estimation?

UNIT-2 SOFTWARE PROJECT PLANNING

2.0 Introduction

2.1 Objective

2.2 Need of Software Project Planning

2.3 Project Planning Objectives

2.4 Software project Estimation

2.5 Decomposition techniques

2.6 Problem Based Estimation

2.7 Process Based Estimation

2.8 Empirical Estimation Models

2.9 The COCOMO Model

2.10 Summary

2.11 Exercise

2.0 INTRODUCTION

Process and project metrics can provide historical perspective and powerful input for the generation of quantitative estimates. Past experience can aid immeasurably as estimates are developed and reviewed. Because estimation lays a foundation for all other project planning activities and project planning provides the road map for successful software engineering, we would be ill-advised to embark without it.

2.1 OBJECTIVE

Objectives of this unit are:

- a) to provide a framework for manager to make reasonable estimates of resources, costs and schedules
- b) to provide the knowledge to make products easier to use.
- c) to reduce the time it takes to get a new product to market.

2.2 NEED OF SOFTWARE PROJECT PLANNING

Software project management begins with a set of activities that are collectively called *project planning*. Before the project can begin, the manager and the software team must estimate the work to be done, the resources that will be required, and the time that will elapse from start to finish. Whenever estimates are made, we look into the future and accept some degree of uncertainty as a matter of course.

To quote Frederick Brooks [BRO75]: “. . . *our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption that is quite untrue, i.e., that all will go well. . . . because we are uncertain of our estimates, software managers often lack the courteous stubbornness to make people wait for a good product.*”

Although estimating is as much art as it is science, this important activity need not be conducted in a haphazard manner. Useful techniques for time and effort estimation do exist.

Important: Would you build a house without knowing how much you were about to spend? Of course not, and since most computer-based systems and products cost considerably more to build than a large house, it would seem reasonable to develop an estimate before you start creating the software.

Steps: Estimation begins with a description of the scope of the product. Until the scope is “bounded” it’s not possible to develop a meaningful estimate. The problem is then

decomposed into a set of smaller problems and each of these is estimated using historical data and experience as guides. It is advisable to generate your estimates using at least two different methods. Problem complexity and risk are considered before a final estimate is made.

If we want to ensure that we have done it right? That's hard, because you won't really know until the project has been completed. However, if you have experience and follow a systematic approach, generate estimates using solid historical data, create estimation data points using at least two different methods, and factor in complexity and risk, you can feel confident that you've given it your best shot.

Observations on Estimating for Planning:

Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information, and the courage to commit to quantitative predictions when qualitative information is all that exists. Estimation carries inherent risk and this risk leads to uncertainty.

Project complexity has a strong effect on the uncertainty inherent in planning. Complexity, however, is a relative measure that is affected by familiarity with past effort. The first-time developer of a sophisticated e-commerce application might consider it to be exceedingly complex. However, a software team developing its tenth e-commerce Web site would consider such work run of the mill. A number of quantitative software complexity measures have been proposed. Such measures are applied at the design or code level and are therefore difficult to use during software planning. However, other, more subjective assessments of complexity can be established early in the planning process.

Project size is another important factor that can affect the accuracy and efficacy of estimates. As size increases, the interdependency among various elements of the software grows rapidly. Problem decomposition, an important approach to estimating, becomes more difficult because decomposed elements may still be formidable. To paraphrase Murphy's law: "What can go wrong will go wrong"—and if there are more things that can fail, more things will fail.

The degree of structural uncertainty also has an effect on estimation risk. In this context, structure refers to the degree to which requirements have been solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information that must be processed.

The availability of historical information has a strong influence on estimation risk. By looking back, we can emulate things that worked and improve areas where problems arose. When comprehensive software metrics are available for past projects, estimates can be made with greater assurance, schedules can be established to avoid past difficulties, and overall risk is reduced.

Risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule. If project scope is poorly understood or project requirements are subject to change, uncertainty and risk become dangerously high. The software planner should demand completeness of function, performance, and interface definitions (contained in a System Specification). The planner, and more important, the customer should recognize that variability in software requirements means instability in cost and schedule.

However, a project manager should not become obsessive about estimation. Modern software engineering approaches take an iterative view of development. In such approaches, it is possible to revisit the estimate and revise it when the customer makes changes to requirements.

Check Your Progress 1.

What is the difference between feasibility study and planning?

2.3 PROJECT PLANNING OBJECTIVES

Project planning is the very important activity. Its major objectives are:

(A) Provide a framework

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded.

(B) Determination of software scope

The first activity in software project planning is the determination of software scope. Function and performance allocated to software during system engineering should be assessed to establish a project scope that is unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. Software scope describes the data and control to be processed, function, performance,

constraints, interfaces, and reliability. Functions described in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

a) *Obtaining information necessary for scope:*

Things are always somewhat hazy at the beginning of a software project. A need has been defined and basic goals and objectives have been enunciated, but the information necessary to define scope has not yet been delineated.

b) *Preliminary meeting or interview:*

The most commonly used technique to bridge the communication gap between the customer and developer and to get the communication process started is to conduct a preliminary meeting or interview.

- i. *Set-1:*** The first set of context-free questions focuses on the customer, the overall goals and benefits. For example, the analyst might ask:
 - Who is behind the request for this work?
 - Who will use the solution?
 - What will be the economic benefit of a successful solution?
 - Is there another source for the solution?
- ii. *Second Set:*** The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice any perceptions about a solution:
 - How would you (the customer) characterize "good" output that would be generated by a successful solution?
 - What problem(s) will this solution address?
 - Can you show me (or describe) the environment in which the solution will be used?
 - Will any special performance issues or constraints affect the way the solution is approached?
- iii. *Third Set:*** The final set of questions focuses on the effectiveness of the meeting with propose the following list:
 - Are you the right person to answer these questions? Are answers "official"?
 - Are my questions relevant to the problem that you have?

- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Check Your Progress 2.

What are the steps involved in identification of project scope and objectives?

2.4 SOFTWARE PROJECT ESTIMATION

Effective software project estimation is one of the most challenging and important activities in software development. Proper project planning and control is not possible without a sound and reliable estimate. As a whole, the software industry doesn't estimate projects well and doesn't use estimates appropriately. We suffer far more than we should as a result and we need to focus some effort on improving the situation. Under-estimating a project leads to under-staffing it, under-scoping the quality assurance effort, and setting too short a schedule. For those who figure on avoiding this situation by generously padding the estimate, over-estimating a project can be just about as bad for the organization! If you give a project more resources than it really needs without sufficient scope controls it will use them. The project is then likely to cost more than it should, take longer to deliver than necessary, and delay the use of your resources on the next project.

The four basic steps in software project estimation are:

- a)** Estimate the size of the development product.
 - i.** By analogy
 - ii.** By counting product features and using an algorithmic approach
- b)** Estimate the effort in person-months or person-hours.
 - i.** Use your organization's own historical data
 - ii.** Use a mature and generally accepted algorithmic approach such as Barry Boehm's COCOMO model or the Putnam Methodology
- c)** Estimate the schedule in calendar months.
- d)** Estimate the project cost in dollars (or local currency)

To achieve reliable cost and effort estimates, a number of options arise:

- a) Delay estimation until late in the project (obviously, we can achieve 100% accurate estimates after the project is complete!).
- b) Base estimates on similar projects that have already been completed.
- c) Use relatively simple decomposition techniques to generate project cost and effort estimates.
- d) Use one or more empirical models for software cost and effort estimation.

The Trouble with Estimates

- a) Estimating size is the most difficult (but not impossible) step intellectually, and is often skipped in favour of going directly to estimating a schedule.
- b) Customers and software developers often don't really recognize that software development is a process of gradual refinement.
- c) Organizations often don't collect and analyse historical data on their performance on development projects.
- d) It is often difficult to get a realistic schedule accepted by management and customers.

Check Your Progress 3.

How do you estimate the effort for your project?

2.5 DECOMPOSITION TECHNIQUES

Software project estimation is a form of problem solving, and in most cases, the problem to be solved is too complex to be considered in one piece. For this reason, we decompose the problem, re-characterizing it as a set of smaller problems.

The decomposition approach is in two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, the project planner must understand the scope of the software to be built and generate an estimate of its "size."

Software Sizing

The accuracy of a software project estimate is predicated on a number of things:

- a. The degree to which the planner has properly estimated the size of the product to be built;

- b. The ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects);
- c. The degree to which the project plan reflects the abilities of the software team; and
- d. The stability of product requirements and the environment that supports the software engineering effort.

There are four different approaches to the sizing problem:

- a. “Fuzzy logic” sizing
- b. Function point sizing
- c. Standard component sizing
- d. Change sizing

AUTOMATED ESTIMATION TOOLS

The decomposition techniques sections are available as part of a wide variety of software tools. These automated estimation tools allow the planner to estimate cost and effort and to perform "what-if" analyses for important project variables such as delivery date or staffing.

Although many automated estimation tools exist, all exhibit the same general characteristics and all perform the following six generic functions:

- a. ***Sizing of project deliverables.*** The “size” of one or more software work products is estimated. Work products include the external representation of software (e.g., screen, reports), the software itself (e.g., KLOC), functionality delivered (e.g., function points), descriptive information (e.g. documents).
- b. ***Selecting project activities.*** The appropriate process framework is selected and the software engineering task set is specified.
- c. ***Predicting staffing levels.*** The number of people who will be available to do the work is specified. Because the relationship between people available and work (predicted effort) is highly nonlinear, this is an important input.
- d. ***Predicting software effort.*** Estimation tools use one or more models that relate the size of the project deliverables to the effort required to produce them.
- e. ***Predicting software cost.*** Given the results of step 4, costs can be computed by allocating labour rates to the project activities noted in step 2.
- f. ***Predicting software schedules.*** When effort, staffing level, and project activities are known, a draft schedule can be produced by allocating labour across software engineering activities based on recommended models for effort distribution

Check Your Progress 4.

Which software project sizing approach develop estimates of the information domain characteristics?

2.6 PROBLEM -BASED ESTIMATION

Lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation:

- (1) As an estimation variable to "size" each element of the software and
- (2) As baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. The project planner begins with a bounded statement of software scope and from this statement attempts to decompose software into problem functions that can each be estimated individually. LOC or FP is then estimated for each function. Alternatively, the planner may choose another component for sizing such as classes or objects, changes, or business processes affected.

Baseline productivity metrics (e.g., LOC/pm or FP/pm⁹) are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project. It is important to note, however, that there is often substantial scatter in productivity metrics for an organization, making the use of a single baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail value that can be tied to past data and used to generate an estimate. Regardless of the estimation variable that is used, the project planner begins by estimating a range of values for each function or information domain value. Using historical data or intuition, the planner

estimates an optimistic, most likely and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified.

A three-point or expected value can then be computed. The expected value for the estimation variable (size), S , can be computed as a weighted average of the optimistic, most likely (s_m), and pessimistic (s_{pess}) estimates.

For example, $S = (s_{opt} + 4s_m + s_{pess})/6$ (5-1)

gives heaviest credence to the “most likely” estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied. Are the estimates correct? The only reasonable answer to this question is: "We can't be sure." Any estimation technique, no matter how sophisticated, must be cross-checked with another approach. Even then, common sense and experience must prevail.

Check Your Progress 5.

What is beta probability distribution?

2.7 PROCESS-BASED ESTIMATION

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated. Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of software process activities must be performed for each function. Functions and related software process activities may be represented as part of a table similar to the one presented in figure 2.1

Common process framework activities	Customer communication				Planning				Risk analysis				Engineering			
	Software engineering tasks															
Product functions																
Text input																
Editing and formatting																
Automatic copy edit																
Page layout capability																
Automatic indexing and TOC																
File management																
Document production																

Figure 2.1: Melding the Problem and the Process

Once problem functions and process activities are melded, the planner estimates the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function. These data constitute the central matrix of the table in Figure 2.1.

Average labour rates (i.e., cost/unit effort) are then applied to the effort estimated for each process activity. It is very likely the labour rate will vary for each task. Senior staff heavily involved in early activities is generally more expensive than junior staff involved in later design tasks, code generation, and early testing.

Costs and effort for each function and software process activity are computed as the last step. If process-based estimation is performed independently of LOC or FP estimation, we now have two or three estimates for cost and effort that may be compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.

Check Your Progress 6

On which base d Process-based estimation techniques require problem decomposition

2.8 EMPIRICAL ESTIMATION MODELS

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP. Values for LOC or FP are estimated using the approach. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model.

The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, the results obtained from such models must be used judiciously.

The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form-

$$E = A + B \times (ev)^C$$

Where A, B, and C are empirically derived constants, E is effort in person-months, and ev is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation, the majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment).

Among the many LOC-oriented estimation models proposed in the literature are

$$E = 5.2 \times (KLOC)^{0.91} \text{Walston-Felix model}$$

$$E = 5.5 + 0.73 \times (KLOC)^{1.16} \text{Bailey-Basili model}$$

$$E = 3.2 \times (KLOC)^{1.05} \text{Boehm simple model}$$

$$E = 5.288 \times (KLOC)^{1.047} \text{Doty model for } KLOC > 9$$

FP-oriented models have also been proposed. These include

$$E = -13.39 + 0.0545 \text{ FP Albrecht and Gaffney model}$$

$$E = 60.62 \times 7.728 \times 10^{-8} \text{ FP}^3 \text{Kemerer model}$$

$$E = 585.7 + 15.12 \text{ FP Matson, Barnett, and Mellichamp model}$$

A quick examination of these models indicates that each will yield a different result for the same values of LOC or FP. The implication is clear. Estimation models must be calibrated for local needs.

Check Your Progress 7

On Which, Empirical estimation models are typically based .

2.9 COCOMO MODEL

The Constructive Cost Model (COCOMO) is an algorithmic software cost estimation model developed by Barry W. Boehm. The model uses a basic regression formula with parameters that are derived from historical project data and current as well as future project characteristics.

First published in Boehm's 1981 book *Software Engineering Economics* as a model for estimating effort, cost, and schedule for software projects. It drew on a study of 63 projects at TRW Aerospace where Boehm was Director of Software Research and Technology. The study examined projects ranging in size from 2,000 to 100,000 lines of code, and programming languages ranging from assembly to PL/I. These projects were based on the waterfall model of software development which was the prevalent software development process in 1981.

References to this model typically call it COCOMO 81. In 1995 COCOMO II was developed and finally published in 2000 in the book *Software Cost Estimation with COCOMO II*. COCOMO II is the successor of COCOMO 81 and is better suited for estimating modern software development projects. It provides more support for modern software development processes and an updated project database. The need for the new model came as software development technology moved from mainframe and overnight batch processing to desktop development, code reusability, and the use of off-the-shelf software components.

COCOMO consists of a hierarchy of three increasingly detailed and accurate forms.

The first level, Basic COCOMO is good for quick, early, rough order of magnitude estimates of software costs, but its accuracy is limited due to its lack of factors to account for difference in project attributes (Cost Drivers). Intermediate COCOMO takes these Cost Drivers into account and Detailed COCOMO additionally accounts for the influence of individual project phases.

Basic COCOMO

Basic COCOMO computes software development effort (and cost) as a function of program size. Program size is expressed in estimated thousands of source lines of code (SLOC, KLOC).

COCOMO applies to three classes of software projects:

- a) *Organic projects* - "small" teams with "good" experience working with "less than rigid" requirements

- b) *Semi-detached projects* - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements
- c) *Embedded projects* - developed within a set of "tight" constraints. It is also combination of organic and semi-detached projects. (Hardware, software, operational,)

The basic COCOMO equations take the form

$$\mathbf{Effort\ Applied\ (E)} = a_b (\text{KLOC})^{b_b} [\text{person-months}]$$

$$\mathbf{Development\ Time\ (D)} = c_b (\text{Effort Applied})^{d_b} [\text{months}]$$

$$\mathbf{People\ required\ (P)} = \text{Effort Applied} / \text{Development Time} [\text{count}]$$

Where, KLOC is the estimated number of delivered lines (expressed in thousands) of code for project. The coefficients a_b , b_b , c_b and d_b are given in the following table:

Software project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Basic COCOMO is good for quick estimate of software costs. However it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and so on.

Intermediate COCOMO

Intermediate COCOMO computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes. This extension considers a set of four "cost drivers", each with a number of subsidiary attributes:-

- a) Product attributes
 - Required software reliability
 - Size of application database
 - Complexity of the product
- b) Hardware attributes
 - Run-time performance constraints
 - Memory constraints
 - Volatility of the virtual machine environment
 - Required turnabout time

c) Personnel attributes

- Analyst capability
- Software engineering capability
- Applications experience
- Virtual machine experience
- Programming language experience

d) Project attributes

- Use of software tools
- Application of software engineering methods
- Required development schedule

Each of the 15 attributes receives a rating on a six-point scale that ranges from "very low" to "extra high" (in importance or value). An effort multiplier from the table below applies to the rating. The product of all effort multipliers results in an effort adjustment factor (EAF). Typical values for EAF range from 0.9 to 1.4.

The Intermediate COCOMO formula now takes the form:

$$E = a_i (KLoC)^{b_i} \cdot EAF$$

Where E is the effort applied in person-months, KLoC is the estimated number of thousands of delivered lines of code for the project, and EAF is the factor calculated above. The coefficient a_i and the exponent b_i are given in the next table.

Software project	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

The Development time D calculation uses E in the same way as in the Basic COCOMO.

Detailed COCOMO

Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

The detailed model uses different effort multipliers for each cost driver attribute. These Phase Sensitive effort multipliers are each to determine the amount of effort required to complete each phase. In detailed COCOMO, the whole software is divided in different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort.

In detailed COCOMO, the effort is calculated as function of program size and a set of cost drivers given according to each phase of software life cycle. A Detailed project schedule is never static. The five phases of detailed COCOMO are:-

- Plan and requirement.
- System design.
- Detailed design.
- Module code and test.
- Integration and test.

Check Your Progress 8.

Write the advantages of COCOMO.

2.10 SUMMARY

This unit covers a small discussion about Software Project Planning, its objective, Basic steps of Software project Estimation, Decomposition techniques, Automated Estimation tools, Problem Based Estimation, Process Based Estimation, Empirical Estimation Models, COCOMO I and II Model.

2.11 EXERCISE

- 1) What is Software Project Planning? What is the objective of Software Project Planning?
- 2) Explain the COCOMO for software cost estimation.
- 3) Discuss the techniques for estimating project duration and determining the staffing pattern.
- 4) Explain Software Project Estimation Techniques.
- 5) Write short note on the Decomposition techniques.
- 6) Define Empirical Estimation Models in detail.
- 7) Estimate the effort required to develop software for a simple module that produces 15 screens, 10 reports and will require around 100 software components. Assume average complexity and average developer / environment maturity. Use the Application Composition Model of COCOMO-II with Object Points. State any assumptions you make.

UNIT-3 RISK ANALYSIS AND MANAGEMENT

3.0 Introduction

3.1 Objective

3.2 What Is Risk?

3.3 Software Risk

3.4 Risk Identification

3.5 Risk Reduction

3.6 Risk Projection

3.7 Risk Refinement

3.8 Risk Mitigation, Monitoring and Management

3.9 Summary

3.10 Exercise

3.0 INTRODUCTION

Software is a difficult undertaking. Lots of things can go wrong, and frankly, many often do. It's for this reason that being prepared— understanding the risks and taking proactive measures to avoid or manage them—is a key element of good software project management. Recognizing what can go wrong is the first step, called “risk identification.” Next, each risk is analysed to determine the likelihood that it will occur and the damage that it will do if it does occur. Once this information is established, risks are ranked, by probability and impact. Finally, a plan is developed to manage those risks with high probability and high impact. Risk mitigation, monitoring, and management (RMMM) plan or a set of risk information sheets is produced. How do I ensure that I've done it right? The risks that are analysed and managed should be derived from thorough study of the people, the product, the process, and the project. The RMMM should be revisited as the project proceeds to ensure that risks are kept up to date. Contingency plans for risk management should be realistic.

3.1 OBJECTIVE

Objectives of this unit are:

- a) to identify the risks and determine if they may be avoided.
- b) to achieve and maintain reduced cost of risk.
- c) to evaluate and assess all risks of loss.
- d) to develop and maintain risk management policies.
- e) to Identify total assets and resources of organizations.
- f) to Calculate values of assets and resources.

3.2 WHAT IS RISK?

In general Risk is:

- First, risk concerns future happenings. Today and yesterday are beyond active concern, as we are already reaping what was previously sowed by our past actions. The question is, can we, therefore, by changing our actions today, create an opportunity for a different and hopefully better situation for ourselves tomorrow.
- Second, that risk involves change, such as in changes of mind, opinion, actions, or places.
- Third, risk involves choice, and the uncertainty that choice itself entails.

When risk is considered in the context of software engineering, Charette's three conceptual underpinnings are always in evidence-

- The future is our concern— what risks might cause the software project to go awry?
- Change is our concern— how will changes in customer requirements, development technologies, target computers, and all other entities connected to the project affect timeliness and overall success?
- Last, we must grapple with choices—what methods and tools should we use, how many people should be involved, how much emphasis on quality is "enough"?

Everyone involved in the software process—managers, software engineers, and customers—participate in risk analysis and management.

As per Peter Drucker, "While it is futile to try to eliminate risk, and questionable to try to minimize it, it is essential that the risks taken be the right risks". Before we can identify the "right risks" to be taken during a software project, it is important to identify all risks that are obvious to both managers and practitioners.

There are two risk strategies- Reactive and Proactive. The majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire fighting mode.

When this fails, "crisis management" takes over and the project is in real danger. A considerably more intelligent strategy for risk management is to be proactive. A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

Check Your Progress 1.

Give the two important characteristics of the risk management?

3.3 SOFTWARE RISK

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics

- Uncertainty—the risk may or may not happen; that is, there are no 100% probable risks.

- Loss—if the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analysed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project. Project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced.

If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors.

Technical risks occur because the problem is harder to solve than we thought it would be.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product.

Candidates for the top five business risks are

- i. Building an excellent product or system that no one really wants (market risk)
- ii. Building a product that no longer fits into the overall business strategy for the company (strategic risk)
- iii. Building a product that the sales force doesn't understand how to sell
- iv. Losing the support of senior management due to a change in focus or a change in people (management risk)
- v. Losing budgetary or personnel commitment (budget risks). It is extremely important to note that simple categorization won't always work.

Some risks are simply unpredictable in advance.

Another general categorization of risks has been proposed by Charette-

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as on-going maintenance requests are serviced).

Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Check Your Progress 2.

What is the difference between the “Known Risks” and Predictable Risks” ?

3.4 RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories: generic risks and product-specific risks. Generic risks are a potential threat to every software project. Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand. To identify product-specific risks, the project plan and the software statement of scope are examined and an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

- Product size- risks associated with the overall size of the software to be built or modified.
- Business impact- risks associated with constraints imposed by management or the marketplace.
- Customer characteristics- risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- Process definition- risks associated with the degree to which the software process has been defined and is followed by the development organization.
- Development environment- risks associated with the availability and quality of the tools to be used to build the product.

- Technology to be built- risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- Staff size and experience- risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory.

Assessing Overall Project Risk

The following questions have derived from risk data obtained by surveying experienced software project managers in different part of the world:

1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and their customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end-users have realistic expectations?
6. Is project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be implemented?
10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

Risk Components and Drivers

The risk components are defined in the following manner:

- ***Performance risk***- the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- ***Cost risk***- the degree of uncertainty that the project budget will be maintained.
- ***Support risk***- the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.

- **Schedule risk**- the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories-

- Negligible
- Marginal
- Critical
- Catastrophic

3.5 RISK REDUCTION

Generate reliability specifications, including quantitative requirements defining the acceptable levels of failure. There are types of functional reliability requirements:

- **Checking requirements** identify checks to ensure that incorrect data is detected before it leads to a failure.
- **Recovery requirements** are geared to help the system recover after a failure has occurred.
- **Redundancy requirements** specify redundant features of the system to be included.
- **Process requirements** for reliability specify the development process to be used may also be included.

3.6 RISK PROJECTION

Risk projection, also called risk estimation, attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur.

The project planner, along with other managers and technical staff, performs four risk projection activities:

- Establish a scale that reflects the perceived likelihood of a risk,
- Delineate the consequences of the risk,
- Estimate the impact of the risk on the project and the product, and
- Note the overall accuracy of the risk projection so that there will be no misunderstandings.

Developing a Risk Table

A risk table provides a project manager with a simple technique for risk projection. A sample risk table is illustrated in figure 3.1.

A project team begins by listing all risks in the first column of the table. This can be accomplished with the help of the risk item checklists. Each risk is categorized in the second column. The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. Individual team members are polled in round-robin fashion until their assessment of risk probability begins to converge.

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
•				
•				
•				

Figure3.1: Sample risk table prior to sorting

Next, the impact of each risk is assessed. Each risk component is assessed using the characterization, and an impact category is determined. The categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.

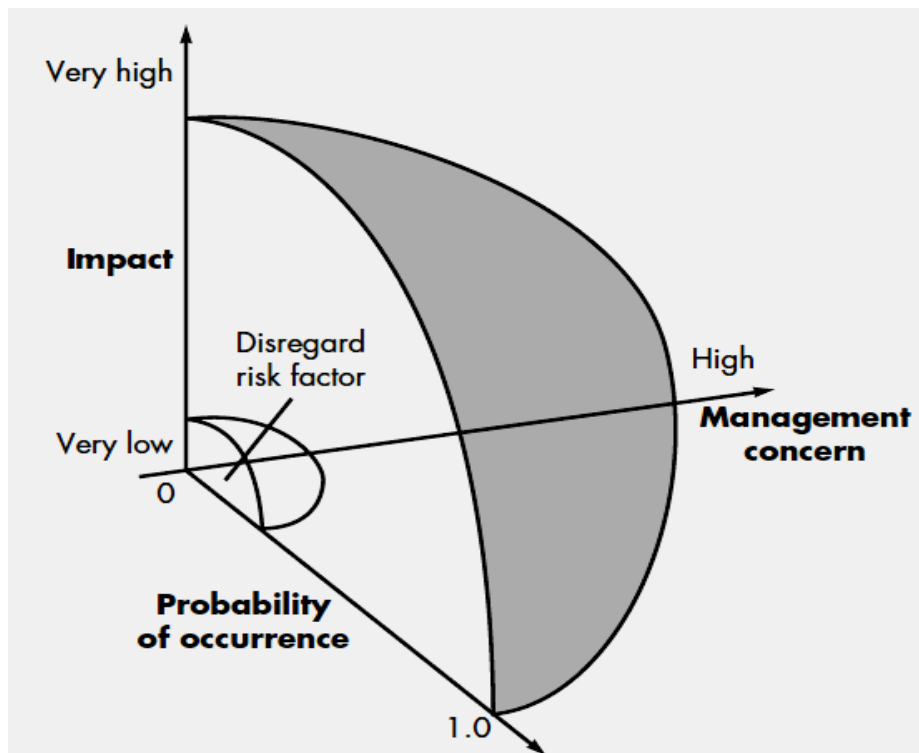


Figure 3.2: Risk and management concern

Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization. The project manager studies the resultant sorted table and defines a cut-off line. The cut-off line (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are re-evaluated to accomplish second-order prioritization. Risk impact and probability have a distinct influence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time, as shown in figure 3.2, However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

All risks that lie above the cut-off line must be managed. The column labelled RMMM contains a pointer into a Risk Mitigation, Monitoring and Management Plan or alternatively, a collection of risk information sheets developed for all risks that lie above the cut-off.

Risk probability can be determined by making individual estimates and then developing a single consensus value. Risk drivers can be assessed on a qualitative probability scale that has the following values: impossible, improbable, probable, and frequent. Mathematical probability can then be associated with each qualitative value

Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many customers are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt. In most cases, a project manager might want the “bad news” to occur as soon as possible, but in some cases, the longer the delay, the better.

The following steps are recommended to determine the overall consequences of a risk:

1. Determine the average probability of occurrence value for each risk component.
2. Determine the impact for each component based on the criteria shown.
3. Complete the risk table and analyse the results as described in the preceding sections.

The overall risk exposure, RE, is determined using the following relationship:

$$RE = P \times C$$

where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

The software team defines a project risk in the following manner:

Risk identification: Only 70 percent of the software components scheduled for re use will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability: 80% (likely).

Risk impact: 60 reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is \$14.00, the overall cost (impact) to develop the components would be $18 \times 100 \times 14 = \$25,200$.

Risk exposure: $RE = 0.80 \times 25,200 \sim \$20,200$. Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cut-off in the risk table) can provide a means for adjusting the final cost estimate for a project.

It can also be used to predict the probable increase in staff resources required at various points during the project schedule.

The risk projection and analysis techniques are applied iteratively as the software project proceeds. The project team should re-visit the risk table at regular intervals, re-evaluating each risk to determine when new circumstances cause its probability and impact to change. As a consequence of this activity, it may be necessary to add new risks to the table, remove some risks that are no longer relevant, and change the relative positions of still others.

Risk Assessment

At this point in the risk management process, we have established a set of triplets of the form:

$$[r_i, l_i, x_i]$$

Where r_i is risk, l_i is the likelihood (probability) of the risk, and x_i is the impact of the risk. During risk assessment, we further examine the accuracy of the estimates that were made during risk projection, attempt to rank the risks that have been uncovered, and begin thinking about ways to control and/or avert risks that are likely to occur.

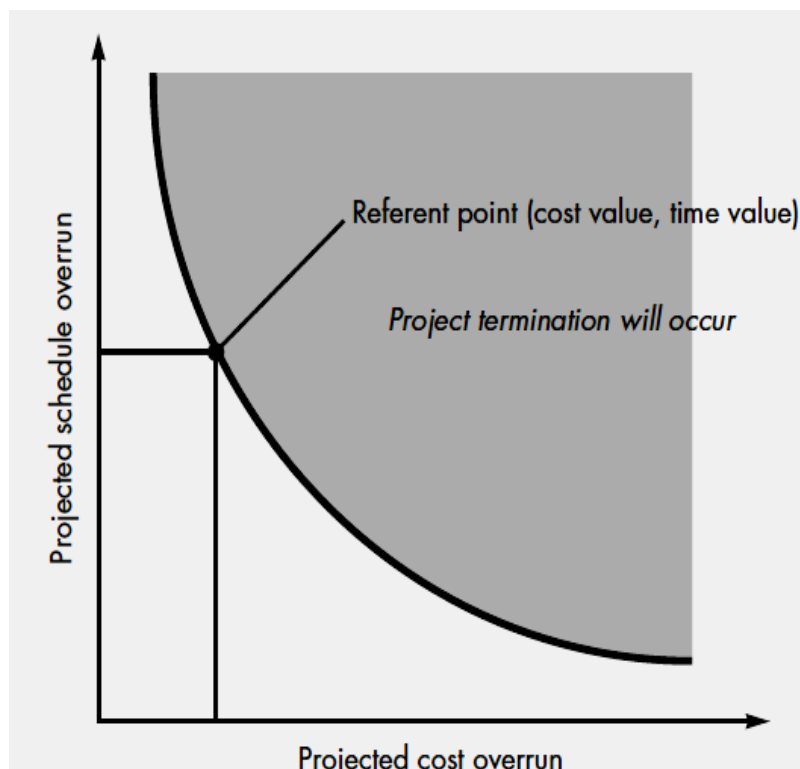


Figure 3.3: Risk referent level

For assessment to be useful, a risk referent level must be defined. For most software projects, the risk components discussed earlier—performance, cost, support, and schedule—also represent risk referent levels. That is, there is a level for performance degradation, cost

overrun, support difficulty, or schedule slippage (or any combination of the four) that will cause the project to be terminated. If a combination of risks create problems that cause one or more of these referent levels to be exceeded, work will stop. In the context of software risk analysis, a risk referent level has a single point, called the referent point or break point, as shown in figure 3.3, at which the decision to proceed with the project or terminate it (problems are just too great) are equally weighted. In reality, the referent level can rarely be represented as a smooth line on a graph. In most cases it is a region in which there are areas of uncertainty; that is, attempting to predict a management decision based on the combination of referent values is often impossible.

Therefore, during risk assessment, we perform the following steps:

- a. Define the risk referent levels for the project.
- b. Attempt to develop a relationship between each (r_i, l_i, x_i) and each of the referent levels.
- c. Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.
- d. Try to predict how compound combinations of risks will affect a referent level.

Check Your Progress 3.

What is risk impact?

3.7 RISK REFINEMENT

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

One way to do this is to represent the risk in condition-transition-consequence (CTC) format. That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted, we can write:

Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

This general condition can be refined in the following manner:

Sub-condition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Sub-condition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Sub-condition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

The consequences associated with these refined sub-conditions remains the same (i.e., 30 percent of software components must be customer engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

Check Your Progress 4.

What is CTC format?

3.8 Risk Mitigation, Monitoring and Management

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues:

- Risk avoidance
- Risk monitoring
- Risk management and contingency planning

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for risk mitigation. For example, assume that high staff turnover is noted as a project risk, r_1 . Based on past history and management intuition, the likelihood, l_1 , of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact, x_1 , is projected at level 2. That is, high turnover will have a critical impact on project cost and schedule.

To *mitigate* this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, and competitive job market).
- Mitigate those causes that are under our control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.

- Organize project teams so that information about each development activity is widely dispersed.
- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed").
- Assign a backup staff member for every critical technologist.

As the project proceeds, risk **monitoring** activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the following factors can be monitored:

- General attitude of team members based on project pressures.
- The degree to which the team has jelled.
- Interpersonal relationships among team members.
- Potential problems with compensation and benefits.
- The availability of jobs within the company and outside it.

In addition to monitoring these factors, the project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of documentation standards and mechanisms to be sure that documents are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor documents carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk **management** and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well underway and a number of people announce that they will be leaving. If the mitigation Strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, the project manager may temporarily refocus resources to those functions that are fully staffed, enabling newcomers who must be added to the team to "get up to speed." Those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode." This might include video-based knowledge capture, the development of "commentary documents," and/or meeting with other team members who will remain on the project.

It is important to note that RMMM steps incur additional project cost. For example, spending the time to "backup" every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, the project planner performs a classic cost/benefit analysis. If a risk aversion step for high turnover will increase both project cost and duration by an estimated 15 percent but the predominant cost factor is "backup," management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent management will likely put all into place.

For a large project, 30 or 40 risks may identify. If between three and seven risk management steps are identified for each, risk management may become a project in itself! For this reason, we adapt the Pareto 80–20 rule to software risk. Experience indicates that 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help the planner to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure). For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don't fall into the critical 20 percent (the risks with highest project priority).

Check Your Progress 5.

What are the three phases of Risk management?

3.9 SUMMARY

This section covers about risk, its analysis, and strategies to manage it. Although technical issues are a primary concern both early on and throughout all project phases, risk management must consider both internal and external sources for cost, schedule, and technical risk. Early and aggressive detection of risk is important because it is typically easier, less costly, and less disruptive to make changes and correct work efforts during the earlier, rather than the later, phases of the project.

Risk management can be divided into three parts: defining a risk management strategy; identifying and analyzing risks; and handling identified risks, including the implementation of risk mitigation plans when needed. At last discuss about the RMMM.

3.10 EXERCISE

- 1) What Is Risk? What is Risk management mean? What are the factors that lead to Risk?
- 2) Explain elaborately the various strategies and steps involved in risk management
- 3) What are four impacts of the project risk?
- 4) Give the Important characteristics of the risk management?
- 5) What are the three phases of Risk management? Explain them.
- 6) What are the ways of identifying the potential risks?
- 7) Define the various steps under risk analysis.
- 8) What Is Risk mitigation, Monitoring and Management Plan?

Software Engineering

THIRD - BLOCK

BLOCK

3

UNIT 1 Software Quality Assurance

UNIT 2 Software Configuration Management

UNIT 3 Analysis Concepts and Principles

Overview

In this section we discuss the overview of this block's content. This block consists of the following units:

Unit 1 Software Quality Assurance

Software Quality Assurance encompasses the entire software development life cycle and the goal is to ensure that the development and/or maintenance processes are continuously improved to produce products that meet specifications/requirements.

The process of Software Quality Control (SQC) is also governed by Software Quality Assurance (SQA).

Unit 2 Software Configuration Management

The purpose of Software Configuration Management is to establish and maintain the integrity of the products of the software project throughout the project's software life cycle. Software Configuration Management involves identifying configuration items for the software project, controlling these configuration items and changes to them, and recording and reporting status and change activity for these configuration items

Unit 3 Analysis Concepts and Principles

Requirements analysis allows the software engineer to refine the software allocation and build models of the data, functional, and behavioural domains that will be treated by software. And the software requirements specification provides the developer and the customer with the means to assess quality once software is built.

UNIT-1 SOFTWARE QUALITY ASSURANCE

1.0 Introduction

1.1 Objective

1.2 Principle of Software Quality Assurance

1.3 Basic concept of Quality

1.4 Quality Control

1.5 Quality Assurance

1.6 Cost of Quality

1.7 Software Review

1.8 Formal Technique review

1.9 Software Reliability

1.10 Summary

1.11 Exercise

1.0 INTRODUCTION

It's not enough to talk the talk by saying that software quality is important, we should:

- i. Explicitly define what is meant when you say “software quality,”
- ii. Create a set of activities that will help ensure that every software engineering work product exhibits high quality,
- iii. Perform quality assurance activities on every software project,
- iv. Use metrics to develop strategies for improving your software process and, as a consequence, the quality of the end product.

1.1 OBJECTIVE

The various objectives of SQA are as follows:

- a) Quality management approach.
- b) Measurement and reporting mechanisms.
- c) Effective software-engineering technology.
- d) A procedure to assure compliance with software-development standards where applicable.
- e) A multi-testing strategy is drawn.
- f) Formal technical reviews that are applied throughout the software process.

1.2 PRINCIPLE OF SOFTWARE QUALITY ASSURANCE

For our purposes, software quality is defined as

“Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.”

There is little question that this definition could be modified or extended. In fact, a definitive definition of software quality could be debated endlessly. The definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.

3. A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

A Software Quality Assurance Plan is created to define a software team's SQA strategy. During analysis, design, and code generation, the primary SQA work product is the formal technical review summary report. During duplicate testing, test plans and procedures are produced. Other work products associated with process improvement may also be generated.

SQA encompasses-

1. A quality management approach
2. Effective software engineering technology (methods and tools)
3. Formal technical reviews that are applied throughout the software process
4. A multi-tiered testing strategy
5. Control of software documentation and the changes made to it
6. A procedure to ensure compliance with software development standards
7. Measurement and reporting mechanisms.

Software quality is:

1. The degree to which a system, component, or process meets specified requirements.
2. The degree to which a system, component, or process meets customer or user needs or expectations.

Software quality assurance is:

1. A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.
2. A set of activities designed to evaluate the process by which the products are developed or manufactured. Contrast with quality control.

SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality by applying solid technical methods and measures, conducting formal technical reviews, and performing well-planned software testing.

- i. ***Prepares an SQA plan for a project.*** The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies
 - evaluations to be performed
 - audits and reviews to be performed
 - standards that are applicable to the project
 - procedures for error reporting and tracking
 - documents to be produced by the SQA group
 - amount of feedback provided to the software project team
- ii. ***Participates in the development of the project's software process description.*** The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.
- iii. ***Reviews software engineering activities to verify compliance with the defined software process.*** The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.
- iv. ***Audits designated software work products to verify compliance with those defined as part of the software process.*** The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.
- v. ***Ensures that deviations in software work and work products are documented and handled according to a documented procedure.*** Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.
- vi. ***Records any noncompliance and reports to senior management.*** Noncompliance items are tracked until they are resolved.

The objectives of SQA activities

Software development (process-oriented):

- i. Assuring an acceptable level of confidence that the software will conform to functional technical requirements.
- ii. Assuring an acceptable level of confidence that the software will conform to managerial scheduling and budgetary requirements.

- iii. Initiating and managing of activities for the improvement and greater efficiency of software development and SQA activities. This means improving the prospects that the functional and managerial requirements will be achieved while reducing the costs of carrying out the software development and SQA activities.

Software maintenance (product-oriented):

- i. Assuring with an acceptable level of confidence that the software maintenance activities will conform to the functional technical requirements.
- ii. Assuring with an acceptable level of confidence that the software maintenance activities will conform to managerial scheduling and budgetary requirements.
- iii. Initiating and managing activities to improve and increase the efficiency of software maintenance and SQA activities. This involves improving the prospects of achieving functional and managerial requirements while reducing costs.

Elements of SQA

- Standards
- Reviews and Audits
- Testing
- Error/defect collection and analysis
- Change management
- Education
- Vendor management
- Security management
- Safety
- Risk management

SQA Goals

- ***Requirements quality.*** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow.
- ***Design quality.*** Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements.

- **Code quality.** Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability.
- **Quality control effectiveness.** A software team should apply limited resources in a way that has the highest Likelihood of achieving a high quality result.
- **ISO 9001:2000 Standard** ISO 9001:2000 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. The requirements delineated by ISO 9001:2000 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques.

Check Your Progress 1.

What are the activities associated with SQA group?

1.3 BASIC CONCEPT OF QUALITY

The American Heritage Dictionary defines quality as “a characteristic or attribute of something.” As an attribute of an item, quality refers to measurable characteristics- things we are able to compare to known standards such as length, colour, electrical properties, and malleability. However, software, largely an intellectual entity, is more challenging to characterize than physical objects.

Nevertheless, measures of a program’s characteristics do exist. These *properties* include cyclomatic complexity, cohesion, number of function points, lines of code, and many others.

When we examine an item based on its measurable characteristics, two kinds of quality may be encountered: quality of design and quality of conformance.

Quality of design refers to the characteristics that designers specify for an item. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases, if the product is manufactured according to specifications.

Quality of conformance is the degree to which the design specifications are followed during manufacturing. Again, the greater the degree of conformance, the higher is the level of quality of conformance.

In software development, quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.

But are quality of design and quality of conformance the only issues that software engineers must consider? Robert Glass argues that a more “intuitive” relationship is in order:

$$\text{User satisfaction} = \text{compliant product} + \text{good quality} + \\ \text{Delivery within budget and schedule}$$

At the bottom line, Glass contends that quality is important, but if the user isn't satisfied, nothing else really matters. DeMarco reinforces this view when he states: “A product's quality is a function of how much it changes the world for the better.” This view of quality contends that if a software product provides substantial benefit to its end-users, they may be willing to tolerate occasional reliability or performance problems.

Check Your Progress 2.

What are the measures of software quality?

1.4 QUALITY CONTROL

Variation control may be equated to quality control. But how do we achieve quality control? Quality control involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it. Quality control includes a feedback loop to the process that created the work product. The combination of measurement and feedback allows us to tune the process when the work products created fail to meet their specifications.

This approach views quality control as part of the manufacturing process. Quality control activities may be fully automated, entirely manual, or a combination of automated tools and human interaction. A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects produced.

Check Your Progress 3.

What is Software Quality Control?

1.5 QUALITY ASSURANCE

Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

Check Your Progress 4.

What is the need of quality assurance?

1.6 COST OF QUALITY

The cost of quality includes all costs incurred in the pursuit of quality or in performing quality-related activities. Cost of quality studies are conducted to provide a baseline for the current cost of quality, identify opportunities for reducing the cost of quality, and provide a normalized basis of comparison. The basis of normalization is almost always dollars. Once we have normalized quality costs on a dollar basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the effect of changes in dollar-based terms. Quality costs may be divided into costs associated with prevention, appraisal, and failure.

Prevention costs include

- quality planning
- formal technical reviews
- test equipment
- training

Appraisal costs include activities to gain insight into product condition the “first time through” each process. Examples of appraisal costs include:

- in-process and inter-process inspection

- equipment calibration and maintenance
- testing

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. Internal failure costs are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

- rework
- repair
- failure mode analysis

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are

- complaint resolution
- product return and replacement
- help line support
- warranty work

Check Your Progress 5.

What are the components of the Cost of Quality?

1.7 SOFTWARE REVIEW

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities that we have called analysis, design, and coding.

Freedman and Weinberg describe the need for reviews this way:

“Technical work needs reviewing for the same reason that pencils need erasers: To err is human. The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else.”

A review - any review - is a way of using the diversity of a group of people to:

- i. Point out needed improvements in the product of a single person or team;

- ii. Confirm those parts of a product in which improvement is either not desired or not needed;
- iii. Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of review. A formal technical review is the most effective filter from a quality assurance standpoint; Conducted by software engineers (and others) for software engineers.

Check Your Progress 6.

What is the need to review software and when review is required?

1.8 FORMAL TECHNICAL REVIEWS

A formal technical review is a software quality assurance activity performed by software engineers (and others). *The objectives of the FTR are*

- i. To uncover errors in function, logic, or implementation for any representation of the software
- ii. To verify that the software under review meets its requirements
- iii. To ensure that the software has been represented according to predefined standards
- iv. To achieve software that is developed in a uniform manner
- v. To make projects more manageable

In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen. The FTR is actually a class of reviews that includes walkthroughs, inspections, round-robin reviews and other small group technical assessments of software. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough are presented as a representative formal technical review.

The Review Meeting

Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components. By narrowing focus, the FTR has a higher likelihood of uncovering errors.

Review Reporting and Record Keeping

During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed.

A review summary report answers three questions:

- i.** What was reviewed?
- ii.** Who reviewed it?
- iii.** What were the findings and conclusions?

The review summary report is a single page form. It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

Review Guidelines

Guidelines for the conduct of formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed. A review that is uncontrolled can often be worse than no review at all. The following represents a minimum set of guidelines for formal technical reviews:

- i.** Review the product, not the producer
- ii.** Set an agenda and maintain it
- iii.** Limit debate and rebuttal
- iv.** Enunciate problem areas
- v.** Take written notes
- vi.** Limit the number of participants and insist upon advance preparation
- vii.** Develop a checklist for each product that is likely to be reviewed
- viii.** Allocate resources and schedule time for FTRs

- ix. Conduct meaningful training for all reviewers
- x. Review your early reviews

Check Your Progress 7.

What are the differences between reviews and formal technical reviews?

1.9 SOFTWARE RELIABILITY

There is no doubt that the reliability of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are acceptable.

Software reliability, unlike many other quality factors, can be measured directly and estimated using historical and developmental data. Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time". To illustrate, program X is estimated to have a reliability of 0.96 over eight elapsed processing hours. In other words, if program X were to be executed 100 times and require eight hours of elapsed processing time, it is likely to operate correctly (without failure) 96 times out of 100.

Whenever software reliability is discussed, a pivotal question arises: What is meant by the term failure? In the context of any discussion of software quality and reliability, failure is non-conformance to software requirements. Yet, even within this definition, there are gradations. Failures can be only annoying or catastrophic. One failure can be corrected within seconds while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

Measures of Reliability and Availability

Early work in software reliability attempted to extrapolate the mathematics of hardware reliability theory to the prediction of software reliability. Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear does not enter into the picture.

There has been debate over the relationship between key concepts in hardware reliability and their applicability to software. Although an irrefutable link has yet be established, it is worthwhile to consider a few simple concepts that apply to both system elements. If we consider a computer-based system, a simple measure of reliability is meantime-between-failure (MTBF), where

$$MTBF = MTTF + MTTR$$

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively. Many researchers argue that MTBF is a far more useful measure than defects/KLOC or defects/FP. stated simply, an end-user is concerned with failures, not with the total error count. Because each error contained within a program does not have the same failure rate, the total error count provides little indication of the reliability of a system. For example, consider a program that has been in operation for 14 months. Many errors in this program may remain undetected for decades before they are discovered. The MTBF of such obscure errors might be 50 or even 100 years. Other errors, as yet undiscovered, might have a failure rate of 18 or 24 months. Even if every one of the first category of errors (those with long MTBF) is removed, the impact on software reliability is negligible.

In addition to a reliability measure, we must develop a measure of availability. Software availability is the probability that a program is operating according to requirement sat a given point in time and is defined as

$$Availability = [MTTF / (MTTF + MTTR)] \times 100\%$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

Check Your Progress 8.

How we can measure the reliability and availability of any software?

1.10 SUMMARY

This chapter covers a brief discussion about Software Quality, and its role, everyone involved in the software engineering process is responsible for quality. If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do. That results in lower costs, and more importantly, improved time-to-market. Before software

quality assurance activities can be initiated, it is important to define ‘software quality’ at a number of different levels of abstraction. Once you understand what quality is, a software team must identify a set of SQA activities that will filter errors out of work products before they are passed on.

1.11 EXERCISE

- 1) How do we define Software Quality? What is Software Quality Control? What are the components of the Cost of Quality?
- 2) What is Software Quality Assurance? What activities are required to perform SQA?
- 3) Define the terms:
 - (a) Quality of Design
 - (b) Quality of Conformance
- 4) What are the factors of Software Quality? Define.
- 5) What is the role of Software Quality in software?
- 6) What is Software Reliability? How can we Measure Reliability and Availability?
- 7) Explain the objective of SQA activities.
- 8) Define the Software Reviews and Formal Technical Review. Are both terms the same or different with each other? Explain.

UNIT-2 SOFTWARE CONFIGURATION MANAGEMENT

- 2.0** Introduction
- 2.1** Objective
- 2.2** Principle of Software Configuration Management
- 2.3** Baseline of SCM
- 2.4** Software Configuration items
- 2.5** SCM process
- 2.6** Version Control
- 2.7** Change Control
- 2.8** Configuration Audit
- 2.9** Status Reporting
- 2.10** Summary
- 2.11** Exercise

2.0 INTRODUCTION

Software configuration management (SCM) is a set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Change is inevitable when computer software is built. And change increases the level of confusion among software engineers who are working on a project. Confusion arises when changes are not analysed before they are made, recorded before they are implemented, reported to those with a need to know, or controlled in a manner that will improve quality and reduce error.

2.1 OBJECTIVE

The objectives of this unit are:

- a) Continuous control of product specifically built situations
- b) Improvement of quality
- c) Active monitoring of changes instead of being driven by changes
- d) Cost-effective project management
- e) Accurate definition of items affected by a change (i.e. design documents, contracts, parts and tools)
- f) Complete design requirements per end-product
- g) Traceability between multi-level contract changes
- h) Elimination of data duplication, allowing for segregated data responsibility
- i) Continuous recording of the product specific design/build situation and deviation reporting

2.2 PRINCIPLE OF SOFTWARE CONFIGURATION MANAGEMENT

As per Babich:

“Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.”

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to,

- i. Identify change

- ii. Control change,
- iii. Ensure that change is being properly implemented, and
- iv. Report changes to others who may have an interest.

It is important to make a clear distinction between software support and software configuration management. Support is a set of software engineering activities that occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that begin when a software engineering project begins and terminate only when the software is taken out of operation.

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made.

The output of the software process is information that may be divided into three broad categories:

- i. Computer programs (both source level and executable forms)
- ii. Documents that describe the computer programs (targeted at both technical practitioners and users)
- iii. Data (contained within the program or external to it)

The items that comprise all information produced as part of the software process are collectively called a software configuration.

As the software process progresses, the number of software configuration items (SCIs) grows rapidly. A System Specification spawns a Software Project Plan and Software Requirements Specification. These in turn spawn other documents to create a hierarchy of information. If each SCI simply spawned other SCIs, little confusion would result. Unfortunately, another variable enters the process—change. Change may occur at any time, for any reason.

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of change:

- New business or market conditions dictate changes in product requirements or business rules.
- New customer needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

Check Your Progress 1.

What is the Origin of changes that are requested for software?

2.3 BASELINE OF SCM

Change is a fact of life in software development. Customers want to modify requirements. Developers want to modify the technical approach. Managers want to modify the project strategy. Why all this modification? The answer is really quite simple. As time passes, all constituencies know more (about what they need, which approach would be best, how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: Most changes are justified!

A **baseline** is a software configuration management concept that helps us to control change without seriously impeding justifiable change.

The IEEE defines a baseline as:

“A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.”

One way to describe a baseline is through analogy: Consider the doors to the kitchen in a large restaurant. One door is marked OUT and the other is marked IN. The doors have stops that allow them to be opened only in the appropriate direction.

If a waiter picks up an order in the kitchen, places it on a tray and then realizes he has selected the wrong dish, he may change to the correct dish quickly and informally before he leaves the kitchen.

If, however, he leaves the kitchen, gives the customer the dish and then is informed of his error, he must follow a set procedure:

- i. Look at the check to determine if an error has occurred
- ii. Apologize profusely
- iii. Return to the kitchen through the in-door
- iv. Explain the problem, and so forth

A baseline is analogous to the kitchen doors in the restaurant. Before a software configuration item becomes a baseline, change may be made quickly and informally. However, once a

baseline is established, we figuratively pass through a swinging one way door. Changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change. In the context of software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review.

Check Your Progress 2.

What is the need for baseline?

2.4 SOFTWARE CONFIGURATION ITEMS

We have already defined a software configuration item as information that is created as part of the software engineering process. In the extreme, a SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. More realistically, an SCI is a document, an entire suite of test cases, or a named program component (e.g., a C++ function or an Ada package). In addition to the SCIs that are derived from software work products; many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, and other CASE tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare, it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be base lined as part of a comprehensive configuration management process.

In reality, SCIs are organized to form configuration objects that may be catalogued in the project database with a single name. A configuration object has a name, attributes, and is "connected" to other objects by relationships. Referring to figure 2.1, the configuration objects, Design Specification, data model, component N, source code and Test Specification are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is, data model and component N are part of the object Design Specification. A double-headed straight arrow indicates an interrelationship. If a change were made to the source code object, the

interrelationships enable a software engineer to determine what other objects (and SCIs) might be affected.

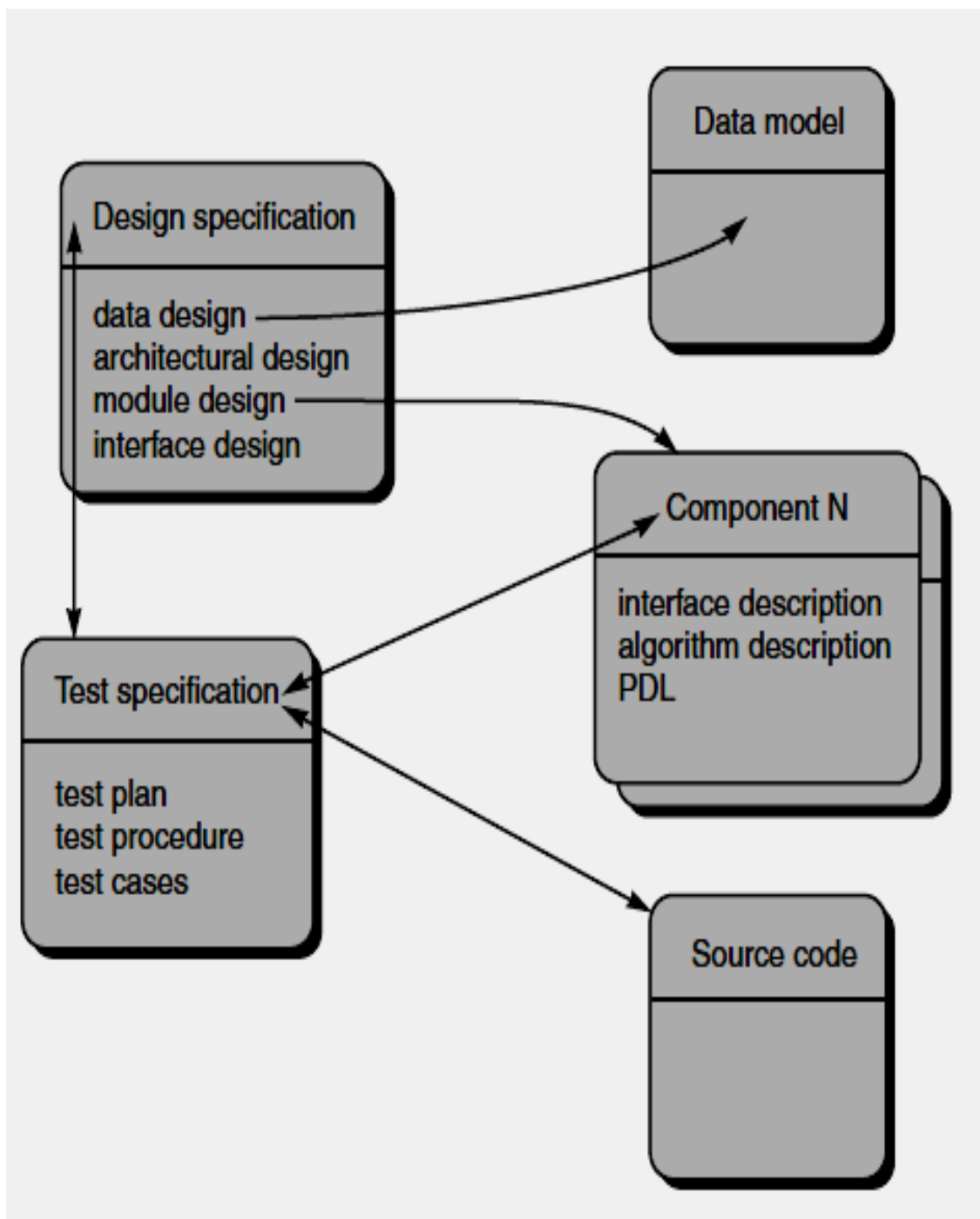


Figure 2.1: Configuration Objects

Check Your Progress 3.

What to identify as configuration items and how?

2.5 SCM PROCESS

Software configuration management is an important element of software quality assurance. Its primary responsibility is the control of change. However, SCM is also responsible for the identification of individual SCIs and various versions of the software, the auditing of the software configuration to ensure that it has been properly developed, and the reporting of all changes applied to the configuration.

Any description of SCM introduces a set of complex questions:

- How does an organization identify and manage the many existing versions of a program in a manner that will enable change to be accommodated efficiently?
- How an organization control changes does before and after software is released to a customer?
- Who has responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to apprise others of changes that are made?

These questions lead us to the definition of five SCM tasks: identification, version control, and change control, configuration auditing, and reporting.

To control and manage software configuration items, each must be separately named and then organized using an object-oriented approach.

Two types of objects can be identified: *basic objects and aggregate objects*. A basic object is a "unit of text" that has been created by a software engineer during analysis, design, code, or test. For example, a basic object might be a section of a requirements specification, a source listing for a component, or a suite of test cases that are used to exercise the code. An aggregate object is a collection of basic objects and other aggregate objects.

Conceptually, it can be viewed as a named (identified) list of pointers that specify basic objects such as data model and component N. Each object has a set of distinct features that identify it uniquely: a name, a description, a list of resources, and a "realization." The object name is a character string that identifies the object unambiguously.

Check Your Progress 4.

What are the Objectives of SCM Process?

2.6 VERSION CONTROL

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process.

Clemm describes version control in the context of SCM:

“Configuration management allows a user to specify alternative configurations of the software system through the selection of appropriate versions. This is supported by associating attributes with each software version, and then allowing a configuration to be specified by describing the set of desired attributes.”

These "attributes" mentioned can be as simple as a specific version number that is attached to each object or as complex as a string of Boolean variables (switches) that indicate specific types of functional changes that have been applied to the system.

One representation of the different versions of a system is the evolution graph presented in figure 2.2. Each node on the graph is an aggregate object, that is, a complete version of the software. Each version of the software is a collection of SCIs (source code, documents, data), and each version may be composed of different variants. To illustrate this concept, consider a version of a simple program that is composed of entities 1, 2, 3, 4, and 5. Entity 4 is used only when the software is implemented using colour displays. Entity 5 is implemented when monochrome displays are available.

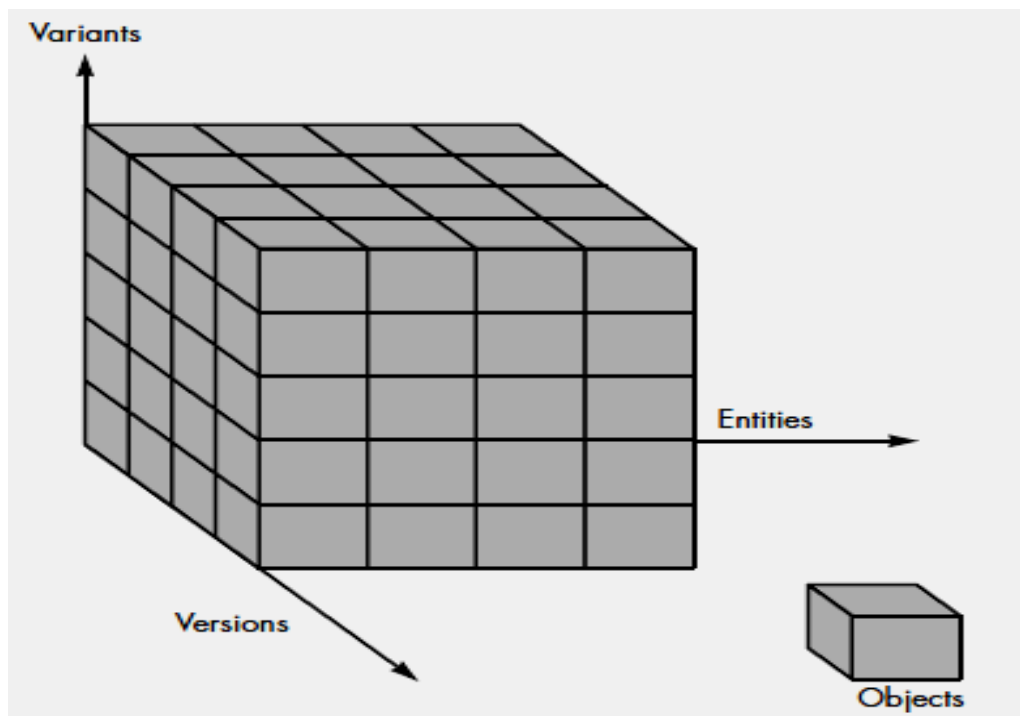


Figure 2.2: Object pool representation of components, variants, and versions

Therefore, two variants of the version can be defined:

- (1) Entities 1, 2, 3, and 4;
- (2) Entities 1, 2, 3, and 5.

To construct the appropriate variant of a given version of a program, each entity can be assigned an "attribute-tuple"—a list of features that will define whether the entity should be used when a particular variant of a software version is to be constructed. One or more attributes is assigned for each variant. For example, a colour attribute could be used to define which entity should be included when colour displays are to be supported.

Another way to conceptualize the relationship between entities, variants and versions (revisions) is to represent them as an object pool. Referring to figure, the relationship between configuration objects and entities, variants and versions can be represented in a three-dimensional space. An entity is composed of a collection of objects at the same revision level. A variant is a different collection of objects at the same revision level and therefore coexists in parallel with other variants.

A new version is defined when major changes are made to one or more objects. A number of different automated approaches to version control have been proposed over the past decade. The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

2.7 CHANGE CONTROL

The reality of change control in a modern software engineering context has been summed up beautifully by James Bach:

“Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could effectively discourage them from doing creative work.”

Bach recognizes that we face a balancing act. Too much change control and we create problems. Too little, and we create other problems.

For a large software engineering project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change. A change request is submitted and evaluated to assess

technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a change report, which is used by a *Change Control Authority (CCA)* - a person or group who makes a final decision on the status and priority of the change. An Engineering Change Order (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit. The object to be changed is "checked out" of the project database, the change is made, and appropriate SQA activities are applied. The object is then "checked in" to the database and appropriate version control mechanisms are used to create the next version of the software.

The "check-in" and "check-out" process implements two important elements of change control - access control and synchronization control. Access control governs which software engineers have the authority to access and modify a particular configuration object. Synchronization control helps to ensure that parallel changes, performed by two different people, don't overwrite one another. Access and synchronization control flow are illustrated schematically in figure 2.3.

Based on an approved change request and ECO, software engineers check-out a configuration object.

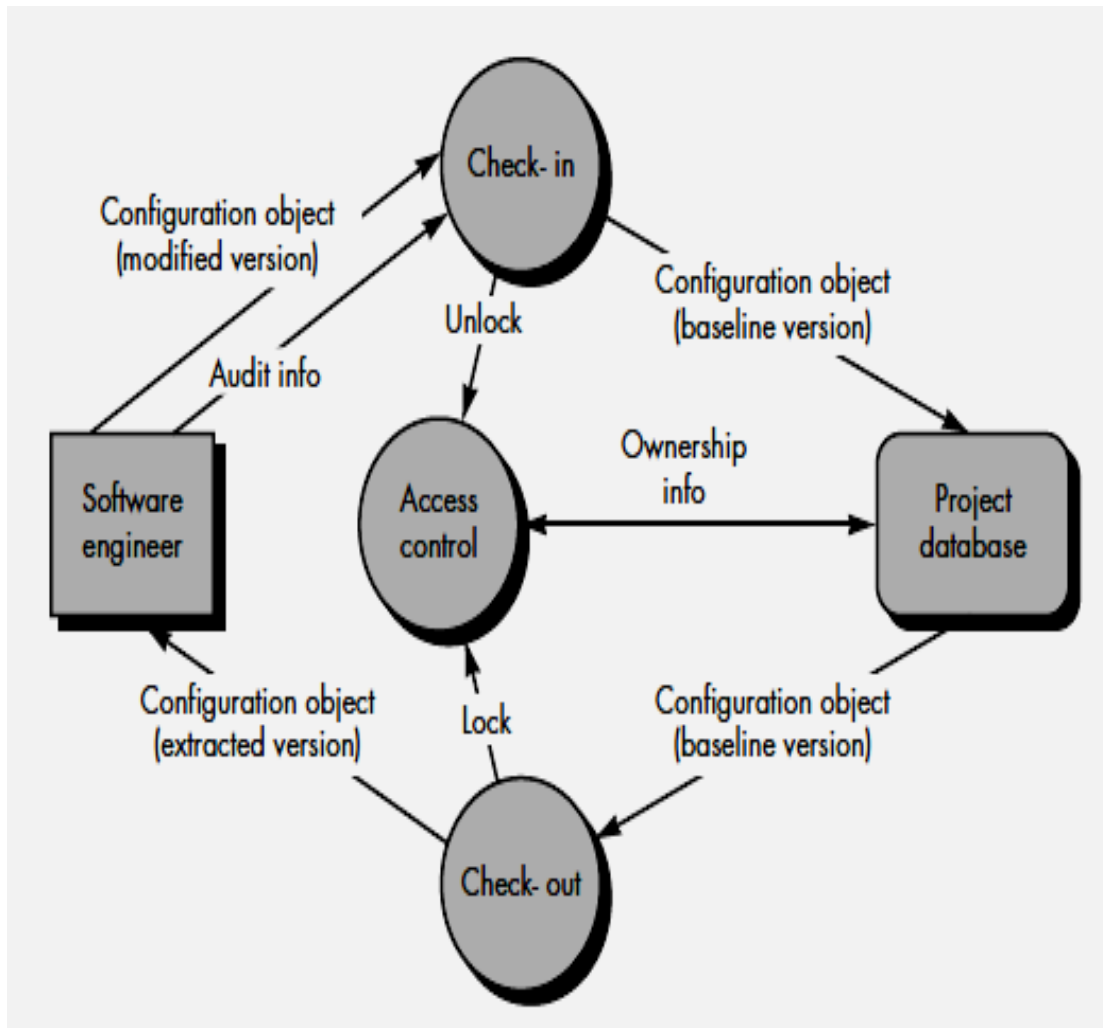


Figure 2.3: Access and synchronization control

An access control function ensures that the software engineer has authority to check out the object, and synchronization control locks the object in the project database so that no updates can be made to it until the currently checked-out version has been replaced. Note that other copies can be checked-out, but other updates cannot be made. A copy of the base lined object, called the extracted version, is modified by the software engineer. After appropriate SQA and testing, the modified version of the object is checked in and the new baseline object is unlocked.

Prior to an SCI becoming a baseline, only informal change control need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements. Once the object has undergone formal technical review and has been approved, a baseline is created. Once an SCI becomes a baseline, project level change control is implemented. Now, to make a change, the developer must gain approval from the project manager or from the CCA if the change affects other

SCIs. In some cases, formal generation of change requests, change reports, and ECOs is dispensed with. However, assessment of each change is conducted and all changes are tracked and reviewed.

When the software product is released to customers, formal change control is instituted. The formal change control procedure has been outlined in figure 2.4

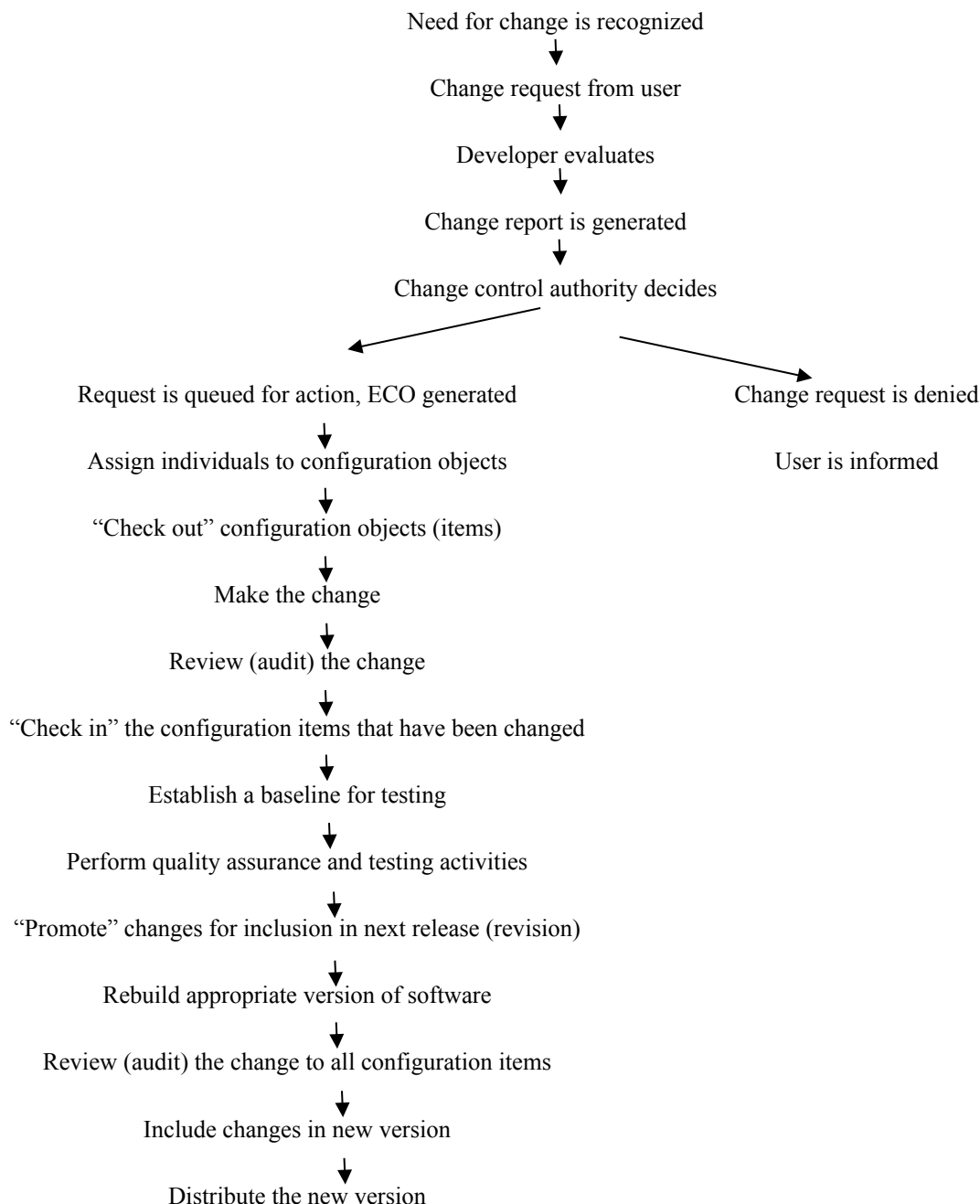


Figure 2.4: The Change Process

The change control authority plays an active role in the second and third layers of control. Depending on the size and character of a software project, the CCA may be composed of one

person—the project manager - or a number of people. The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question.

Check Your Progress 5.

What is the difference between version control and change control?

2.8 CONFIGURATION AUDIT

Identification, version control, and change control help the software developer to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can we ensure that the change has been properly implemented? The answer is twofold: (1) formal technical reviews and (2) the software configuration audit.

The formal technical review focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side effects. A formal technical review should be conducted for all but the most trivial changes. A software configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review.

The audit asks and answers the following questions:

- i.** Has the change specified in the ECO been made? Have any additional modifications been incorporated?
- ii.** Has a formal technical review been conducted to assess technical correctness?
- iii.** Has the software process been followed and have software engineering standards been properly applied?
- iv.** Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?
- v.** Have SCM procedures for noting the change, recording it, and reporting it been followed?
- vi.** Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a formal technical review. However, when SCM is a formal activity, the SCM audit is conducted separately by the quality assurance group.

Check Your Progress 6.

What are the requirements of internal auditing?

2.9 STATUS REPORTING

Configuration status reporting (sometimes called status accounting) is an SCM task that answers the following questions:

- i. What happened?
- ii. Who did it?
- iii. When did it happen?
- iv. What else will be affected?

Each time an SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA, a CSR entry is made. Each time a configuration audit is conducted, the results are reported as part of the CSR task. Output from CSR may be placed in an on-line database, so that software developers or maintainers can access change information by keyword category.

In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners appraised of important changes.

Configuration status reporting plays a vital role in the success of a large software development project. When many people are involved, it is likely that "the left hand not knowing what the right hand is doing" syndrome will occur. Two developers may attempt to modify the same SCI with different and conflicting intents. A software engineering team may spend months of effort building software to an obsolete hardware specification. The person who would recognize serious side effects for a proposed change is not aware that the change is being made. CSR helps to eliminate these problems by improving communication among all people involved.

Check Your Progress 7.

How to write effective weekly status report?

2.10 SUMMARY

SCM is the process that defines how to control and manage change.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process.

The need for an SCM process is acutely felt when there are many developers and many versions of the software. Suffice to say that in a complex scenario where bug fixing should happen on multiple production systems and enhancements must be continued on the main code base, SCM acts as the backbone which can make this happen.

This chapter covers about the SCM, its objectives, features, Baseline, SCM activities, Software Configuration Items Configuration Audit and Status Report.

2.11 EXERCISE

- 1) What is SCM? What are the Features supported by SCM?
- 2) What is SCM Process? What are the Objectives of SCM Process?
- 3) List the SCM Activities.
- 4) Define distinction between SCM and Software Support.
- 5) Describe the various Software Configuration Management Tasks.
- 6) Explain Software Configuration Item
- 7) What Is Base line criteria in SCM? Also write its role in SCM.
- 8) Define configuration Audit and Status Reporting?
- 9) Define the Version Control and Change Control in detail.

UNIT-3 ANALYSIS CONCEPTS AND PRINCIPLES

3.0 Introduction

3.1 Objective

3.2 Analysis Concepts and Principles

3.3 Requirement Elicitation for Software analysis principles

3.4 The Information Domain

3.5 Modelling

3.6 Partitioning

3.7 Essential and Implementation Views

3.8 Specification

3.9 Specification Principles

3.10 Representation

3.11 The Software Requirement Specification

3.12 Summary

3.13 Exercise

3.0 INTRODUCTION

Requirements analysis provides the software designer with a representation of information, function, and behaviour that can be translated to data, architectural, interface, and component-level designs.

Initially, the analyst studies the System Specification and the Software Project Plan. It is important to understand software in a system context and to review the software scope that was used to generate planning estimates. Next, communication for analysis must be established so that problem recognition is ensured. The goal is recognition of the basic problem elements as perceived by the customer/users.

3.1 OBJECTIVE

The objectives of this unit are:

- a) *to understand the problem before beginning to create the analysis model*
- b) *to develop prototypes to help user to understand how human-machine interactions*
- c) *record the origin of and the reasons for every requirement*
- d) *use multiple views of requirements*
- e) *prioritize requirements*
- a) *work to eliminate ambiguity*
- f) to explain about the User interface design.
- g) to introduce the concept of data acquisition system.
- h) to know about the monitoring and control system and defining to implement them.

3.2 ANALYSIS CONCEPTS AND PRINCIPLES

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design as shown in figure 3.1. Requirements engineering activities result in the specification of software's operational characteristics (function, data, and behaviour), indicate software's interface with other system elements, and establish constraints that software must meet.

Software requirements analysis may be divided into five areas of effort:

- i. Problem recognition
- ii. Evaluation and synthesis
- iii. Modelling
- iv. Specification

v. Review

Problem evaluation and solution synthesis is the next major area of effort for analysis. The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behaviour in the context of events that affect the system, establish system interface characteristics, and uncover additional design constraints. Each of these tasks serves to describe the problem so that an overall approach or solution may be synthesized.

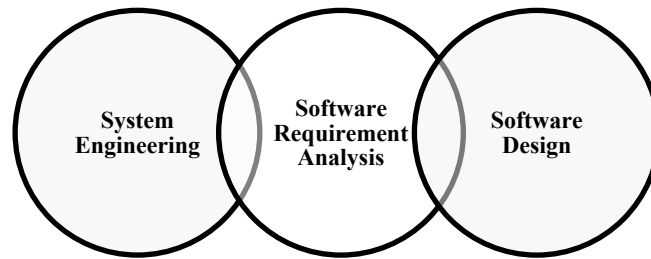


Figure 3.1: Analysis as a bridge between system engineering and software design

Once problems have been identified, the analyst determines what information is to be produced by the new system and what data will be provided to the system.

Upon evaluating current problems and desired information (input and output), the analyst begins to synthesize one or more solutions. To begin, the data objects, processing functions, and behaviour of the system are defined in detail. Once this information has been established, basic architectures for implementation are considered.

The process of evaluation and synthesis continues until both analyst and customer feel confident that software can be adequately specified for subsequent development steps.

Throughout evaluation and solution synthesis, the analyst's primary focus is on "what," not "how." What data does the system produce and consume, what functions must the system perform, what behaviours does the system exhibit, what interfaces are defined and what constraints apply?

During the evaluation and solution synthesis activity, the analyst creates models of the system in an effort to better understand data and control flow, functional processing, operational behaviour, and information content. The model serves as a foundation for software design and as the basis for the creation of specifications for the software.

Detailed specifications may not be possible at this stage. The customer may be unsure of precisely what is required. The developer may be unsure that a specific approach will properly

accomplish function and performance. For these, and many other reasons, an alternative approach to requirements analysis, called prototyping, may be conducted.

Check Your Progress 1.

What are the Objectives of Requirement Analysis ?

3.3 REQUIREMENT ELICITATION FOR SOFTWARE ANALYSIS PRINCIPLES

Before requirements can be analysed, modelled, or specified they must be gathered through an elicitation process. A customer has a problem that may be amenable to a computer-based solution. A developer responds to the customer's request for help.

Initiating the Process

The most commonly used requirements elicitation technique is to conduct a meeting or interview. The first meeting between a software engineer (the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents. Neither person knows what to say or ask; both are worried that what they do say will be misinterpreted; both are thinking about where it might lead; both want to get the thing over with, but at the same time, both want it to be a success. Yet, communication must be initiated. The analyst may start by asking context-free questions. That is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself.

The first set of context-free questions focuses on the customer, the overall goals, and the benefits. For example, the analyst might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development. The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution:

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the meeting. Gause and Weinberg call these meta-questions and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Facilitated Application Specification Techniques

Too often, customers and software engineers have an unconscious "us and them" mind-set. Rather than working as a team to identify and refine requirements, each constituency defines its own "territory" and communicates through a series of memos, formal position papers, documents, and question and answer sessions. History has shown that this approach doesn't work very well. Misunderstandings abound, important information is omitted, and a successful working relationship is never established. It is with these problems in mind that a number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification. Called facilitated application specification techniques (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements. FAST has been used predominantly by the information systems community, but the technique offers potential for improved communication in applications of all kinds.

Many different approaches to FAST have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- A meeting is conducted at a neutral site and attended by both software engineers and customers.
- Rules for preparation and participation are established.

- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used.
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

To better understand the flow of events as they occur in a typical FAST meeting, we present a brief scenario that outlines the sequence of events that lead-up to the meeting, occur during the meeting, and follow the meeting.

Initial meetings between the developer and customer occur and basic questions and answers help to establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customer write a one- or two-page "product request." A meeting place, time, and date for FAST are selected and a facilitator is chosen. Attendees from both the development and customer/user organizations are invited to attend. The product request is distributed to all attendees before the meeting date.

ANALYSIS PRINCIPLES

Over the past two decades, a large number of analysis modelling methods have been developed. Investigators have identified analysis problems and their causes and have developed a variety of modelling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view.

However, all analysis methods are related by a set of operational principles:

- The information domain of a problem must be represented and understood.
- The functions that the software is to perform must be defined.
- The behaviour of the software (as a consequence of external events) must be represented.
- The models that depict information function and behaviour must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
- The analysis process should move from essential information toward implementation detail.

By applying these principles, the analyst approaches a problem systematically. The information domain is examined so that function may be understood more completely.

Models are used so that the characteristics of function and behaviour can be communicated in a compact fashion. Partitioning is applied to reduce complexity. Essential and implementation views of the software are necessary to accommodate the logical constraints imposed by processing requirements and the physical constraints imposed by other system elements.

In addition to these operational analysis principles, a set of guiding principles for requirements engineering are:

- Understand the problem before you begin to create the analysis model. There is a tendency to rush to a solution, even before the problem is understood. This often leads to elegant software that solves the wrong problem!
- Develop prototypes that enable a user to understand how human/machine interaction will occur. Since the perception of the quality of software is often based on the perception of the “friendliness” of the interface, prototyping (and the iteration that results) are highly recommended.
- Record the origin of and the reason for every requirement. This is the first step in establishing traceability back to the customer.
- Use multiple views of requirements. Building data, functional, and behavioural models provide the software engineer with three different views. This reduces the likelihood that something will be missed and increases the likelihood that inconsistency will be recognized.
- Rank requirements. Tight deadlines may preclude the implementation of every software requirement. If an incremental process model is applied, those requirements to be delivered in the first increment must be identified.
- Work to eliminate ambiguity. Because most requirements are described in a natural language, the opportunity for ambiguity abounds. The use of formal technical reviews is one way to uncover and eliminate ambiguity.

Check Your Progress 2.

What are the Difficulties in Elicitations?

3.4 THE INFORMATION DOMAIN

All software applications can be collectively called data processing. Interestingly, this term contains a key to our understanding of software requirements. Software is built to process data, to transform data from one form to another; that is, to accept input, manipulate it in some way, and produce output. This fundamental statement of objective is true whether we build batch software for a payroll system or real-time embedded software to control fuel flow to an automobile engine.

It is important to note, however, that software also processes events. An event represents some aspect of system control and is really nothing more than Boolean data—it is either on or off, true or false, there or not there. For example, a pressure sensor detects that pressure exceeds a safe value and sends an alarm signal to monitoring software. The alarm signal is an event that controls the behaviour of the system.

Therefore, data (numbers, text, images, sounds, video, etc.) and control (events) both reside within the information domain of a problem.

The first operational analysis principle requires an examination of the information domain and the creation of a data model.

The information domain contains three different views of the data and control as each is processed by a computer program:

- a. Information content and relationships (the data model),
- b. Information flow, and
- c. Information structure.

To fully understand the information domain, each of these views should be considered. Information content represents the individual data and control objects that constitute some larger collection of information transformed by the software. For example, the data object, pay check, is a composite of a number of important pieces of data: the payee's name, the net amount to be paid, the gross pay, deductions, and so forth. Therefore, the content of pay check is defined by the attributes that are needed to create it. Similarly, the content of a control object called system status might be defined by a string of bits. Each bit represents a separate item of information that indicates whether or not a particular device is on- or off-line.

Data and control objects can be related to other data and control objects. For example, the data object pay check has one or more relationships with the objects timecard, employee, bank, and others. During the analysis of the information domain, these relationships should be defined.

Information flow represents the manner in which data and control change as each move through a system. Referring to figure 3.2, input objects are transformed to intermediate information (data and/or control), which is further transformed to output.

Along this transformation path (or paths), additional information may be introduced from an existing data store (e.g., a disk file or memory buffer). The transformations applied to the data are functions or sub-functions that a program must perform. Data and control that move between two transformations (functions) define the interface for each function.

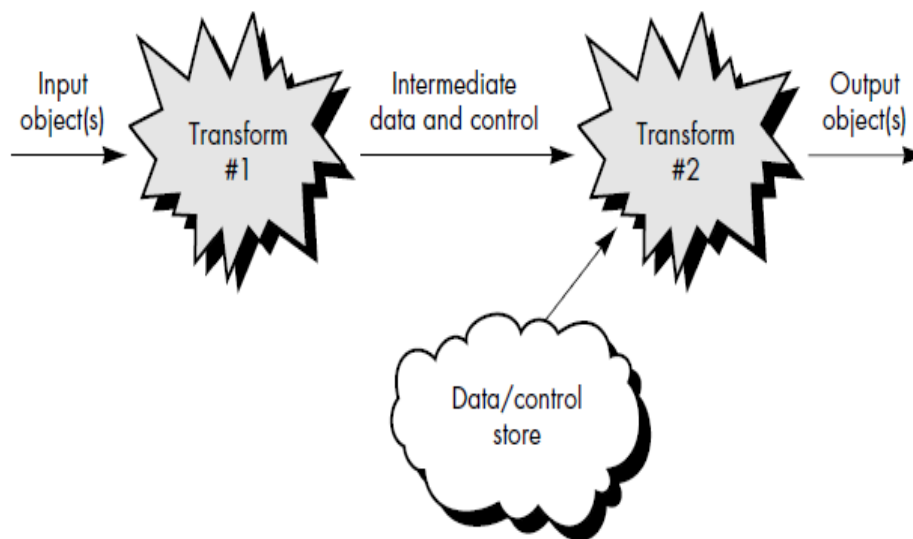


Figure 3.2: Information flow and transformation

Information structure represents the internal organization of various data and control items. Are data or control items to be organized as an n-dimensional table or as a hierarchical tree structure? Within the context of the structure, what information is related to other information? Is all information contained within a single structure or are distinct structures to be used? How does information in one information structure relate to information in another structure? These questions and others are answered by an assessment of information structure. It should be noted that data structure, a related concept discussed later in this book, refers to the design and implementation of information structure within the software.

3.5 MODELLING

We create functional models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (a building, a plane, a machine), we can build a model that

is identical in form and shape but smaller in scale. However, when the entity to be built is software, our model must take a different form. It must be capable of representing the information that software transforms, the functions (and sub-functions) that enable the transformation to occur, and the behaviour of the system as the transformation is taking place. The second and third operational analysis principles require that we build models of function and behaviour.

Functional models -Software transforms information, and in order to accomplish this, it must perform at least three generic functions: input, processing, and output. When functional models of an application are created, the software engineer focuses on problem specific functions. The functional model begins with a single context level model (i.e., the name of the software to be built). Over a series of iterations, more and more functional detail is provided, until a thorough delineation of all system functionality is represented. Behavioural models. Most software responds to events from the outside world. This stimulus/response characteristic forms the basis of the behavioural model. A computer program always exists in some state—an externally observable mode of behaviour (e.g., waiting, computing, printing, polling) that is changed only when some event occurs.

For example, software will remain in the wait state until

- i. An internal clock indicates that some time interval has passed,
- ii. An external event (e.g., a mouse movement) causes an interrupt, or
- iii. An external system signals the software to act in some manner.

A behavioural model creates a representation of the states of the software and the events that cause software to change state. Models created during requirements analysis serve a number of important roles:

- The model aids the analyst in understanding the information, function, and behaviour of a system, thereby making the requirements analysis task easier and more systematic.
- The model becomes the focal point for review and, therefore, the key to a determination of completeness, consistency, and accuracy of the specifications.
- The model becomes the foundation for design, providing the designer with an essential representation of software that can be "mapped" into an implementation context.

Although the modelling method that is used is often a matter of personal (or organizational) preference, the modelling activity is fundamental to good analysis work..

Check Your Progress 3.

What are the objectives of Analysis modelling?

3.6 PARTITIONING

Problems are often too large and complex to be understood as a whole. For this reason, we tend to partition (divide) such problems into parts that can be easily understood and establish interfaces between the parts so that overall function can be accomplished. The fourth operational analysis principle suggests that the information, functional, and behavioural domains of software can be partitioned. In essence, partitioning decomposes a problem into its constituent parts.

Conceptually, we establish a hierarchical representation of function or information and then partition the uppermost element by

- a. Exposing increasing detail by moving vertically in the hierarchy or
- b. Functionally decomposing the problem by moving horizontally in the hierarchy.

The software allocation for Safe Home (derived as a consequence of system engineering and FAST activities) can be stated in the following paragraphs:

Safe Home software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the Safe Home control panel shown in figure 3.3:

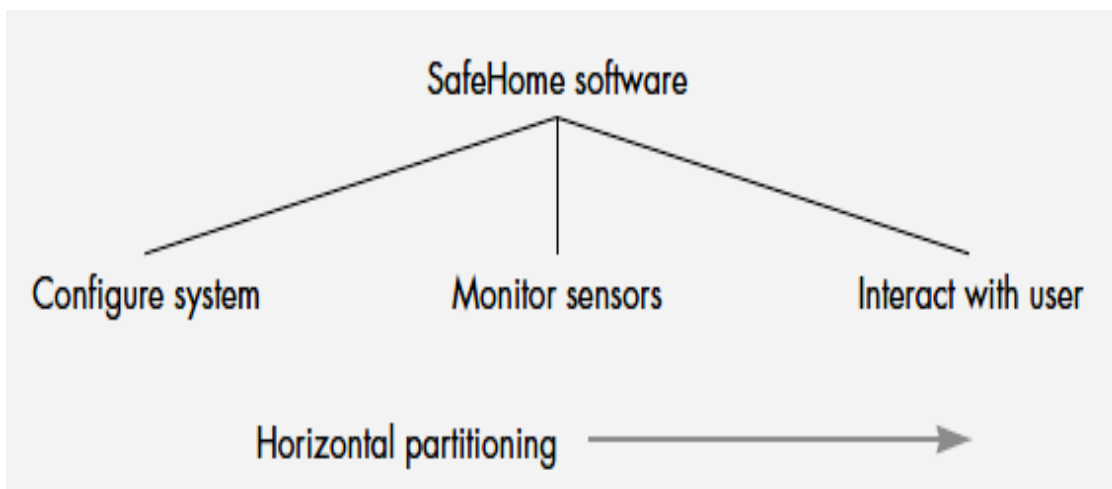


Figure 3.3: Horizontal partitioning of Safe Home function

During installation, the Safe Home control panel is used to "program" and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialling when a sensor event occurs. When a sensor event is recognized, the software invokes an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialled every 20 seconds until telephone connection is obtained.

All interaction with Safe Home is managed by a user-interaction subsystem that reads input provided through the keypad and function keys, displays prompting messages on the LCD display, display system status information on the LCD display. Keyboard interaction takes the following form.

The requirements for Safe Home software may be analysed by partitioning the information, functional, and behavioural domains of the product. To illustrate, the functional domain of the problem will be partitioned. Figure 3.3 illustrates a horizontal decomposition of Safe Home software. The problem is partitioned by representing constituent Safe Home software functions, moving horizontally in the functional hierarchy. Three major functions are noted on the first level of the hierarchy. The sub functions associated with a major Safe Home function may be examined by exposing detail vertically in the hierarchy, as illustrated in figure 3.4.

Moving downward along a single path below the function monitor sensors, partitioning occurs vertically to show increasing levels of functional detail. The partitioning approach that we have applied to Safe Home functions can also be applied to the information domain and behavioural domain as well. In fact, partitioning of information flow and system behaviour will provide additional insight into software requirements. As the problem is partitioned, interfaces between functions are derived. Data and control items that move across an interface should be restricted to inputs required to perform the stated function and outputs that are required by other functions or system elements.

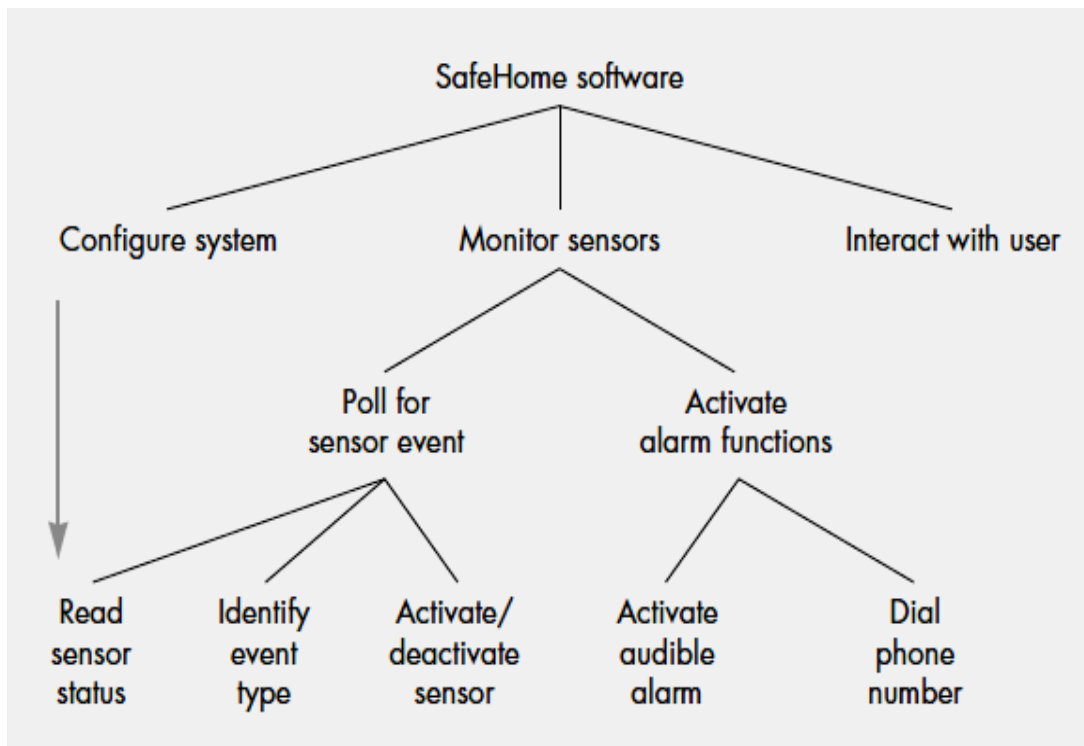


Figure 3.4: Vertical partitioning of Safe Home function

Check Your Progress 4.

What is the difference between horizontal and vertical partitioning?

3.7 ESSENTIAL AND IMPLEMENTATION VIEWS

An essential view of software requirements presents the functions to be accomplished and information to be processed without regard to implementation details. For example, the essential view of the Safe Home function read sensor status does not concern itself with the physical form of the data or the type of sensor that is used. In fact, it could be argued that read status would be a more appropriate name for this function, since it disregards details about the input mechanism altogether. Similarly, an essential data model of the data item phone number (implied by the function dial phone number) can be represented at this stage without regard to the underlying data structure (if any) used to implement the data item. By focusing attention on the essence of the problem at early stages of requirements engineering, we leave our options open to specify implementation details during later stages of requirements specification and software design.

The implementation view of software requirements presents the real world manifestation of processing functions and information structures. In some cases, a physical representation is developed as the first step in software design. However, most computer-based systems are specified in a manner that dictates accommodation of certain implementation details. A Safe Home input device is a perimeter sensor (not a watch dog, a human guard, or a booby trap). The sensor detects illegal entry by sensing a break in an electronic circuit. The general characteristics of the sensor should be noted as part of a software requirements specification. The analyst must recognize the constraints imposed by predefined system elements (the sensor) and consider the implementation view of function and information when such a view is appropriate.

We have already noted that software requirements engineering should focus on what the software is to accomplish, rather than on how processing will be implemented. However, the implementation view should not necessarily be interpreted as a representation of how. Rather, an implementation model represents the current mode of operation; that is, the existing or proposed allocation for all system elements. The essential model (of function or data) is generic in the sense that realization of function is not explicitly indicated.

Check Your Progress 5.

How SRS can prevent from risk.

3.8 SPECIFICATION

There is no doubt that the mode of specification has much to do with the quality of solution. Software engineers who have been forced to work with incomplete, inconsistent or misleading specifications have experienced the frustration and confusion that invariably results. The quality, timeliness, and completeness of the software suffer as a consequence.

Check Your Progress 6.

Who should be writing Software requirements specifications

3.9 SPECIFICATION PRINCIPLES

Specification, regardless of the mode through which we accomplish it, may be viewed as a representation process. Requirements are represented in a manner that ultimately leads to

successful software implementation. A number of specification principles, adapted from the work of Balzer and Goodman, can be proposed:

1. Separate functionality from implementation.
2. Develop a model of the desired behaviour of a system that encompasses data and the functional responses of a system to various stimuli from the environment.
3. Establish the context in which software operates by specifying the manner in which other system components interact with software.
4. Define the environment in which the system operates and indicate how “a highly intertwined collection of agents react to stimuli in the environment produced by those agents”.
5. Create a cognitive model rather than a design or implementation model. The cognitive model describes a system as perceived by its user community.
6. Recognize that “the specifications must be tolerant of incompleteness and augmentable.” A specification is always a model—an abstraction—of some real (or envisioned) situation that is normally quite complex. Hence, it will be incomplete and will exist at many levels of detail.
7. Establish the content and structure of a specification in a way that will enable it to be amenable to change.

This list of basic specification principles provides a basis for representing software requirements. However, principles must be translated into realization.

Check Your Progress 7.

Why is SRS also known as the black box specification of system ?

3.10 REPRESENTATION

We have already seen that software requirements may be specified in a variety of ways. However, if requirements are committed to paper or an electronic presentation medium (and they almost always should be!) a simple set of guidelines is well worth following:

Representation format and content should be relevant to the problem. A general outline for the contents of a Software Requirements Specification can be developed. However, the representation forms contained within the specification are likely to vary with the application area. For example, a specification for a manufacturing automation system might use different symbology, diagrams and language than the specification for a programming language compiler. Information contained within the specification should be nested. Representations

should reveal layers of information so that a reader can move to the level of detail required. Paragraph and diagram numbering schemes should indicate the level of detail that is being presented. It is sometimes worthwhile to present the same information at different levels of abstraction to aid in understanding.

Diagrams and other notational forms should be restricted in number and consistent in use. Confusing or inconsistent notation, whether graphical or symbolic, degrades understanding and fosters errors. Representations should be revisable. The content of a specification will change. Ideally, CASE tools should be available to update all representations that are affected by each change.

Investigators have conducted numerous studies on human factors associated with specification. There appears to be little doubt that symbology and arrangement affect understanding. However, software engineers appear to have individual preferences for specific symbolic and diagrammatic forms. Familiarity often lies at the root of a person's preference, but other more tangible factors such as spatial arrangement, easily recognizable patterns, and degree of formality often dictate an individual's choice.

Check Your Progress 8.

Benefits to a well-written Software Requirement Specification

3.11 THE SOFTWARE REQUIREMENT SPECIFICATION

The Software Requirements Specification is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behaviour, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.

In addition it also contains non-functional requirements. Non-functional requirements impose constraints on the design or implementation (such as performance engineering requirements, quality standards, or design constraints).

Software requirements specification establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do as well as what it is not expected to do. Software requirements specification permits a rigorous assessment of

requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules.

The software requirements specification document enlists enough and necessary requirements that are required for the project development. To derive the requirements we need to have clear and thorough understanding of the products to be developed or being developed. This is achieved and refined with detailed and continuous communications with the project team and customer till the completion of the software. The SRS may be one of a contract deliverable Data Item Descriptions or have other forms of organizationally-mandated content.

An example organization of an SRS is as follows:

- Introduction
 - Purpose
 - Definitions
 - System overview
 - References
- Overall description
 - Product perspective
 - System Interfaces
 - User Interfaces
 - Hardware interfaces
 - Software interfaces
 - Communication Interfaces
 - Memory Constraints
 - Operations
 - Site Adaptation Requirements
 - Product functions
 - User characteristics
 - Constraints, assumptions and dependencies
- Specific requirements
 - External interface requirements
 - Functional requirements
 - Performance requirements
 - Design constraints
 - Standards Compliance

- Logical database requirement
- Software System attributes
 - Reliability
 - Availability
 - Security
 - Maintainability
 - Portability
- Other requirements

Characteristics of SRS:

An SRS should be:

- a. **Correct:** An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet. Traceability makes this procedure easier and less prone to error.
- b. **Unambiguous:** An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term.
- c. **Complete:** An SRS is complete if, and only if, it includes the following elements:
 - i) All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.
 - ii) Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.
 - iii) Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.
- d. **Consistent:** Consistency refers to internal consistency. If an SRS does not agree with some higher-level document, such as a system requirements specification, then it is not correct. An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict.

Ranked for importance or stability

An SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement. Typically, all of the

requirements that relate to a software product are not equally important. Some requirements may be essential, especially for life-critical applications, while others may be desirable. Each requirement in the SRS should be identified to make these differences clear and explicit-

- a. **Verifiable:** An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. Non verifiable requirements include statements such as "works well", "good human interface", and "shall usually happen". These requirements cannot be verified because it is impossible to define the terms "good", "well", or "usually".
- b. **Modifiable:** An SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. Modifiability generally requires an SRS to
 - i. Have a coherent and easy-to-use organization with a table of contents, an index, and explicit cross-referencing;
 - ii. Not be redundant (i.e., the same requirement should not appear in more than one place in the SRS);
 - iii. Express each requirement separately, rather than intermixed with other requirements.
- c. **Traceable:** An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. The following two types of traceability are recommended:
 - i. *Backward traceability (i.e., to previous stages of development).* This depends upon each requirement explicitly referencing its source in earlier documents.
 - ii. *Forward traceability (i.e., to all documents spawned by the SRS).* This depends upon each requirement in the SRS having a unique name or reference number.

Check Your Progress 9.

What are the characteristics of SRS?

3.12 SUMMARY

In this section we cover the topics software requirement analysis and specification, their principle, objectives, goal, and characteristics, Modelling, Partitioning –vertical and horizontal both, their benefits and about information domain. A software engineer who takes these principles to heart is more likely to develop a software specification that will provide an excellent foundation for design. A Software requirements specification (SRS), a requirements specification for a software system, is a description of the behaviour of a system to be developed and may include a set of use cases that describe interactions the users will have with the software. We also cover the topics like objectives, goal, and characteristics of Software requirement analysis and specification, Modelling, Partitioning –vertical and horizontal both, their benefits and about information domain.

3.13 EXERCISE

- 1) What is requirement analysis? What is the role of requirement analysis?
- 2) Explain Analysis Modelling Approaches with example.
- 3) What is horizontal partitioning? What are the benefits of horizontal partitioning?
- 4) What is vertical partitioning? What are the advantages of vertical partitioning?
- 5) Explain software Requirement Specification. What are the Objectives of Requirement Analysis ?
- 6) What are the characteristics of SRS?
- 7) What are the Difficulties in Elicitations?
- 8) What are the difference between requirements definition and requirement specification .

Software Engineering

FORTH - BLOCK

BLOCK

4

UNIT 1 DESIGN CONCEPT AND PRINCIPLE

UNIT 2 SOFTWARE TESTING

UNIT 3 TYPES OF SOFTWARE TESTING

UNIT 4 RE ENGINEERING

UNIT 5 CASE

Overview

In this section we discuss the overview of this block's content. This block consists of the following units:

Unit 1 Design Concept and Principle

Software design is the process of implementing software solution. One of the main input of software design is the software requirements analysis (SRA). The design concepts provide the software designer with a foundation from which methods can be applied. Furthermore, a software design may be platform-independent or platform-specific, depending upon the availability of the technology used for the design. If design is proper according to the need of customer then the chances of error is reduce.

Unit 2 Software Testing

In software development, software testing play a vital role. After coding and before delivered to the customer, software is tested according to the need or requirement of customer

In this unit several testing are defined which are used to test the software. And once software is error free and maintains all the requirements of customer then delivered to the customer.

Unit 3 Types of Software Testing

Testing is necessary for successful execution of software before delivered to the customer. In this section, we describe different types of software testing. These software testing are applied to achieve different objectives when testing a software application.

Unit 4 Re Engineering

This section covers how old/existing software can be improved and performed effectively. The objective of re-engineering is to improve the system structure to make it easier to understand and maintain .Which type of activities involved in the software re-engineering process and to explain the problems of re-engineering. Also describes reverse and forward engineering and their need as well as benefits.

Unit 5 CASE

A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form of automated support for software engineering. In a more restrictive sense, a CASE tool means any tool used to automate some activity associated with software development. Many CASE tools are available. Some of these CASE tools assist in phase related tasks such as specification, structured analysis, design, coding, testing, etc. CASE plays an interesting role in the software development life cycle.

UNIT-1 DESIGN CONCEPT AND PRINCIPLE

- 1.0** Introduction
- 1.1** Objectives
- 1.2** Design Principle
- 1.3** Abstraction
- 1.4** Refinement
- 1.5** Modularity
- 1.6** Software Architecture
- 1.7** Control Hierarchy
- 1.8** Structural Partitioning
- 1.9** Data Structure
- 1.10** Software Procedure
- 1.11** Information Hiding
- 1.12** Effective Modular design
- 1.13** Cohesion
- 1.14** Coupling
- 1.15** Summary
- 1.16** Exercise

1.0 INTRODUCTION

A software design creates meaningful engineering representation (or model) of some software product that is to be built. Designers must strive to acquire a repertoire of alternative design information and learn to choose the elements that best match the analysis model. A design model can be traced to the customer's requirements and can be assessed for quality against predefined criteria. During the design process the software requirements model (data, function, and behaviour) is transformed into design models that describe the details of the data structures, system architecture, interfaces, and components necessary to implement the system. Each design product is reviewed for quality before moving to the next phase of software development.

Software Design

- Encompasses the set of principles, concepts, and practices that lead to the development of a high quality system or product
- Design principles establish an overriding philosophy that guides the designer as the work is performed
- Design concepts must be understood before the mechanics of design practice are applied
- Goal of design engineering is to produce a model or representation that is bug free (firmness), suitable for its intended uses (commodity), and pleasurable to use (delight)
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve.

1.1 OBJECTIVE

The objectives of this unit are:

- a) to introduce the process of software design
- b) to describe the different stages in this design process
- c) to show how object-oriented and functional design strategies are complementary
- d) to discuss some design quality attributes

1.2 DESIGN PRINCIPLE

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

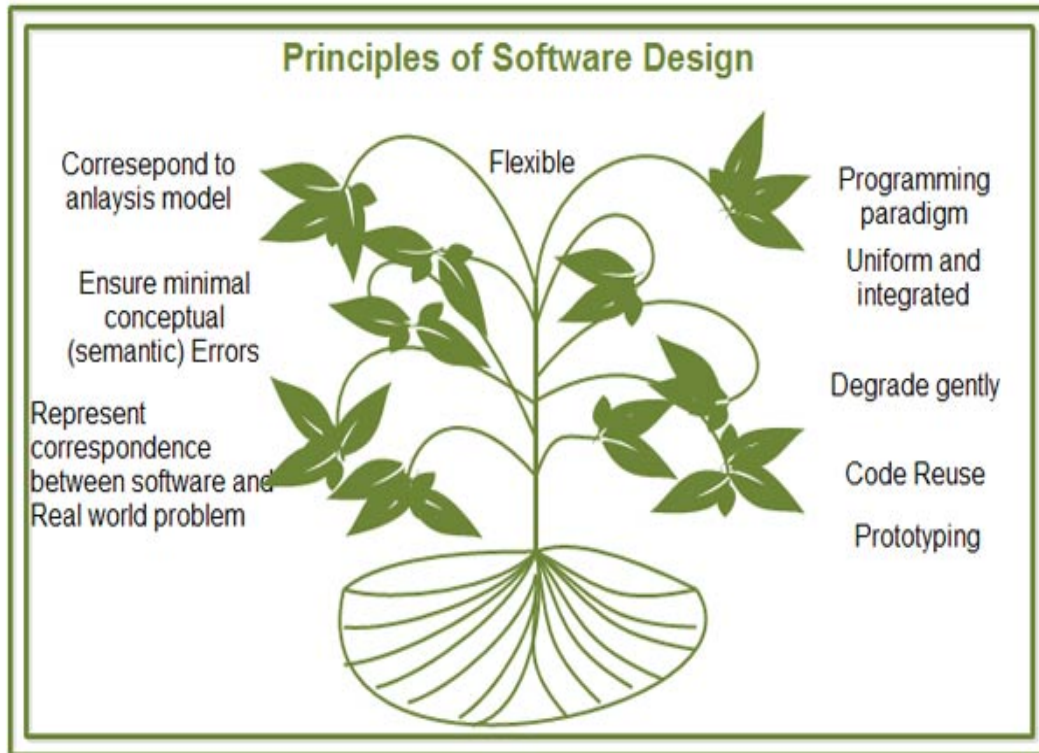


Figure 1.1: Principles of Design

Some of the commonly used design principles as mentioned in figure 1.1 are as following.

- i. ***Software design should correspond to the analysis model:*** Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.
- ii. ***Choose the right programming paradigm:*** A programming paradigm describes the structure of the software system. Depending on the nature and type of application, different programming paradigms such as procedure oriented, object-oriented, and prototyping paradigms can be used. The paradigm should be chosen keeping constraints in mind such as time, availability of resources and nature of user's requirements.
- iii. ***Software design should be uniform and integrated:*** Software design is considered uniform and integrated, if the interfaces are properly defined among the design components. For this, rules, format, and styles are established before the design team starts designing the software.

- iv. ***Software design should be flexible:*** Software design should be flexible enough to adapt changes easily. To achieve the flexibility, the basic design concepts such as abstraction, refinement, and modularity should be applied effectively.
- v. ***Software design should ensure minimal conceptual (semantic) errors:*** The design team must ensure that major conceptual errors of design such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.
- vi. ***Software design should be structured to degrade gently:*** Software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.
- vii. ***Software design should represent correspondence between the software and real-world problem:*** The software design should be structured in such a way that it always relates with the real-world problem.
- viii. ***Software reuse:*** Software engineers believe on the phrase: 'do not reinvent the wheel'. Therefore, software components should be designed in such a way that they can be effectively reused to increase the productivity.
- ix. ***Designing for testability:*** A common practice that has been followed is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed (designed and implemented) and then handed over to the testers who subsequently determine whether the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as if any type of design or implementation errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should be involved with analysts to prepare tests for determining whether the user requirements are being met.
- x. ***Prototyping:*** Prototyping should be used when the requirements are not completely defined in the beginning. The user interacts with the developer to expand and refine the requirements as the development proceeds. Using prototyping, a quick 'mock-up' of the system can be developed. This mock-up can be used as effective means to give the users a feel of what the system will look like and demonstrate functions that will be included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer's requirements.

Design Concepts

- **Abstraction** – allows designers to focus on solving a problem without being concerned about irrelevant lower level details (procedural abstraction - named sequence of events and data abstraction – named collection of data objects)
- **Software Architecture** – overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
 - Structural models – architecture as organized collection of components
 - Framework models – attempt to identify repeatable architectural patterns
 - Dynamic models – indicate how program structure changes as a function of external events
 - Process models – focus on the design of the business or technical process that system must accommodate
 - Functional models – used to represent system functional hierarchy
- **Design Patterns** – description of a design structure that solves a particular design problem within a specific context and its impact when applied
- **Separation of concerns** – any complex problem is solvable by subdividing it into pieces that can be solved independently
- **Modularity** - the degree to which software can be understood by examining its components independently of one another
- **Information Hiding** – information (data and procedure) contained within a module is inaccessible to modules that have no need for such information
- **Functional Independence** – achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models
 - Cohesion - qualitative indication of the degree to which a module focuses on just one thing
 - Coupling - qualitative indication of the degree to which a module is connected to other modules and to the outside world
- **Refinement** – process of elaboration where the designer provides successively more detail for each design component
- **Aspects** – a representation of a cross-cutting concern that must be accommodated as refinement and modularization occur
- **Refactoring** – process of changing a software system in such a way internal structure is improved without altering the external behaviour or code design.

Design considerations

There are many aspects to consider in the design of a piece of software. The importance of each consideration should reflect the goals and expectations that the software is being created to meet. Some of these aspects are:

- *Compatibility* - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- *Extensibility* - New capabilities can be added to the software without major changes to the underlying architecture.
- *Modularity* - the resulting software involves well defined, independent components which indicate to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
- *Fault-tolerance* - The software is resistant to and able to recover from component failure.
- *Maintainability* - A measure of how easily bug fixes or functional modifications can be accomplished. High maintainability can be the product of modularity and extensibility.
- *Reliability* (Software durability) - The software is able to perform a required function under stated conditions for a specified period of time.
- *Reusability* - The ability to use some or all the aspects of the pre-existing software in other projects with little to no modification.
- *Robustness* - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with resilience to low memory conditions.
- *Security* - The software is able to withstand and resist hostile acts and influences.
- *Usability* - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.
- *Performance* - The software performs its tasks within a time-frame that is acceptable for the user, and does not require too much memory.
- *Portability* - The software should be usable across a number of different conditions and environments.

- *Scalability* - The software adapts well to increasing data or number of users.

Top-down and bottom-up approaches of designing:

Top-down and bottom-up are both strategies of information processing and knowledge ordering, used in a variety of fields including software, humanistic and scientific theories and management and organization. In practice, they can be seen as a style of thinking and teaching.

A ***top-down approach*** (also known as stepwise design and in some cases used as a synonym of decomposition) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model. Top down approach starts with the big picture. It breaks down from there into smaller segments.

A ***bottom-up approach*** is the piecing together of systems to give rise to more complex systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of information processing based on incoming data from the environment to form a perception. Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output). In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose. In the software development process, the top-down and bottom-up approaches play a key role.

Check Your Progress 1.

List out the elements of design model.

1.3 ABSTRACTION

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

Each step in the software process is accomplished through various levels of abstraction. At the highest level, an outline of the solution to the problem is presented whereas at the lower levels, the solution to the problem is presented in detail. For example, in the requirements analysis phase, a solution to the problem is presented using the language of problem environment and as we proceed through the software process, the abstraction level reduces and at the lowest level, source code of the software is produced.

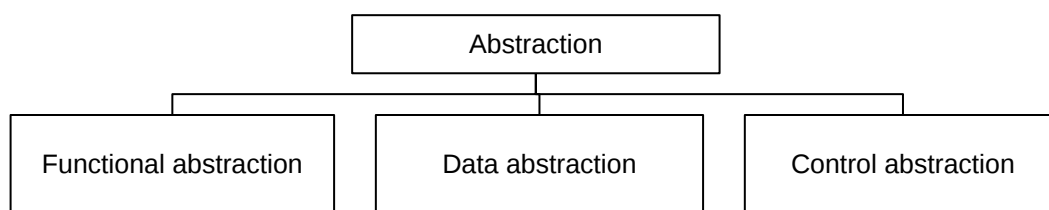


Figure 1.2: Types of abstraction

There are three commonly used abstraction mechanisms in software design, namely, functional abstraction, data abstraction and control abstraction as shown in figure 1.2. 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.

- a. **Functional abstraction:** This involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as 'groups'. Within these groups there exist routines which may be visible or hidden. Visible routines can be used within the containing groups as well as within other groups, whereas hidden routines are hidden from other groups and can be used within the containing group only.
- b. **Data abstraction:** This involves specifying data that describes a data object. For example, the data object window encompasses a set of attributes (window type,

window dimension) that describe the window object clearly. In this abstraction mechanism, representation and manipulation details are ignored.

- c. **Control abstraction:** This 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 the architectural design level, this abstraction mechanism permits specifications of sequential subprogram and exception handlers without the concern for exact details of implementation.

Check Your Progress 2.

What are different levels of abstraction?

1.4 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 input, process, and output.

1. INPUT

Get user's name (string) through a prompt.

Get user's grade (integer from 0 to 10) through a prompt and validate.

2. PROCESS

3. OUTPUT

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

1. INPUT Get user's name through a prompt.

Get user's grade through a prompt.

While (invalid grade)

Ask again:

2. PROCESS

3. OUTPUT

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

Check Your Progress 3.

What is stepwise refinement?

1.5 MODULARITY

The real power of partitioning comes if a system is partitioned into modules so that the modules are solvable and modifiable separately. It will be even better if the modules are also separately compilable (then, changes in a module will not require recompilation of the whole system). A system is considered modular if it consists of discrete components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

Modularity is a clearly a desirable property in a system. Modularity helps in system debugging. Isolating the system problem to a component is easier if the system is modular. In system repair, hanging a part of the system is easy as it affects few other parts and in system building, a modular system can be easily built by “putting its modules together.”

A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well-defined abstraction and have a clear interface through which it can interact with other modules. Modularity is where abstraction and partitioning come together. For easily understandable and maintainable systems, modularity is clearly the basic objective; partitioning and abstraction can be viewed as concepts that help achieve modularity.

As figure 1.3 represents, Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as modules. A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing the modules, they are integrated together to meet the software requirements.

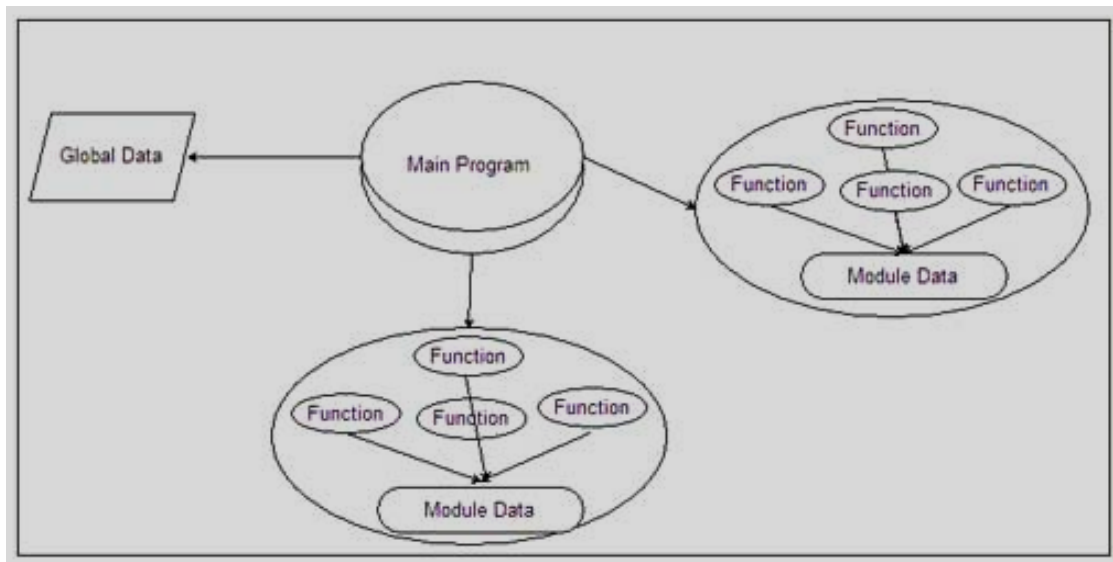


Figure 1.3: Modules in Software Programs

Larger the number of modules a system is divided into, greater will be the effort required to integrate the modules. Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conducts maintenance work without adversely affecting the functioning of the software.

Check Your Progress 4.

How can we evaluate a design method to determine if it will lead to efficient modularity?

1.6 SOFTWARE ARCHITECTURE

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

- Provides an insight to all the interested stakeholders that enable them to communicate with each other
- Highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase
- Creates intellectual models of how the system is organized into components and how these components interact with each other.

Currently, software architecture is represented in an informal and unplanned manner. Though the architectural concepts are often represented in the infrastructure (for supporting particular architectural styles) and the initial stages of a system configuration, the lack of an explicit independent characterization of architecture restricts the advantages of this design concept in the present scenario.

Software architecture comprises two elements of design model, namely, data design and architectural design.

Check Your Progress 5.

How are the architectural designs analysed? Explain.

1.7 CONTROL HIERARCHY

Control structure is a program structure that represents the organization of a program component and implies a hierarchy of control. Hierarchy of modules represents the control relationships. A super-ordinate module controls another module. A subordinate module is controlled by another module.

Measures relevant to control hierarchy: depth, width, fan-in, fan-out as shown in figure 1.4.

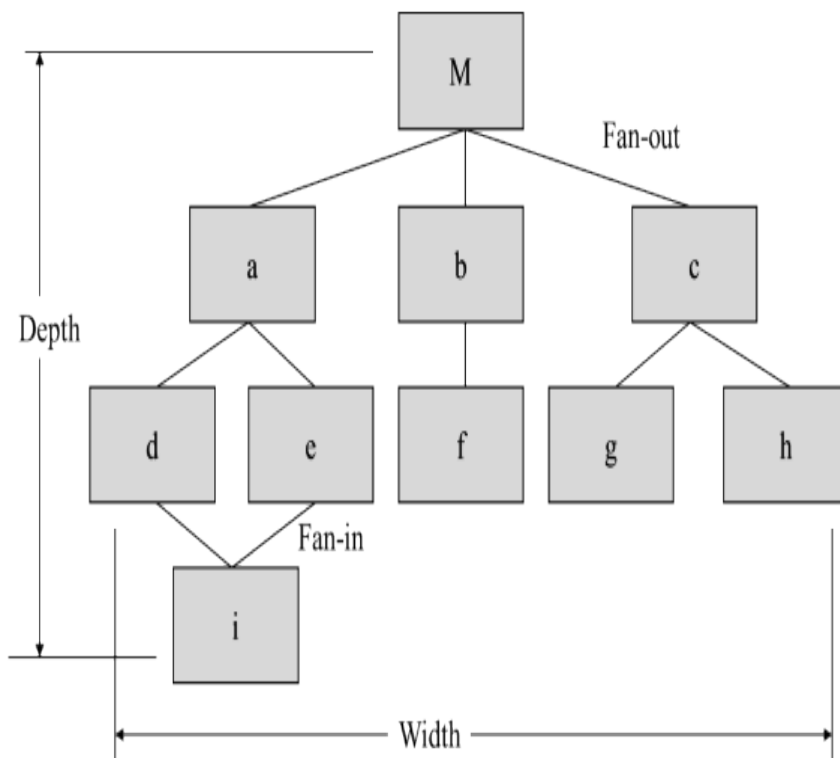


Figure1.4: Structure of Hierarchy

Check Your Progress 6.

Explain width, depth, fan-in, fan-out in control hierarchy.

1.8 STRUCTURAL PARTITIONING

Program structure is partitioned horizontally and vertically as figure 1.5 shown. Horizontal partitioning defines separate branches for each major program function - input, process, and output. Vertical partitioning (aka factoring) defines control (decision-making) at the top and work at the bottom.

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 are used to communicate between functions and execute the functions. Structural partitioning provides the following benefits.

- The testing and maintenance of software becomes easier.
- The negative impacts spread slowly.
- The software can be extended easily.

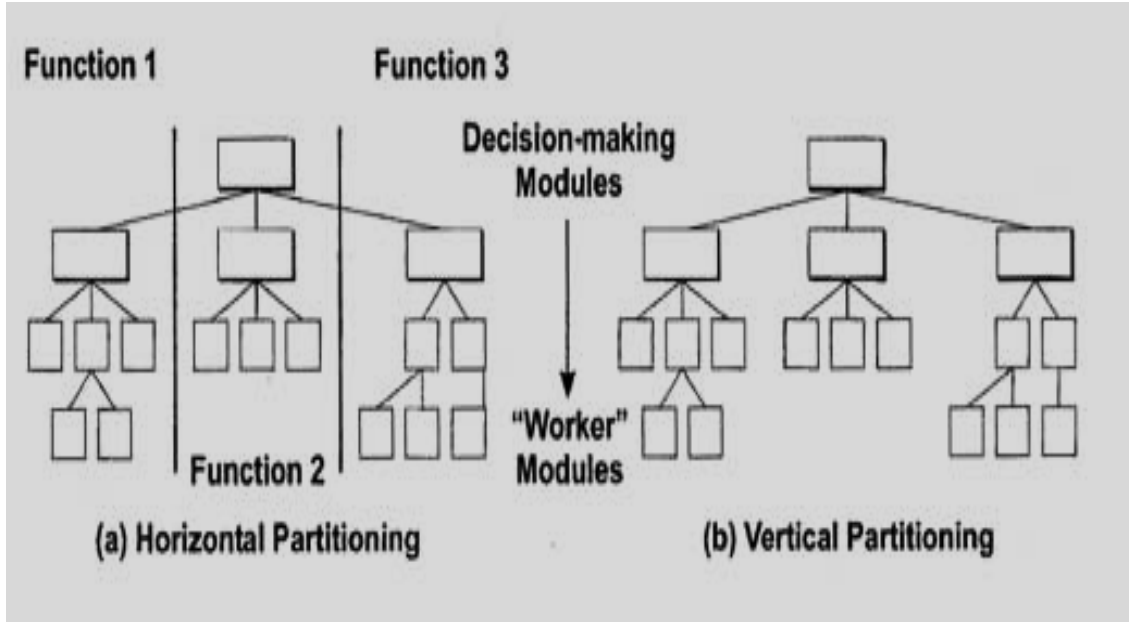


Figure1.5: Horizontal and Vertical Partitioning

Besides these advantages, horizontal partitioning has some disadvantage also. It requires more data to permit across the module interface, which makes the control flow of the problem

more complex. This usually happens in cases where data moves rapidly from one function to another.

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

Check Your Progress 7.

What are the benefits of horizontal partitioning?

1.9 DATA STRUCTURE

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture.

Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. Entire texts have been dedicated to these topics, and a complete discussion is beyond the scope of this book. However, it is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies.

The organization and complexity of a data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

A scalar item is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer or floating point number that is 8 to 64 bits long, or a character string that; is hundreds or thousands of bytes long.

When scalar items are organized as a list or contiguous group, a sequential vector is formed. Vectors are the most common of all data structures and open the door to variable indexing of information. When the sequential vector is extended to two, three, and ultimately, an arbitrary

number of dimensions, an n-dimensional space is created. The most common n-dimensional space is the two-dimensional matrix. In many programming languages, an n-dimensional space is called an array.

Items, vectors, and spaces may be organized in a variety of formats. A linked list is a data structure that organizes non-contiguous scalar items...vectors, or spaces in a manner (called nodes) that enables them to be processed as a list. Each node contains the appropriate data organization (e.g., a vector) and one or more pointers that indicate the address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.

Other data structures incorporate or are constructed using the fundamental data structures just described. For example, a hierarchical data structure is implemented using multilinked lists that contain scalar items, vectors, and possibly, n-dimensional spaces. A hierarchical structure is commonly encountered in applications that require information categorization and associativity.

It is important to note that data structures, like program structure, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structure that can be implemented as a vector or a linked list. Depending on the level of design detail, the internal workings of a stack may or may not be specified.

Check Your Progress 8.

Application area of hierarchical data structure.

1.10 SOFTWARE PROCEDURE

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

There is, of course, a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described.

1.11 INFORMATION HIDING

Modules should be specified and designed in such a way that the data structures and processing details of one module are not accessible to other modules. They pass only that much information to each other, which is required to accomplish the software functions. The way of hiding unnecessary details is referred to as information hiding as shown in figure 1.6. IEEE defines information hiding as 'the technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus each module is a 'black box' to the other modules in the system.

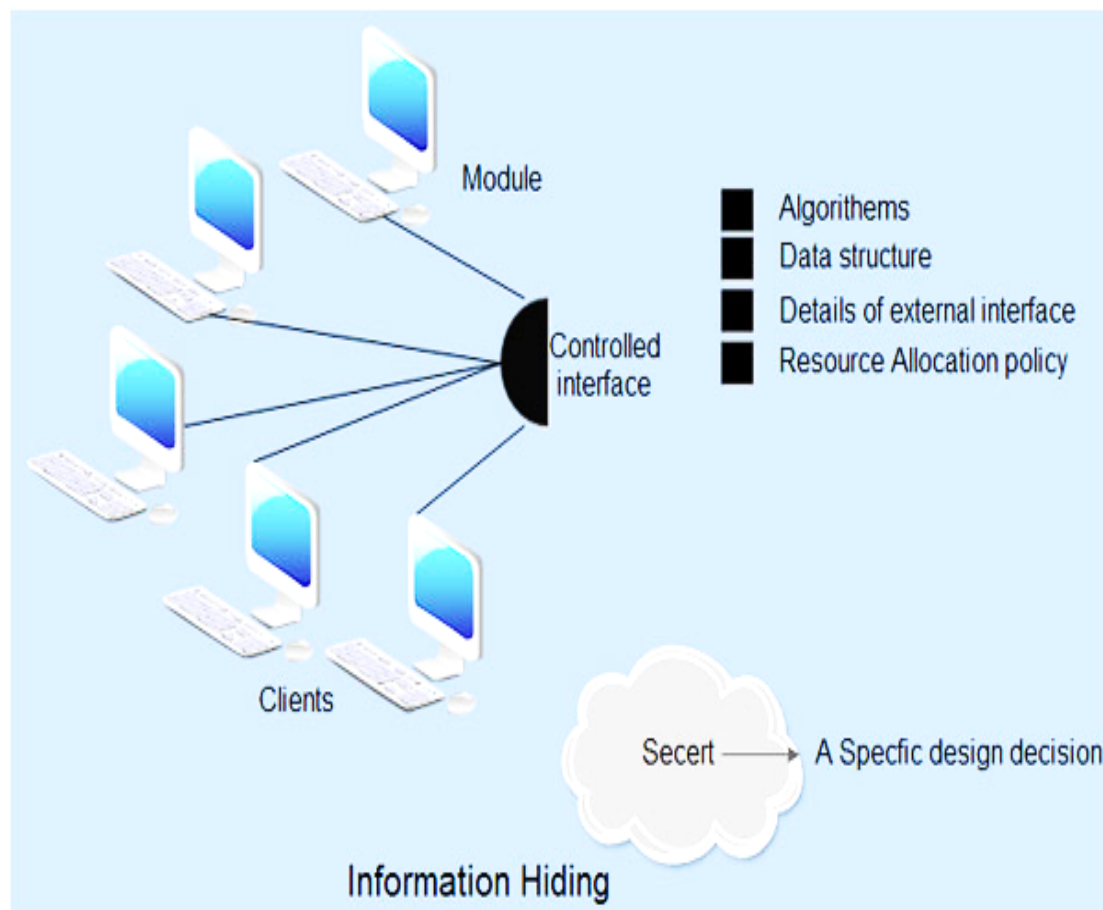


Figure1.6: Information Hiding

Information hiding is of immense use when modifications are required during the testing and maintenance phase. Some of the advantages associated with information hiding are listed below:

- i.** Leads to low coupling
- ii.** Emphasizes communication through controlled interfaces
- iii.** Decreases the probability of adverse effects
- iv.** Restricts the effects of changes in one component on others

- v. Results in higher quality software.

Check Your Progress 9.

Why information hiding is important?

1.12 EFFECTIVE MODULAR DESIGN

Effective modular design is the *direct outgrowth of*:

- Modularity
- Information hiding

Effective modular design is *measured by*:

- Cohesion
- Coupling

Criteria to evaluate Efficiency:

There are five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

- i. ***Modular decomposability***- If a design method provides a systematic mechanism for decomposing the problem into sub-problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.
- ii. ***Modular composability***- If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.
- iii. ***Modular understand ability***- If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.
- iv. ***Modular continuity***- If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.
- v. ***Modular protection***- If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

Design heuristics for effective modularity

- i. Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.

- ii. Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.
- iii. Keep the scope of effect of a module within the scope of control of that module.
- iv. Evaluate module interfaces to reduce complexity and redundancy and improve consistency.
- v. Define modules whose function is predictable, but avoid modules that are overly restrictive.
- vi. Strive for “controlled entry” modules by avoiding "pathological connections."

Check Your Progress 10.

What is the benefit of modular design?

1.13 COHESION

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design. Measure of how well module fits together. A component should implement a single logical function or single logical entity. All the parts should contribute to the implementation.

There are many levels of cohesion as described below:

- i. ***Coincidental cohesion:*** the parts of a component are not related but simply bundled into a single component. Harder to understand and not reusable.
- ii. ***Logical association:*** similar functions such as input, error handling, etc. put together. Functions fall in same logical class. May pass a flag to determine which ones executed. Interface difficult to understand. Code for more than one function may be intertwined, leading to severe maintenance problems. Difficult to reuse.
- iii. ***Temporal cohesion:*** all of statements activated at a single time, such as start up or shut down, are brought together. Initialization, clean up. Functions weakly related to one another, but more strongly related to functions in other modules so may need to change lots of modules when do maintenance.
- iv. ***Procedural cohesion:*** a single control sequence, e.g., a loop or sequence of decision statements. Often cuts across functional lines. May contain only part of a complete function or parts of several functions. Functions still weakly connected, and again unlikely to be reusable in another product.
- v. ***Communicational cohesion:*** operate on same input data or produce same output data. May be performing more than one function. Generally acceptable if alternate

structures with higher cohesion cannot be easily identified. Still problems with reusability.

- vi. **Sequential cohesion:** output from one part serves as input for another part. May contain several functions or parts of different functions.
- vii. **Informational cohesion:** performs a number of functions, each with its own entry point, with independent code for each function, all performed on same data structure. Different than logical cohesion because functions not intertwined.
- viii. **Functional cohesion:** each part necessary for execution of a single function. e.g., compute square root or sort the array. Usually reusable in other contexts. Maintenance easier.
- ix. **Type cohesion:** modules that support a data abstraction. Not strictly a linear scale. Functional much stronger than rest while first two much weaker than others. Often many levels may be applicable when considering two elements of a module. Cohesion of module considered as highest level of cohesion that is applicable to all elements in the module.

Check Your Progress 11.

How should software be designed considering cohesion?

1.14 COUPLING

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program. Coupling is an indication of the strength of interconnections between program units. Highly coupled have program units dependent on each other. Loosely coupled are made up of units that are independent or almost independent. Modules are independent if they can function completely without the presence of the other. Obviously, can't have modules completely independent of each other. Must interact so that can produce desired outputs. The more connections between modules, the more dependent they are in the sense that more info about one module is required to understand the other module.

Three factors:

- Number of interfaces
- Complexity of interfaces
- Type of info flow along interfaces

Want to minimize number of interfaces between modules, minimize the complexity of each interface, and control the type of info flow. An interface of a module is used to pass information to and from other modules.

In general, modules tightly coupled if they use shared variables or if they exchange control info. Loose coupling if info held within a unit and interface with other units via parameter lists. Tight coupling if shared global data. If need only one field of a record, don't pass entire record. Keep interface as simple and small as possible.

Two types of information flow: data or control.

- Passing or receiving back control info means that the action of the module will depend on this control info, which makes it difficult to understand the module.
- Interfaces with only data communication result in lowest degree of coupling, followed by interfaces that only transfer control data. Highest if data is hybrid.

Types of Coupling, ranked highest to lowest:

- i. **Content coupling:** if one directly references the contents of the other. When one module modifies local data values or instructions in another module. (Can happen in assembly language) if one refers to local data in another module. If one branches into a local label of another.
- ii. **Common coupling:** access to global data. Modules bound together by global data structures.
- iii. **Control coupling:** passing control flags (as parameters or global) so that one module controls the sequence of processing steps in another module.
- iv. **Stamp coupling:** similar to common coupling except that global variables are shared selectively among routines that require the data. E.g., packages in Ada. More desirable than common coupling because fewer modules will have to be modified if a shared data structure is modified. Pass entire data structure but need only parts of it.
- v. **Data coupling:** use of parameter lists to pass data items between routines.

How does one determine the cohesion level of a module? There is no mathematical formula that can be used. We have to use our judgment for this. A useful technique for determining if a module has functional cohesion is to write a sentence that describes, fully and accurately, the function or purpose of the module. The following tests can, then, be made:

- If the sentence must be a compound sentence, if it contains a comma, or it has more than one verb, the module is probably performing more than one function, and it probably has sequential or communicational cohesion.
- If the sentence contains words relating to time, like “first,” “next,” “when,” and “after”, the module, probably, has sequential or temporal cohesion.
- If the predicate of the sentence does not contain a single specific object following the verb (such as “edit all data”), the module probably has logical cohesion.
- Words like “initialize,” and “clean up” imply temporal cohesion.

Modules with functional cohesion can always be described by a simple sentence. However, if a description is a compound sentence, it does not mean that the module does not have functional cohesion. Functionally cohesive modules can also be described by compound sentences. If we cannot describe it using a simple sentence, the module is not likely to have functional cohesion.

Check Your Progress 12.

List the coupling factors.

1.15 SUMMARY

In This section we discuss about the important role of designing in SDLC. and different type of characteristics , concepts, aspects, approaches of designing may be used for better programing . Which type of data structure, architecture is used in different stage. How module can be correlate and tie with each other and how data can be flow among the entire module. And for all purpose how can we improve the basic of designing of software.

1.16 EXERCISE

- 1) Define design process. List the principles of a software design.
- 2) What are the benefits of modular design?
- 3) What is a cohesive module? What are the different types of Cohesion?
- 4) What is coupling? What are the various types of coupling?
- 5) What are the common activities in design process?
- 6) What is horizontal partitioning and define the benefits of horizontal partitioning?
- 7) What is vertical partitioning? What are the advantages of vertical partitioning?
- 8) What is the difference between top-down design and bottom-up design?

UNIT-2 TESTING

2.0 Introduction

2.1 Objective

2.2 Role of Testing

2.3 Principles

2.4 Unit testing

2.5 Integration testing

2.6.1 Top down Integration

2.6.2 Bottom Up Integration

2.6 System testing

2.7 Summary

2.8 Exercise

2.0 INTRODUCTION

Testing begins at the component level and works outward toward the integration of the entire computer-based system. Different testing techniques are appropriate at different points in time. The developer of the software conducts testing and may be assisted by independent test groups for large projects. Testing and debugging are different activities. Debugging must be accommodated in any testing strategy.

2.1 OBJECTIVE

Software Testing has different goals and objectives. The major objectives of Software testing are as follows:

- Finding defects which may get created by the programmer while developing the software.
- Gaining confidence in and providing information about the level of quality.
- To prevent defects.
- To make sure that the end result meets the business and user requirements.
- To ensure that it satisfies the BRS that is Business Requirement Specification and SRS that is System Requirement Specifications.
- To gain the confidence of the customers by providing them a quality product.

2.2 ROLE OF TESTING

Organizing for Software Testing

- The role of the Independent Test Group (ITG) is to remove the conflict of interest inherent when the builder is testing his or her own product.
- The developer should do no testing at all.
- Software is tossed "over the wall" to people to test it mercilessly.
- Testers are not involved with the project until it is time for it to be tested.
- The developer and ITGC must work together throughout the software project to ensure that thorough tests will be conducted.

Software Testing Strategy for Traditional Software Architectures

- *Unit Testing* - makes heavy use of testing techniques that exercise specific control paths to detect errors in each software component individually.

- *Integration Testing* - focuses on issues associated with verification and program construction as components begin interacting with one another.
- *Validation Testing* - provides assurance that the software validation criteria (established during requirements analysis) meets all functional, behavioural, and performance requirements.
- *System Testing* - verifies that all system elements mesh properly and that overall system function and performance has been achieved.

Software Testing Strategy for Object-Oriented Architectures

- *Unit Testing* - components being tested are classes not modules
- *Integration Testing* - as classes are integrated into the architecture regression tests are run to uncover communication and collaboration errors between objects .
- *Systems Testing* - the system as a whole is tested to uncover requirement errors.

Strategic Testing Issues

- Specify product requirements in a quantifiable manner before testing starts.
- Specify testing objectives explicitly.
- Identify categories of users for the software and develop a profile for each.
- Develop a test plan that emphasizes rapid cycle testing.
- Build robust software that is designed to test itself.
- Use effective formal reviews as a filter prior to testing.
- Conduct formal technical reviews to assess the test strategy and test cases.
- Develop a continuous improvement approach for the testing process.

TEST PLAN: A test plan is a document detailing a systematic approach to testing a system such as a machine or software. The plan typically contains a detailed understanding of the eventual work flow. A test plan documents the strategy that will be used to verify and ensure that a product or system meets its design specifications and other requirements. A test plan is usually prepared by or with significant input from test engineers.

Elements of Test Plan: There are three major elements that should be described in the test plan:

- i. **Test coverage:** Test coverage in the test plan states what requirements will be verified during what stages of the product life. Test Coverage is derived from design

specifications and other requirements, such as safety standards or regulatory codes, where each requirement or specification of the design ideally will have one or more corresponding means of verification. Test coverage for different product life stages may overlap, but will not necessarily be exactly the same for all stages. For example, some requirements may be verified during Design Verification test, but not repeated during Acceptance test. Test coverage also feeds back into the design process, since the product may have to be designed to allow test access.

- ii. **Test methods:** Test methods in the test plan state how test coverage will be implemented. Test methods may be determined by standards, regulatory agencies, or contractual agreement, or may have to be created new. Test methods also specify test equipment to be used in the performance of the tests and establish pass/fail criteria. Test methods used to verify hardware design requirements can range from very simple steps, such as visual inspection, to elaborate test procedures that are documented separately.
- iii. **Test responsibilities:** Test responsibilities include what organizations will perform the test methods and at each stage of the product life. This allows test organizations to plan, acquire or develop test equipment and other resources necessary to implement the test methods for which they are responsible. Test responsibilities also includes, what data will be collected, and how that data will be stored and reported (often referred to as "deliverables"). One outcome of a successful test plan should be a record or report of the verification of all design specifications and requirements as agreed upon by all parties.

Testability

Software testability defines how easily a computer program can be tested. There are some metric to measure the testability. The checklist that follows provides a set of characteristics that lead to testable software.

- Operability
- Observability
- Controllability
- Decomposability
- Simplicity
- Stability
- Understandability

Attributes of a Good Test:

- High probability of finding an error
- Not redundant
- Should be best of breed
- Neither too simple nor too complex

Method of testing

Software testing methods are traditionally divided into white- and black-box testing.

In White-box testing or **glass box testing** tests internal structures or workings of a program, as opposed to the functionality exposed to the end-user. In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. While white-box testing can be applied at the unit, integration and system levels of the software testing process, it is usually done at the unit level.

Black-box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation, without seeing the source code. The testers are only aware of what the software is supposed to do, not how it does it.

One advantage of the black box technique is that no programming knowledge is required. Because they do not examine the source code, there are situations when a tester writes many test cases to check something that could have been tested by only one test case, or leaves some parts of the program untested. This method of test can be applied to all levels of software testing: unit, integration, system and acceptance.

There is one more box testing that is grey box testing. **Grey-box testing** involves having knowledge of internal data structures and algorithms for purposes of designing tests, while executing those tests at the user, or black-box level. The tester is not required to have full access to the software's source code.

Levels of Testing

Different levels of testing are used in the testing process; each level of testing aims to test different aspects of the system.

There are generally three recognized levels of testing:

- unit testing
- integration testing

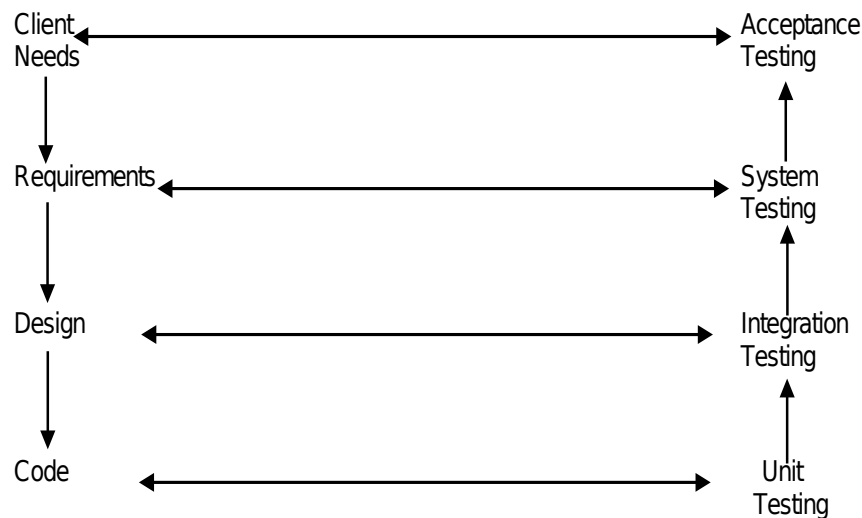


Figure 2.1: Levels of testing

These different levels (figure 2.1) of testing attempt to detect different types of faults.

Unit testing is, essentially, for verification of the code produced during the coding phase, hence the goal is to test the internal logic of the modules.

In *integration testing*, many unit-tested modules are combined into subsystems, which are then tested. The goal here is to see if the modules can be integrated properly.

In *system testing and acceptance testing*, the entire software system is tested. The reference document for this process is the requirements document, and the goal is to see if the software meets its requirements.

Check Your Progress 1.

What are the reasons behind to perform white box testing?

2.3 PRINCIPLES

Software testing is an extremely creative and intellectually challenging task. When testing follows the principles given below, the creative element of test design and execution rivals any of the preceding software development steps.

- i. ***Testing shows the presence of bugs:*** Testing an application can only reveal that one or more defects exist in the application, however, testing alone cannot prove that the

application is error free. Therefore, it is important to design test cases which find as many defects as possible.

- ii. ***Exhaustive testing is impossible:*** Unless the application under test (UAT) has a very simple logical structure and limited input, it is not possible to test all possible combinations of data and scenarios. For this reason, risk and priorities are used to concentrate on the most important aspects to test.
- iii. ***Early testing:*** The sooner we start the testing activities the better we can utilize the available time. As soon as the initial products, such the requirement or design documents are available, we can start testing. It is common for the testing phase to get squeezed at the end of the development lifecycle, i.e. when development has finished, so by starting testing early, we can prepare testing for each level of the development lifecycle. Another important point about early testing is that when defects are found earlier in the lifecycle, they are much easier and cheaper to fix. It is much cheaper to change an incorrect requirement than having to change functionality in a large system that is not working as requested or as designed!
- iv. ***Defect clustering:*** During testing, it can be observed that most of the reported defects are related to small number of modules within a system. i.e. small number of modules contain most of the defects in the system. This is the application of the Pareto Principle to software testing: approximately 80% of the problems are found in 20% of the modules.
- v. ***The pesticide paradox:*** If you keep running the same set of tests over and over again, chances are no more new defects will be discovered by those test cases. Because as the system evolves, many of the previously reported defects will have been fixed and the old test cases do not apply anymore. Anytime a fault is fixed or a new functionality added, we need to do regression testing to make sure the new changed software has not broken any other part of the software. However, those regression test cases also need to change to reflect the changes made in the software to be applicable and hopefully find new defects.
- vi. ***Testing is context dependent:*** Different methodologies, techniques and types of testing is related to the type and nature of the application. For example, a software application in a medical device needs more testing than games software. More importantly a medical device software requires risk based testing, be compliant with medical industry regulators and possibly specific test design techniques. By the same token, a very popular website needs to go through rigorous performance testing as well as

functionality testing to make sure the performance is not affected by the load on the servers.

- vii. ***Absence of errors fallacy:*** Just because testing didn't find any defects in the software, it doesn't mean that the software is ready to be shipped. Were the executed tests really designed to catch the most defects? or where they designed to see if the software matched the user's requirements? There are many other factors to be considered before making a decision to ship the software.

Other principles to note are:

- Testing must be done by an independent party.
- Assign best personnel to the task.
- Test for invalid and unexpected input conditions as well as valid conditions
- Keep software static during test.
- Provide expected test results if possible.

Check Your Progress 2.

What are the Basic Principles of Software Testing?

2.4 UNIT TESTING

Unit testing comprises the set of tests performed by an individual programmer prior to integration of the unit into a larger system. The situation is illustrated as follows:

Coding and debugging → Unit Testing → Integration Testing

A program unit is usually small enough programmer who developed it can test it in great detail, and certainly in greater detail the will be possible when the unit is integrated into an evolving software product.

There are four categories of tests that a programmer will typically perform on a program unit:

- i. ***Functional test*** cases involve exercising the code with nominal input values for which the expected results are known, as well as boundary values (minimum values, maximum values, and values on and just outside the functional boundaries) and special values such as logically related inputs, 1x1 matrices, the identity matrix, files of identical elements, and empty files.
- ii. ***Performances testing*** determines the amount of execution time spend in various parts of the unit, program throughout, response time, and device utilization by the program unit. A certain amount of performance tuning may be done during unit testing.

However, caution must be exercised to avoid expending too much effort on fine-tuning of a program unit that contributes little to the overall performance of the entire system. Performance testing is most productive at the subsystem and system levels.

- iii. **Stress tests** are those tests designed to intentionally break the unit. A great deal can be learned about the strengths and limitations of a program by examining the manner in which a program unit breaks.
- iv. **Structure tests** are concerned with exercising the internal logic of a program and traversing particular execution paths. Some authors refer collectively to functional, performance, and stress testing as “black box” testing, while structure testing is referred to as “white box” or “glass box”. The major activities in structural attesting are deciding which path to exercise, deriving test data to exercise those and measuring the test coverage achieved when the test case are exercised.

Check Your Progress 3.

What errors are commonly found during Unit Testing?

2.5 INTEGRATION TESTING

Integration testing is a software testing methodology used to test individual software components or units of code to verify interaction between various software components and detect interface defects. Components are tested as a single group or organized in an iterative manner. After the integration testing has been performed on the components, they are readily available for system testing.

Integration is a key software development life cycle (SDLC) strategy. Generally, small software systems are integrated and tested in a single phase, whereas larger systems involve several integration phases to build a complete system, such as integrating modules into low-level subsystems for integration with larger subsystems. Integration testing encompasses all aspects of a software system's performance, functionality and reliability.

Most unit-tested software systems are comprised of integrated components that are tested for error isolation due to grouping. Module details are presumed accurate, but prior to integration testing, each module is separately tested via partial component implementation, also known as a stub.

The two main integration testing strategies are as follows:

- Bottom-Up: Involves low-level component testing, followed by high-level components. Testing continues until all hierarchical components are tested. Bottom-up testing facilitates efficient error detection.
- Top-Down: Involves testing the top integrated modules first. Subsystems are tested individually. Top-down testing facilitates detection of lost module branch links.

The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items.

Bottom-up integration is the traditional strategy to integrate the components of a software system into a functioning whole. Bottom-up integration consists of unit testing, followed by subsystem testing, followed by testing of the entire system. Unit testing has the goal of discovering errors in the individual modules of the system. Modules are tested in isolation from one another in an artificial environment known as a “test harness,” which consists of the driver programs and data necessary to exercise the modules. Unit testing should be as exhaustive as possible to ensure that each representative handled by each module has been tested. Unit testing is eased by a system structure that is composed of small, loosely coupled modules.

A subsystem consists of several modules that communicate with each other through well-defined interfaces. Normally, a subsystem implements a major segment operation of the interfaces between modules in the subsystem. Both control and of subsystem testing: lower level subsystems are successively combined to form higher-level subsystems. In most software systems, exhaustive testing of subsystem capabilities is not feasible due to the combinational complexity of the module interfaces; therefore, test cases must be carefully chosen to exercise the interfaces in the desired manner.

System testing is concerned with subtleties in the interfaces, decision logic, control flow, recovery procedures, throughput, capacity, and timing characteristics of the entire system. Careful test planning is required to determine the extent and nature of system testing to be performed and to establish criteria by which the results will be evaluated.

Disadvantages of bottom-up testing include the necessity to write and debug test harness for the modules and subsystems, and the level of complexity that results from combining modules and subsystems into larger and larger units. The extreme case of complexity results when each module is unit tested in isolation and “big bang” approach to integration testing. The main problem with big-bang integration is the difficulty of isolating the sources of error.

Test harnesses provide data environments and calling sequences for the routines and subsystems that are being tested in isolation. Test harness preparation can amount to 50 per cent or more of the coding and debugging effort for a software product.

Top-down integration starts with the main routine and one or two immediately subordinate routines in the system structure. After this top-level, when “skeleton” has been thoroughly tested, it becomes the test harness for its immediately subordinate routines. Top-down integration requires the use of program stubs to simulate the effect of lower-level routines that are called by those being tested.

2.5.1 TOP-DOWN INTEGRATION

Method

- The control module is implemented and tested first.
- Imported modules are represented by surrogate modules.
- Surrogates have the same interfaces as the imported modules and simulate their input/output behaviour.
- After the test of the control module, all other modules of the software systems are tested in the same way; i.e. their operations are represented by surrogate procedures until the development has progressed enough to allow implementation and testing of the operations.
- The test advances stepwise with the implementation. Implementation and phases merge, and the integration test of subsystems becomes superfluous.

The advantages

- Design errors are detected as early as possible, saving development time and costs because corrections in the module design can be made before their implementation.
- The characteristics of a software system are evident from the start, which enables a simple test of the development state and the acceptance by the user.
- The software system can be tested thoroughly from the start with test cases without providing (expensive) test environments.

The drawbacks

- Strict top-down testing proves extremely difficult because designing usable surrogate objects can prove very complicated, especially for complex operations.
- Errors in lower hierarchy levels are hard to localize.

2.5.2 BOTTOM-UP INTEGRATION

Method

- *Bottom-up testing* inverts the top-down approach.
- First those operations are tested that require no other program components; then their integration to a module is tested.
- After the module test the integration of multiple (tested) modules to a subsystem is tested, until finally the integration of the subsystems, i.e., the overall system, can be tested.

The advantages

- The advantages of bottom-up testing prove to be the drawbacks of top-down testing (and vice versa).
- The bottom-up test method is solid and proven. The objects to be tested are known in full detail. It is often simpler to define relevant test cases and test data.
- The bottom-up approach is psychologically more satisfying because the tester can be certain that the foundations for the test objects have been tested in full detail.

The drawbacks

- The characteristics of the finished product are only known after the completion of all implementation and testing, which means that design errors in the upper levels are detected very late.
- Testing individual levels also inflicts high costs for providing a suitable test environment.

Check Your Progress 4.

What are the approaches of integration testing?

2.6 SYSTEM TESTING

System testing is the type of testing to check the behaviour of a complete and fully integrated software product based on the software requirements specification (SRS) document. The main focus of this testing is to evaluate Business / Functional / End-user requirements.

This is black box type of testing where external working of the software is evaluated with the help of requirement documents & it is totally based on Users point of view. For this type of testing do not required knowledge of internal design or structure or code.

This testing is to be carried out only after System Integration Testing is completed where both Functional & Non-Functional requirements are verified.

In the integration testing testers are concentrated on finding bugs/defects on integrated modules. But in the Software System Testing testers are concentrated on finding bugs/defects based on software application behaviour, software design and expectation of end user.

Importance of system testing

- In Software Development Life Cycle the System Testing is performed as the first level of testing where the System is tested as a whole.
- In this step of testing check if system meets functional requirement or not.
- System Testing enables you to test, validate and verify both the Application Architecture and Business requirements.
- The application/System is tested in an environment that particularly resembles the effective production environment where the application/software will be lastly deployed.

Entry Criteria for System Testing:

- Unit testing should be finished.
- Integration of modules should be fully integrated.
- As per the specification document software development is completed.
- Testing environment is available for testing (similar to Staging environment)

Steps of System testing

Step 1) First & important step is preparation of System Test Plan

Step 2) Second step is to creation Test Cases

Step 3) Creation of test data which used for System testing.

Step 4) Automated test case execution.

Step 5) Execution of normal test case & update test case if using any test management tool.

Step 6) Bug Reporting, Bug verification & Regression testing.

Step 7) Repeat testing life cycle (if required).

Types of System Testing

There are more than 50 types of System Testing. Below are some types of system testing used in any software development-

- **Usability Testing** - Usability testing mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives
- **Load Testing** - Load testing is necessary to know that a software solution will perform under real life loads.
- **Regression Testing** - Regression testing involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.
- **Recovery Testing** - Recovery testing is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.
- **Migration Testing** - Migration testing is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.
- **Functional Testing** - Also known as functional completeness testing, functional testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.
- **Hardware/Software Testing** - IBM refers to Hardware/Software testing as "HW/SW Testing". This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

Check Your Progress 5.

What do system testing do?

2.7 SUMMARY

In this section we discuss why we used testing the software, how we can we test, what type of case, criteria and approaches are used to developed the software. Software testing helps in finalizing the software application or product against business and user requirements. It is very important to have good test coverage in order to test the software application completely and make it sure that it's performing well and as per the specifications.

While determining the test coverage the test cases should be designed well with maximum possibilities of finding the errors or bugs. The test cases should be very effective. This objective can be measured by the number of defects reported per test cases. Higher the number of the defects reported the more effective are the test cases.

Once the delivery is made to the end users or the customers they should be able to operate it without any complaints. In order to make this happen the tester should know as how the customers are going to use this product and accordingly they should write down the test scenarios and design the test cases. This will help a lot in fulfilling all the customer's requirements.

Software testing makes sure that the testing is being done properly and hence the system is ready for use. Good coverage means that the testing has been done to cover the various areas like functionality of the application, compatibility of the application with the OS, hardware and different types of browsers, performance testing to test the performance of the application and load testing to make sure that the system is reliable and should not crash or there should not be any blocking issues. It also determines that the application can be deployed easily to the machine and without any resistance. Hence the application is easy to install, learn and use.

2.8 EXERCISE

- 1) What is the difference between black-box testing and white-box testing?
- 2) Define the different type of system testing?
- 3) Write down the advantages and disadvantages of top down integration testing.
- 4) Write down the advantages and disadvantages of bottom up integration testing.
- 5) What are the testing principles the software engineer must apply while performing the software testing?

UNIT-3 TYPES OF TESTING

3.0 Introduction

3.1 Objective

3.2 Types of Testing

- 3.2.1** Installation Testing
- 3.2.2** Compatibility Testing
- 3.2.3** Sanity and Smoke Testing
- 3.2.4** Regression Testing
- 3.2.5** Validation Testing
- 3.2.6** Alpha Testing
- 3.2.7** Beta Testing
- 3.2.8** Acceptance Testing
- 3.2.9** Recovery Testing
- 3.2.10** Security Testing
- 3.2.11** Stress Testing
- 3.2.12** Performance Testing

3.3 Summary

3.4 Exercise

3.0 INTRODUCTION

Testing should systematically uncover different classes of errors in a minimum amount of time and with a minimum amount of effort. A secondary benefit of testing is that it demonstrates that the software appears to be working as stated in the specifications. The data collected through testing can also provide an indication of the software's reliability and quality. But, testing cannot show the absence of defect -- it can only show that software defects are present.













3.1 OBJECTIVE

The objectives of this unit are:

- a) meets the requirements that guided its design and development.
- b) works as expected.
- c) can be implemented with the same characteristics.
- d) satisfies the needs of stakeholders.

3.2 TYPES OF TESTING

There are several types of testing. Some are as follows:

-  Installation Testing
-  Compatibility Testing
-  Sanity and Smoke Testing
-  Regression Testing
-  Validation Testing
-  Alpha Testing
-  Beta Testing
-  Acceptance Testing
-  Recovery Testing
-  Security Testing
-  Stress Testing
-  Performance Testing

3.2 .1 INSTALLATION TESTING

This type of testing assures that the system is installed correctly and working at actual customer's hardware.

Check Your Progress 1.

What are the steps carried out in installation testing?

3.2.2 COMPATIBILITY TESTING

A common cause of software failure is a lack of its compatibility with other application software, operating systems, or target environments that differ from the original. Compatibility testing is one of the test types performed by testing team. Compatibility testing checks if the software can be run on different hardware, operating system, bandwidth, databases, web servers, application servers, hardware peripherals, emulators, different configuration, processor, different browsers and different versions of the browsers etc.,

3.2.3 SANITY AND SMOKE TESTING

Sanity testing determines whether it is reasonable to proceed with further testing.

Smoke testing consists of minimal attempts to operate the software, designed to determine whether there are any basic problems that will prevent it from working at all.

Check Your Progress 2.

What are the benefits of Smoke Testing?

3.2.4 REGRESSION TESTING

When some errors occur in a program then these are rectified. For rectification of these errors, changes are made to the program. Due to these changes some other errors may be incorporated in the program. Therefore, all the previous test cases are tested again. This type of testing is called regression testing.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that supports it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.
- As integration testing proceeds, the number of regression tests can grow quite large.

Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Check Your Progress 3.

List the steps for regression test.

3.2.5 VALIDATION TESTING

Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

A product can pass while verification, as it is done on the paper and no running or functional application is required. But, when same points which were verified on the paper is actually developed then the running application or product can fail while validation. This may happen because when a product or application is built as per the specification but these specifications are not up to the mark hence they fail to address the user requirements.

Advantages of Validation:

- During verification if some defects are missed then during validation process it can be caught as failures.
- If during verification some specification is misunderstood and development had happened then during validation process while executing that functionality the difference between the actual result and expected result can be understood.

- Validation is done during testing like feature testing, integration testing, system testing, load testing, compatibility testing, stress testing, etc.
- Validation helps in building the right product as per the customer's requirement and helps in satisfying their needs.

Validation is basically done by the testers during the testing. While validating the product if some deviation is found in the actual result from the expected result then a bug is reported or an incident is raised.

If the validation tests are carried out by a third party, they are known as independent validation and verification. The developer needs to provide the user manual to the third party tester. This manual should clearly contain the standard working conditions of the software. The user manual should have the various working conditions of the software, so that the tester can simulate real-life conditions. These third party organizations submit a validation report to the developer after the software is tested. The developer, upon receipt of this report, makes the desired changes to the software, and again tests it to check whether the customer needs are met or not.

Software validation testing is an important part of the software development life cycle (SDLC), apart from verification, debugging, and certification. Validation testing ensures that the software meets the quality standards set by the customer, and that the product meets customer requirements.

Check Your Progress 4.

What are the conditions that exist

3.2.6 ALPHA TESTING

Alpha testing is a type of acceptance testing; performed to identify all possible issues/bugs before releasing the product to everyday users or public. The focus of this testing is to simulate real users by using black box and white box techniques. The aim is to carry out the tasks that a typical user might perform. Alpha testing is carried out in a lab environment and usually the testers are internal employees of the organization. To put it as simple as possible, this kind of testing is called alpha only because it is done early on, near the end of the development of the software, and before beta testing as in figure 3.1.

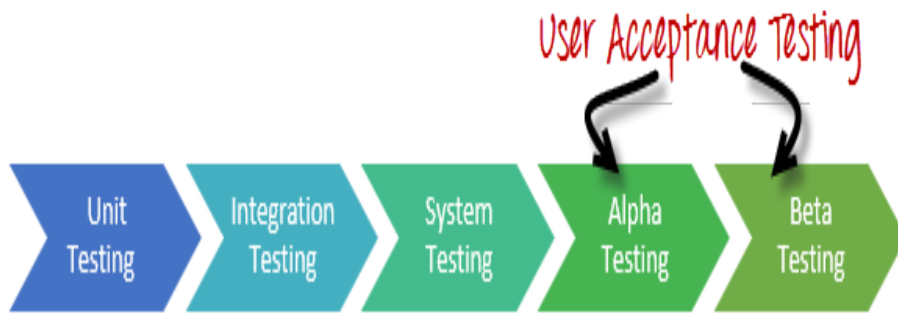


Figure 3.1: Position of Alpha & Beta Testing

Entry Criteria for Alpha testing:

- Software requirements document or Business requirements specification
- Test Cases for all the requirements
- Testing Team with good knowledge about the software application
- Test Lab environment setup
- QA Build ready for execution
- Test Management tool for uploading test cases and logging defects
- Traceability Matrix to ensure that each design requirement has at least one test case that verifies it

Exit Criteria for Alpha testing

- All the test cases have been executed and passed.
- All severity issues need to be fixed and closed
- Delivery of Test summary report
- Make sure that no more additional features can be included
- Sign off on Alpha testing

Advantages of Alpha Testing:

- Provides better view about the reliability of the software at an early stage
- Helps simulate real time user behaviour and environment.
- Detect many showstopper or serious errors
- Ability to provide early detection of errors with respect to design and functionality

Disadvantages of Alpha Testing:

- In depth functionality cannot be tested as software is still under development stage. Sometimes developers and testers are dissatisfied with the results of alpha testing.

Check Your Progress 5.

Which testing is performed for Virtual Environment?

3.2.7 BETA TESTING

Beta Testing of a product is performed by "real users" of the software application in a "real environment" and can be considered as a form of external user acceptance testing.

Beta version of the software is released to a limited number of end-users of the product to obtain feedback on the product quality. Beta testing reduces product failure risks and provides increased quality of the product through customer validation.

It is the final test before shipping a product to the customers. Direct feedback from customers is a major advantage of Beta Testing. This testing helps to tests the product in real time environment.

Types of Beta Testing

There are different types of Beta tests, and they are as follows:

- ***Traditional Beta testing***: Product is distributed to the target market, and related data is gathered in all aspects. This data can be used for Product improvement.
- ***Public Beta Testing***: Product is publicly released to the outside world via online channels and data can be gathered from anyone. Based on feedback, product improvements can be done. For example, Microsoft conducted the largest of all Beta Tests for its OS -- Windows 8 before officially releasing it.
- ***Technical Beta Testing***: Product is released to the internal group of an organization and gathers feedback/data from the employees of the organization.
- ***Focused Beta***: Product is released to the market for gathering feedback on specific features of the program. For example, important functionality of the software.
- ***Post release Beta***: Product is released to the market and data is gathered to make improvements for the future release of the product.

Entrance criteria for Beta Testing:

- Sign off document on Alpha testing

- Beta version of the software should be ready
- Environment ready to release the software application to the public
- Tool to capture real time faults

Exit Criteria for Beta Testing:

- All major and minor issues are closed
- Feedback report should be prepared from public
- Delivery of Beta test summary report

Advantages Beta Testing

- Reduces product failure risk via customer validation.
- Beta Testing allows a company to test post-launch infrastructure.
- Improves product quality via customer feedback
- Cost effective compared to similar data gathering methods
- Creates goodwill with customers and increases customer satisfaction

Disadvantages Beta Testing

- Test Management is an issue. As compared to other testing types which are usually executed inside a company in a controlled environment, beta testing is executed out in the real world where you seldom have control.
- Finding the right beta users and maintaining their participation could be a challenge

Comparison of Alpha and Beta Testing

Alpha Testing	Beta Testing
Alpha testing performed by Testers who are usually internal employees of the organization	Beta testing is performed by Clients or End Users who are not employees of the organization
Alpha Testing performed at developer's site	Beta testing is performed at client location or end user of the product
Reliability and security testing are not performed in-depth Alpha Testing	Reliability, Security, Robustness are checked during Beta Testing
Alpha testing involves both the white box and black box techniques	Beta Testing typically uses black box testing
Alpha testing requires lab environment or	Beta testing doesn't require any lab environment

testing environment	or testing environment. Software is made available to the public and is said to be real time environment
Long execution cycle may be required for Alpha testing	Only few weeks of execution are required for Beta testing
Critical issues or fixes can be addressed by developers immediately in Alpha testing	Most of the issues or feedback is collected from Beta testing will be implemented in future versions of the product
Alpha testing is to ensure the quality of the product before moving to Beta testing	Beta testing also concentrates on quality of the product, but gathers users input on the product and ensures that the product is ready for real time users.

Check Your Progress 6.

How beta testing improve product quality?

3.2.8 ACCEPTANCE TESTING

Acceptance testing is a formal type of software testing that is performed by end user when the features have been delivered by developers. The aim of this testing is to check if the software confirms to their business needs and to the requirements provided earlier.

Acceptance Criteria

Acceptance criteria are defined on the basis of the following attributes

- Functional Correctness and Completeness
- Data Integrity
- Data Conversion
- Usability
- Performance
- Timeliness
- Confidentiality and Availability
- Installability and Upgradability
- Scalability
- Documentation

Acceptance Test Plan - Attributes

The acceptance test activities are carried out in phases. Firstly, the basic tests are executed, and if the test results are satisfactory then the execution of more complex scenarios are carried out.

The Acceptance test plan has the following attributes:

- Introduction
- Acceptance Test Category
- operation Environment
- Test case ID
- Test Title
- Test Objective
- Test Procedure
- Test Schedule
- Resources

The acceptance test activities are designed to reach at one of the conclusions:

1. Accept the system as delivered
2. Accept the system after the requested modifications have been made
3. Do not accept the system

Acceptance Test Report - Attributes

The Acceptance test Report has the following attributes:

- Report Identifier
- Summary of Results
- Variations
- Recommendations
- Summary of To-do List
- Approval Decision

Check Your Progress 7.

List acceptance test plan attribute.

3.2.9 RECOVERY TESTING

Many computer-based systems must recover from faults and resume operation within a pre-specified time. In some cases, a system may be fault tolerant; that is, processing faults must

not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If the recovery is automated (performed by system itself), re-initialization mechanisms, data recovery, and restart are each evaluated for correctness. If the recovery requires human intervention, the mean time to repair is evaluated to determine whether it is within acceptable limits.

Recovery testing is a type of non-functional testing technique performed in order to determine how quickly the system can recover after it has gone through system crash or hardware failure. Recovery testing is the forced failure of the software to verify if the recovery is successful.

Steps of Recovery Plan:

- Determining the feasibility of the recovery process.
- Verification of the backup facilities.
- Ensuring proper steps are documented to verify the compatibility of backup facilities.
- Providing Training within the team.
- Demonstrating the ability of the organization to recover from all critical failures.
- Maintaining and updating the recovery plan at regular intervals.

Check Your Progress 8.

What is non-functional testing?

3.2.10 SECURITY TESTING

Any computer-based system that manages sensitive information or causes actions that can harm or benefit individuals is a target for improper or illegal penetration.

Security testing attempts to verify that protection mechanism built into a system will protect it from unauthorized penetration. During security testing, the tester plays the role of the individual who desires to penetrate the system. The tester may attack the system with custom software designed to break down any defences that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to find the key to system entry; and so on.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost greater than the value of the information that will be obtained in order to deter potential threats.

The prime objective of security testing is to find out how vulnerable a system may be and to determine whether its data and resources are protected from potential intruders. Online transactions have increased rapidly of late making security testing as one of the most critical areas of testing for such web applications. Security testing is more effective in identifying potential vulnerabilities when performed regularly.

Normally, security testing has the following attributes:

- Authentication
- Authorization
- Confidentiality
- Availability
- Integrity
- Non-repudiation
- Resilience

Why Security Testing

System testing, in the current scenario, is a must to identify and address web application security vulnerabilities to avoid any of the following:

- Loss of customer trust.
- Disturbance to your online means of revenue generation/collection.
- Website downtime, time loss and expenditures in recovering from damage.
- Cost associated with securing web applications against future attacks.
- Related legal implications and fees for having lax security measures in place.

Check Your Progress 9.

What is the need of security testing?

3.2.11 STRESS TESTING

Stress testing refers to the testing of software or hardware to determine whether its performance is satisfactory under any extreme and unfavourable conditions, which may occur as a result of heavy network traffic, process loading, under-clocking, overclocking and maximum requests for resource utilization.

Most systems are developed under the assumption of normal operating conditions. Thus, even if a limit is crossed, errors are negligible if the system undergoes stress testing during development.

Stress tests are designed to confront program functions with abnormal situations. Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,

- i. Special tests may be designed that generate 10 interrupts are seconds, when one or two is the average rate;
 - ii. Input data rates may be increased by an order of magnitude to determine how input functions will respond;
 - iii. Test cases that require maximum memory or other resources may be executed;
 - iv. Test cases that may cause excessive hunting for disk resident data may be created; or
 - v. Test cases that may cause thrashing in a virtual operating system may be designed.
- The testers attempt to break the program.

Context of Stress testing:

- ***Software:*** Stress testing emphasizes availability and error handling under extremely heavy loads to ensure software does not crash due to insufficient resources. Software stress testing focuses on identified transactions to break transactions, which are heavily stressed during testing, even when a database has no load. The stress testing process loads concurrent users beyond normal system levels to find the system's weakest link.
- ***Hardware:*** Stress testing ensures stability in normal computing environments.
- ***Websites:*** Stress testing determines the limitations of any of the site's functionalities.
- ***CPU:*** Modifications such as over volting, under volting, under locking and over locking are verified to determine whether they can withstand heavy loads by running a CPU-intensive program to test for system crashes or hangs. CPU stress testing is also known as torture testing.

Benefits of Stress Testing

The most significant benefit of stress testing is that you can check the application to see whether it works under any type of stress. Stress testing can uncover many loopholes or weaknesses like memory leaks and even race conditions. Race conditions are the conflicts you will sometimes see, when two tests run concurrently.

A memory leak usually occurs when the test uses the allocated memory and it does not return the said memory space for the memory allocation. This will lead to system failure, as the available memory is eaten up completely. Stress testing may not be a proper type of software testing system. In many cases, many different tests are capable of knowing a software application's ability to perform well in a real-time ambiance.

Stress testing can provide you with the necessary data that is hard to find anywhere else.

Check Your Progress 10.

Write some benefits of stress testing.

3.2.12 PERFORMANCE TESTING

Performance testing, a non-functional testing technique performed to determine the system parameters in terms of responsiveness and stability under various workload. Performance testing measures the quality attributes of the system, such as scalability, reliability and resource usage.

Performance Testing Goal:

The focus of Performance testing is checking a software program for

- Speed - Determines whether the application responds quickly
- Scalability - Determines maximum user load the software application can handle.
- Stability - Determines if the application is stable under varying loads

Performance Testing Techniques:

- ***Load testing*** - It is the simplest form of testing conducted to understand the behaviour of the system under a specific load. Load testing will result in measuring important business critical transactions and load on the database, application server, etc., are also monitored.
- ***Soak testing*** - Soak Testing also known as endurance testing, is performed to determine the system parameters under continuous expected load. During soak tests the parameters such as memory utilization is monitored to detect memory leaks or other performance issues. The main aim is to discover the system's performance under sustained use.

- ***Spike testing*** - Spike testing is performed by increasing the number of users suddenly by a very large amount and measuring the performance of the system. The main aim is to determine whether the system will be able to sustain the workload.

Attributes of Performance Testing:

- Speed
- Scalability
- Stability
- Reliability

Common Performance Problems

- Long Load time
- Poor response time
- Poor scalability
- Bottlenecking

Performance Testing Process

- i. Identify your testing environment
- ii. Identify the performance acceptance criteria
- iii. Plan & design performance tests
- iv. Configuring the test environment
- v. Implement test design
- vi. Analyse, tune and retest

Check Your Progress 11.

Give any three types of performance test.

3.3 SUMMARY

In this unit lots of testing are defined. This is not necessary to use all the testing in single software. Every testing has its own characteristics, requirement and limitations. These testing are used according to the need and requirements of software as well as testing team.

3.4 EXERCISE

- 1) What is the need of security testing?

- 2) Explain the techniques of performance testing.
- 3) What is the difference between alpha testing and beta testing?
- 4) Distinguish between verification and validation.

UNIT-4 REENGINEERING

4.0 Introduction

4.1 Objective

4.2 Concept of Re-Engineering

4.3 Concept of Reverse Engineering

4.4 Concept of Restructuring

4.5 Concept of Forward Engineering

4.6 Summary

4.7 Exercise

4.0 INTRODUCTION

The essence of software re-engineering is to improve or transform existing software so that it can be understood, controlled, and used anew. The need for software re-engineering has increased greatly, as heritage software systems have become obsolescent in terms of their architecture, the platforms on which they run, and their suitability and stability to support evolution to support changing needs. Software re-engineering is important for recovering and reusing existing software assets, putting high software maintenance costs under control, and establishing a base for future software evolution. Basically, re-engineering is taking existing legacy software that has become expensive to maintain or whose system architecture or implementation are obsolete, and redoing it with current software and/or hardware technology. The difficulty lies in the understanding of the existing system. Usually requirements, design and code documentation is no longer available, or is very out of date, so it is unclear what functions are to be moved. Often the system contains functions that are no longer needed, and those should not be moved to the new system.

4.1 OBJECTIVE

The objectives of this unit are:

- a) to obtain quantum gains in the performance of the process in terms of time, cost, output, quality to customers
- b) to simplify and streamline the process
- c) to obtain dramatic improvement in operational effectiveness.

4.2 CONCEPT OF RE-ENGINEERING

Re-engineering is the examination, analysis and alteration of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form. The process typically encompasses a combination of other processes such as reverse engineering, re-documentation, restructuring, translation, and forward engineering. The goal is to understand the existing software (specification, design, implementation) and then to re-implement it to improve the system's functionality, performance or implementation.

- *Abstraction level* – ideally want to be able to derive design information at the highest level possible
- *Completeness* – level of detail provided at a given abstraction level
- *Interactivity* – degree to which humans are integrated with automated reverse engineering

tools

- *Directionality* – one-way means the software engineer doing the maintenance activity is given all information extracted from source code, two-way means the information is fed +to a reengineering tool that attempts to regenerate the old program
- *Extract abstractions* – meaningful specification of processing performed is derived from old source code

Re-engineering Objectives

The number of large systems being built from scratch is diminishing, while the number of legacy systems in use is very high. While the functionality of existing systems remains constant, the context of new systems, such as the application environment, system level hardware and software, are different. Enhancements to the functionality of the existing systems may also be needed, but although the re-engineering effort may be configured for enhancements, they should not be incorporated until after the re-engineering is complete. This allows for comparison of functionality between the existing system and the new system. The problem is that systems currently in use, "legacy" systems, have become lacking in good design structure and code organization, making changes to the software difficult and costly. Corporations do not want to "trash" these systems because there are many built in subtle business application processes that have evolved over time that would be lost. Often the developers of the legacy systems are not available to verify or explain this information; the only source is the current software code. The original expense of developing the logic and components of the software systems should not be wasted, so reuse through re-engineering is desired. The challenge in software re-engineering is to take existing systems and instill good software development methods and properties, generating a new target system that maintains the required functionality while applying new technologies. Although specific objectives of a re-engineering task are determined by the goals of the corporations, there are four general re-engineering objectives:

- Preparation for functional enhancement
- Improve maintainability
- Migration
- Improve reliability

Goals of Reengineering

- *Port to other Platform*- when hardware or software support becomes obsolete

- *Design extraction*- to improve maintainability, portability, etc.
- *Exploitation of New Technology*- new language features, standards, libraries, etc. It is used when tools to support restructuring are readily available

Software Reengineering Activities

- *Inventory analysis* – sorting active software applications by business criticality, longevity, current maintainability, and other local criteria helps to identify reengineering candidates
- *Document restructuring*– need to decide to live with weak documentation, update poor documents if they are used, or fully rewrite the documentation for critical systems focusing on the "essential minimum"
- *Reverse engineering* – process of design recovery - analyzing a program in an effort to create a representation of the program at some abstraction level higher than source code
- *Code restructuring* – source code is analysed and violations of structured programming practices are noted and repaired, the revised code also needs to be reviewed and tested
- *Data restructuring* – usually requires full reverse engineering, current data architecture is dissected and data models are defined, existing data structures are reviewed for quality
- *Forward engineering* – also called reclamation or renovation, recovers design information from existing source code and uses this information to reconstitute the existing system to improve its overall quality and/or performance

The complete lifecycle of Software Re-Engineering includes:

- *Product Management*: Risks analysis, root cause analysis, business analysis, requirements elicitation and management, product planning and scoping, competitive analysis
- *Research and Innovation*: Definition of a problem, data gathering and analysis, identifying a solution and developing best-of-breed or innovative algorithms, verification of quality for data and results, patent preparation
- *Product Development*: Technology analysis and selection, software architecture and design, data architecture, deployment architecture, prototyping and production code development, comprehensive software testing, data quality testing, and product packaging and deployment preparation
- *Product Delivery and Support*: Hardware/Platform analysis and selection, deployment and release procedures definition, installations and upgrades, tracking support issues, organizing maintenance releases.

🏢 **Project Management:** Brings efficiency and productivity to your software re-engineering project by utilizing modern, practical software project management, software quality assurance, data quality assurance, and advanced risk management techniques.

Software Development Levels of Abstraction in Re-engineering

Levels of Abstraction that underlie the software development process also underlie the re-engineering process. Each level corresponds to a phase in the development life cycle and defines the software system at a particular level of detail (or abstraction) is depicted in figure 4.1.

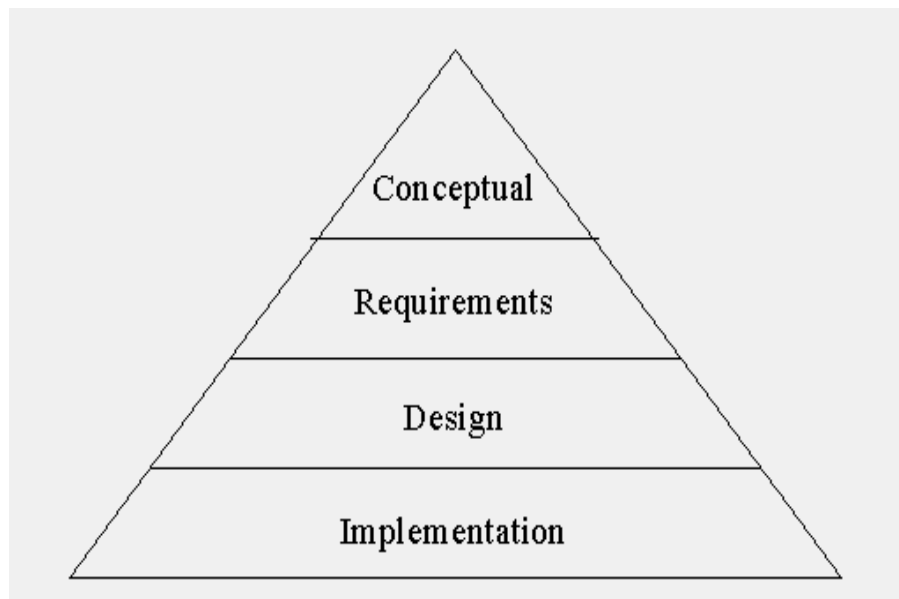


Figure 4.1: Levels of Abstraction

General Model for Software Re-engineering

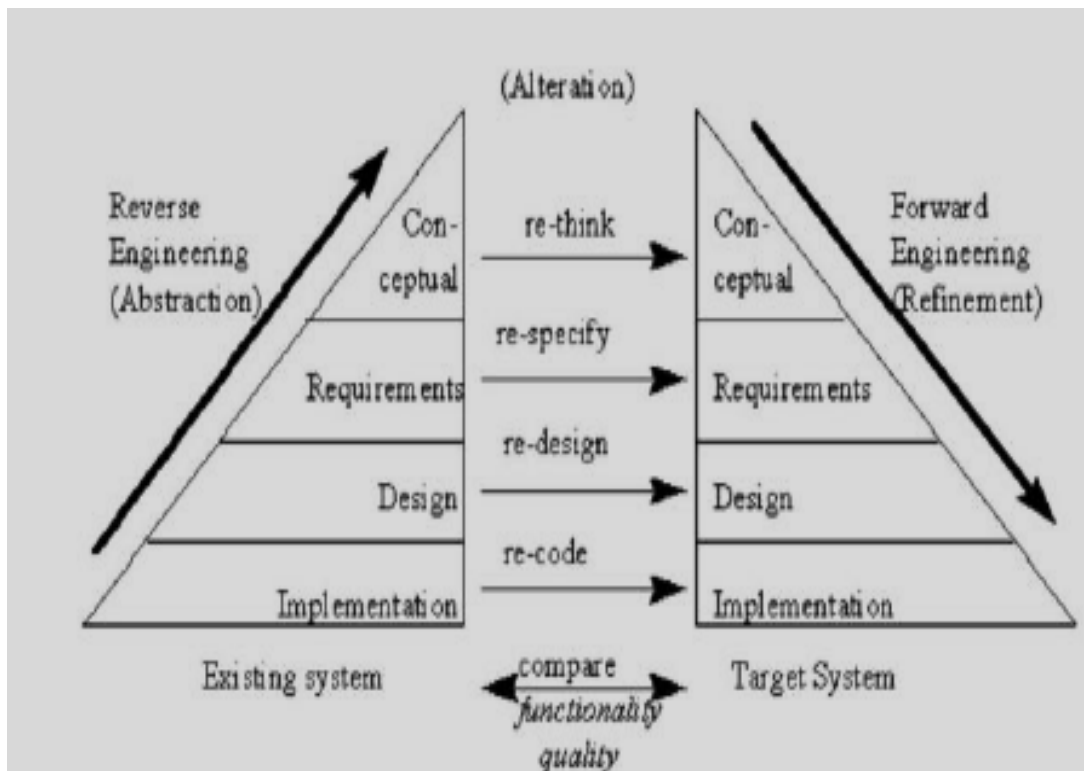


Figure 4.2: General Model for Software Re-engineering

Re-engineering starts with the source code of an existing legacy system and concludes with the source code of a target system as depicted in figure 4.2. This process may be as simple as using a code translation tool to translate the code from one language to another (FORTRAN to C) or from one operating system to another (UNIX to DOS). On the other hand, the re-engineering task may be very complex, using the existing source code to recreate the design, identify the requirements in the existing system then compare them to current requirements, removing those no longer applicable, restructure and redesign the system (using object-oriented design), and finally code the new target system.

Re-engineering advantages

- *Reduced risk:* There is a high risk in new software development. There may be development problems, staffing problems and specification problems
- *Reduced cost:* The cost of re-engineering is often significantly less than the costs of developing new software.

Generic Reengineering Process

As represented in figure 4.3, re-engineering process is defined as-

- *Requirement analysis:* analyse on which parts of your requirements have changed

- *Model capture*: reverse engineer from the source-code into a more abstract form, typically some form of a design model
- *Problem detection*: identify design problems in that abstract model
- *Problem resolution*: propose an alternative design that will solve the identified problem
- *Program transformations*: make the necessary changes to the code, so that it adheres to the new design yet preserves all the required functionality

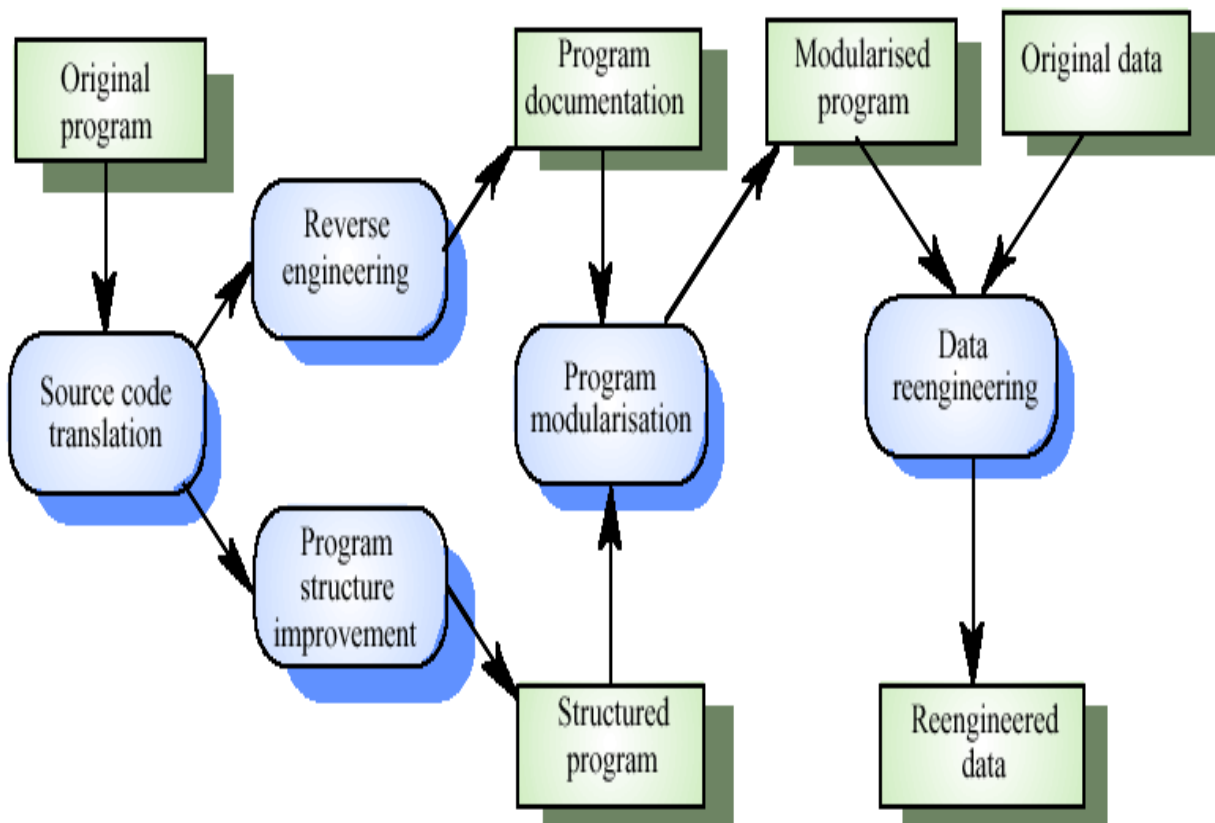


Figure 4.3: Re-engineering process

Re-engineering cost factors

- The quality of the software to be re-engineered
- The tool support available for re-engineering
- The extent of the data conversion which is required
- The availability of expert staff for re-engineering

Re-engineering Phases and Tasks

There is a core process that every organization should follow when re-engineering. Reengineering poses its own technical challenges and without a comprehensive development

process will waste time and money. Automation and tools can only support this process, not pre-empt it. The re-engineering process can be broken into five phases and associated tasks, starting with the initial phase of determining the feasibility and cost effectiveness of reengineering, and concluding with the transition to the new target system.

These five reengineering development phases are:

- Re-engineering Team Formation
- Project Feasibility Analysis
- Analysis and Planning
- Re-engineering Implementation
- Testing and Transition

Check Your Progress I.

When to Re-Engineer?

4.3 CONCEPT OF REVERSE ENGINEERING

Reverse engineering is the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. In reverse engineering, the requirements and the essential design, structure and content of the legacy system must be recaptured as depicted in figure 4.4 . In addition to capturing technical relationships and interactions, information and rules about the business application and process that have proved useful in running the business must also be retrieved. This involves extracting design artefacts and building or synthesizing abstractions that are less implementation dependent. The key objectives in reverse engineering are to generate alternative views, recover lost information, detect side effects, synthesize higher abstractions, and facilitate reuse. The effectiveness of this process will affect the success of the reengineering project. Reverse engineering does not involve changes to the system or creating a new system, it is the process of examination without changing its overall functionality.

Principles of reverse engineering

- Systematic process of acquiring important design factors and information regarding engineering aspects from an existing product

- A process which analyses a product/technology to find out the design aspects and its functions
- A kind of analysis which engages an individual in a process of constructive learning of design and its functionality of systems and products

Reverse Engineering Activities

- Understanding data
 - internal data structures – program code is examined with the intention of grouping related program variables
 - database structure – often done prior to moving from one database paradigm to another (e.g. flat file to relational)
- Understanding processing - source code is analysed to at varying levels of detail (system, program, component, pattern, statement) to understand procedural abstractions and overall functionality

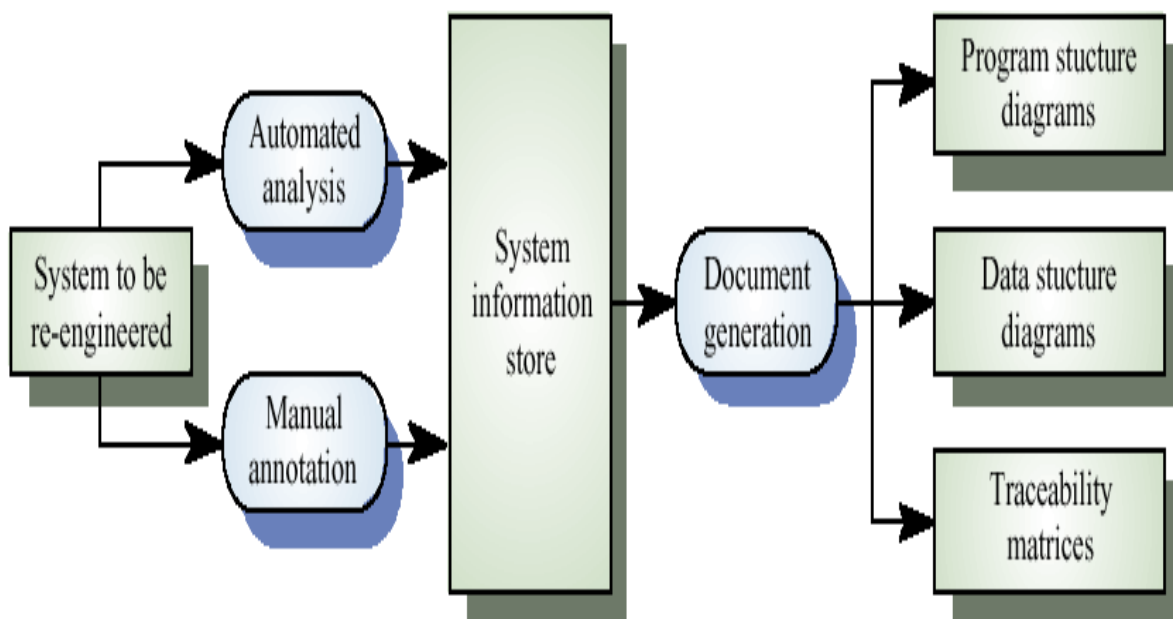


Figure 4.4: Reverse Engineering Process

Objectives of reverse engineering:

- To recover lost information
- To facilitate migration between platforms
- To improve and/or provide new documentation
- To extract reusable components
- To reduce maintenance effort

- To cope with complexity
- To detect side effects
- To assist migration to a case environment
- To develop similar or competitive products

Goals of reverse engineering:

- Cope with complexity
- Need techniques to understand large, complex systems
- Recover lost information
- Extract what changes have been made and why
- Detect side effects
- Help understand ramifications of changes
- Synthesize higher abstractions
- Identify latent abstractions in software
- Facilitate reuse
- Detect candidate reusable artefacts and components

Uses of Reverse Engineering

- ***Interfacing.*** Reverse engineering can be used when a system is required to interface to another system and how both systems would negotiate is to be established. Such requirements typically exist for interoperability.
- ***Military or commercial espionage.*** Learning about an enemy's or competitor's latest research by stealing or capturing a prototype and dismantling it. It may result in development of similar product, or better countermeasures for it.
- ***Improve documentation shortcomings.*** Reverse engineering can be done when documentation of a system for its design, production, operation or maintenance have shortcomings and original designers are not available to improve it. Reverse engineering of software can provide the most current documentation necessary for understanding the most current state of a software system.
- ***Obsolescence.*** Integrated circuits often seem to have been designed on obsolete, proprietary systems, which means that when those systems can no longer be maintained, the only way to incorporate the functionality into new technology is to reverse-engineer the existing chip and then re-design it using newer tools, and using the understanding gained, as a guide. Another obsolescence originated problem which can be solved by

reverse engineering is the need to support existing, legacy devices which are no longer supported by their OEM. This problem is particularly critical in military operations.

- **Software modernization** - often knowledge is lost over time, which can prevent updates and improvements. Reverse engineering is generally needed in order to understand the 'as is' state of existing or legacy software in order to properly estimate the effort required to migrate system knowledge into a 'to be' state. Much of this may be driven by changing functional, compliance or security requirements.
- **Product security analysis.** To examine how a product works, what are specifications of its components, estimate costs and identify potential patent infringement. Acquiring sensitive data by disassembling and analysing the design of a system component. Intent may be to remove copy protection, circumvention of access restrictions.
- **Bug fixing.** To fix legacy software this is no longer supported by its creators.
- **Creation of unlicensed/unapproved duplicates,** such duplicates are called sometimes clones in the computing domain.
- **Academic/learning purposes.** Reverse engineering for learning purposes may be understand the key issues of an unsuccessful design and subsequently improve the design.
- **Competitive technical intelligence.** Understand what one's competitor is actually doing, versus what they say they are doing.
- **Saving money,** when one finds out what a piece of electronics is capable of, it can spare a user from purchase of a separate product.
- **Repurposing,** in which opportunities to repurpose stuff that is otherwise obsolete can be incorporated into a bigger body of utility.

Reverse engineering of machines

As computer-aided design (CAD) has become more popular, reverse engineering has become a viable method to create a 3D virtual model of an existing physical part for use in 3D CAD, CAM, CAE or other software. The reverse-engineering process involves measuring an object and then reconstructing it as a 3D model. The physical object can be measured using 3D scanning technologies like CMMs, laser scanners, structured light digitizers, or Industrial CT Scanning (computed tomography). The measured data alone, usually represented as a point cloud, lacks topological information and is therefore often processed and modelled into a more usable format such as a triangular-faced mesh, a set of NURBS surfaces, or a CAD model.

Hybrid Modelling is commonly used term when NURBS and parametric modelling are implemented together. Using a combination of geometric and freeform surfaces can provide a powerful method of 3D modelling. Areas of freeform data can be combined with exact geometric surfaces to create a hybrid model. A typical example of this would be the reverse engineering of a cylinder head, which includes freeform cast features, such as water jackets and high tolerance machined areas.

Reverse engineering is also used by businesses to bring existing physical geometry into digital product development environments, to make a digital 3D record of their own products, or to assess competitors' products. It is used to analyse, for instance, how a product works, what it does, and what components it consists of, estimate costs, and identify potential patent infringement, etc. Value engineering is a related activity also used by businesses. It involves de-constructing and analysing products, but the objective is to find opportunities for cost cutting.

Reverse engineering of software

The term reverse engineering as applied to software means different things to different people. Reverse engineering is the process of analysing a subject system to create representations of the system at a higher level of abstraction. It can also be seen as "going backwards through the development cycle". In this model, the output of the implementation phase (in source code form) is reverse-engineered back to the analysis phase, in an inversion of the traditional waterfall model. Another term for this technique is program comprehension. Reverse engineering is a process of examination only: the software system under consideration is not modified (which would make it re-engineering). Software anti-tamper technology like obfuscation is used to deter both reverse engineering and re-engineering of proprietary software and software-powered systems.

In practice, two main types of reverse engineering emerge.

In the first case, source code is already available for the software, but higher-level aspects of the program, perhaps poorly documented or documented but no longer valid, are discovered.

In the second case, there is no source code available for the software, and any efforts towards discovering one possible source code for the software are regarded as reverse engineering. This second usage of the term is the one most people are familiar with. Reverse engineering of software can make use of the clean room design technique to avoid copyright infringement.

On a related note, black box testing in software engineering has a lot in common with reverse engineering. The tester usually has the API, but their goals are to find bugs and undocumented features by bashing the product from outside.

Other purposes of reverse engineering include security auditing, removal of copy protection, circumvention of access restrictions often present in consumer electronics, customization of embedded systems (such as engine management systems), in-house repairs or retrofits, enabling of additional features on low-cost "crippled" hardware (such as some graphics card chip-sets), or even mere satisfaction of curiosity.

Reverse engineering of protocols

Protocols are sets of rules that describe message formats and how messages are exchanged (i.e., the protocol state-machine). Accordingly, the problem of protocol reverse-engineering can be partitioned into two sub problems; message format and state-machine reverse-engineering.

The message formats have traditionally been reverse-engineered through a tedious manual process, which involved analysis of how protocol implementations process messages, but recent research proposed a number of automatic solutions. Typically, these automatic approaches either group observed messages into clusters using various clustering analyses, or emulate the protocol implementation tracing the message processing.

There has been less work on reverse-engineering of state-machines of protocols. In general, the protocol state-machines can be learned either through a process of offline learning, which passively observes communication and attempts to build the most general state-machine accepting all observed sequences of messages, and online learning, which allows interactive generation of probing sequences of messages and listening to responses to those probing sequences. In general, offline learning of small state-machines is known to be NP-complete, while online learning can be done in polynomial time.

Other components of typical protocols, like encryption and hash functions, can be reverse-engineered automatically as well. Typically, the automatic approaches trace the execution of protocol implementations and try to detect buffers in memory holding unencrypted packets.

Reverse engineering of Hardware

Hardware reverse engineering involves taking apart a device to see how it works. For example, if a processor manufacturer wants to see how a competitor's processor works, they can purchase a competitor's processor, disassemble it, and then make a processor similar to it.

However, this process is illegal in many countries. In general, hardware reverse engineering requires a great deal of expertise and is quite expensive.

Benefits of Reverse Engineering for Software Maintenance

- *Corrective change*: abstraction of unnecessary detail gives greater insight into the parts of the program to be corrected. And it is easier to identify defective program components and the source of residual errors
- *Adaptive/perfective change*: Eases understanding of system's components and their interrelationships, showing where new requirements fit and how they relate to existing components. It Extracted information when be used during enhancement of the system or for the development of another product
- *Preventive change*: It brings benefit to future maintenance of a system

Check Your Progress2.

What are the main objectives of reverse engineering?

4.4 CONCEPT OF RESTRUCTURING

Software restructuring is recognized as a promising method to improve logical structure and understand ability of a software system which is composed of modules with loosely-coupled elements. There are several methods of restructuring an ill-structured module at the software maintenance phase. The methods identify modules performing multiple functions and restructure such modules. For identifying the multi-function modules, the notion of the tightly-coupled module that performs a single specific function is formalized. This method utilizes information on data and control dependence, and applies program slicing to carry out the task of extracting the tightly-coupled modules from the multi-function module. The identified multi-function module is restructured into a number of functional strength modules or an informational strength module. The module strength is used as a criterion to decide how to restructure. The methods can also be readily automated and incorporated in a software tool. Restructuring involves examining the existing system and rewriting parts of it to improve its overall structure. Restructuring may be particularly useful when changes are confined to part of the system. Only this part need be restructured. Other parts need not be changed or revalidated. If a program is written in a high-level language, it is possible to restructure that program automatically although the computer time required to do so may be great.

A Theorem has been given on the basis for program restructuring. It says that, any program may be rewritten in terms of simple IF-THEN-ELSE conditionals and WHILE loops and that unconditional GOTO statements were not required.

Method

Step 1. Construct a program flow graph.

Step 2. Apply simplification and transformation techniques to the graph to construct while loops and simple conditional statements.

It may well be that a combination of automatic and manual system restructuring is the best approach. The control structure could be improved automatically and this makes the system easier to understand. The abstraction and data structures of the program may then be discovered, documented and improved using a manual approach. Decisions on whether to restructure or rewrite a program can only be made on a case-by-case basis.

Some of the factors which must be taken into account are

- Is a significant proportion of the system stable and not subject to frequent change? If so, this suggests restructuring rather than rewriting as it is only really necessary to restructure that part of the program which is to be changed.
- Does the program rely on obsolete support software such as compilers, etc.? If so, this suggests it should be rewritten in a modern language as the future availability of the support software cannot be guaranteed.
- Are tools available to support the restructuring process? If not, manual restructuring is the only option.

System restructuring offers an opportunity to control maintenance costs and I believe that it will become increasingly important. The rate of change of hardware development means that many embedded software systems which are still in use must be changed as the hardware on which they execute cannot be supported.

Types of Restructuring

- Code restructuring
- Program logic modelled using Boolean algebra and series of transformation rules are applied to yield restructured logic
- Create resource exchange diagram showing data types, procedure and variables shared between modules, restructure program architecture to minimize module coupling
- Data restructuring

- Analysis of source code
- Data redesign
- Data record standardization
- Data name rationalization
- File or database translation

Check Your Progress3.

How does restructuring help in maintaining a program?

4.5 CONCEPT OF FORWARD ENGINEERING

Forward engineering is the process of building from a high-level model or concept to build in complexities and lower-level details. This type of engineering has different principles in various software and database processes.

Generally, forward engineering is important in IT because it represents the 'normal' development process. For example, building from a model into an implementation language. This will often result in loss of semantics, if models are more semantically detailed, or levels of abstraction.

Forward engineering is thus related to the term 'reverse engineering,' where there is an effort to build backward, from a coded set to a model, or to unravel the process of how something was put together.

It's crucial to note, though, that reverse engineering is also a term widely used in IT to describe attempts to take a software product or other technology apart and inspect how it works. In this type of contrast, forward engineering would be a logical 'forward-moving' design, where reverse engineering would be a form of creative deconstruction.

Some experts provide specific examples of forward engineering, including the use of abstract database models or templates into physical database tables. Other examples include a situation where developers or others make models or diagrams into concrete code classes, or specific code modules.

Check Your Progress4.

How forward engineering is related to reverse engineering?

In this section we discuss Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented. When system changes are mostly confined to part of the system then re-engineer that part and how re-engineering

reduced the cost and risk. Also define forward engineering as well as Reverse engineering which is the process of deriving the system design and specification from its source code .

4.7 EXERCISE

- 1) What is Reengineering? And what are the objectives of reengineering. What are the common mistakes made when beginning reengineering?
- 2) Why does reengineering take so long?
- 3) What is the main difficulty in reengineering in general?
- 4) Write the advantages of re-engineering.
- 5) Define the activities of Re-engineering.
- 6) How Reverse and Forward Engineering is related with Re-engineering. explain.
- 7) Write the goals of reverse engineering. Also write its advantages.
- 8) Write Short Notes on-
 Forward Engineering, Restructuring

UNIT-5 CASE: Computer Aided Software Engineering

5.0 Introduction

- 5.1 Objective**
- 5.2 Tools: What is CASE?**
- 5.3 Building Blocks of CASE**
- 5.4 A Taxonomy of CASE Tools**
- 5.5 Integrated CASE Environments**
- 5.6 The Integration Architecture**
- 5.7 The CASE Repository**
- 5.8 Summary**
- 5.9 Exercise**

5.0 INTRODUCTION

CASE tools are a class of software that automate many of the activities involved in various life cycle phases. For example, when establishing the functional requirements of a proposed application, prototyping tools can be used to develop graphic models of application screens to assist end users to visualize how an application will look after development. Subsequently, system designers can use automated design tools to transform the prototyped functional requirements into detailed design documents. Programmers can then use automated code generators to convert the design documents into code. Automated tools can be used collectively, as mentioned, or individually. For example, prototyping tools could be used to define application requirements that get passed to design technicians who convert the requirements into detailed designs in a traditional manner using flowcharts and narrative documents, without the assistance of automated design software.

5.1 OBJECTIVE

The objectives of this unit are:

- a) to identify the role of CASE tools in the software development process.
- b) to identify the criteria for selecting a CASE tool.
- c) to identify the benefits and limitations of CASE tools.

5.2 Tools: What is CASE?

Computer-aided software engineering (CASE) is the application of a set of tools and methods to a software system with the desired end result of high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems together with automated tools that can be used in the software development process.

Reasons for using case tools:

The primary reasons for using a CASE tool are:

- To increase productivity
- To help produce better quality software at lower cost











Types of tools for CASE are:

- Business process engineering tools.
- Process modelling and management tool

- Project planning tools
- Risk analysis tools
- Project management tools
- Requirement tracing tools
- Metrics management tools
- Documentation tools
- System software tools
- Quality assurance tools
- Database management tools
- Software configuration management tools
- Analysis and design tools
- Interface design and development tools
- Prototyping tools
- Programming tools
- Web development tools
- Integration and testing tools
- Static analysis tools
- Dynamic analysis tools
- Test management tools
- Client/Server testing tools
- Re-engineering tools

Benefits of CASE

Every program you create using the Program Generator automatically includes such as:

-  Data Dictionary
-  User defined codes
-  Vocabulary overrides
-  Action code security
-  Business unit security
-  Standard function exits
-  Function exit and option exit security
-  Cursor sensitive help
-  Program help
-  DREAM Writer

Classification of CASE Tools:

Existing CASE tools can be classified along 4 different dimensions:

1. Life-cycle support
2. Integration dimension
3. Construction dimension
4. Knowledge-based CASE dimension

Applications

- A **CASE repository** is a system developers' database. It is a place where developers can store system models, detailed descriptions and specifications, and other products of system development. Synonyms include dictionary and encyclopedia.
- **Forward engineering** requires the systems analyst to draw system models, either from scratch or from templates. The resulting models are subsequently transformed into program code.
- **Reverse engineering** allows a CASE tool to read existing program code and transform that code into a representative system model that can be edited and refined by the systems analyst.

However, tools that are concerned with analysis and design, and with using design information to create parts (or all) of the software product, are most frequently thought of as CASE tools. CASE applied, for instance, to a database software product, might normally involve:

- Modelling business / real-world processes and data flow
- Development of data models in the form of entity-relationship diagrams
- Development of process and function descriptions

Major Risk Factors: Common CASE risks and associated controls include:

- **Inadequate standardization:** Linking CASE tools from different vendors (design tool from Company X, programming tool from Company Y) may be difficult if the products do not use standardized code structures and data classifications. File formats can be converted, but usually not economically. Controls include using tools from the same vendor, or using tools based on standard protocols and insisting on

demonstrated compatibility. Additionally, if organizations obtain tools for only a portion of the development process, they should consider acquiring them from a vendor that has a full line of products to ensure future compatibility if they add more tools.

- ***Unrealistic expectations:*** Organizations often implement CASE technologies to reduce development costs. Implementing CASE strategies usually involves high start-up costs. Generally, management must be willing to accept a long-term payback period. Controls include requiring senior managers to define their purpose and strategies for implementing CASE technologies.
- ***Slow implementation:*** Implementing CASE technologies can involve a significant change from traditional development environments. Typically, organizations should not use CASE tools the first time on critical projects or projects with short deadlines because of the lengthy training process. Additionally, organizations should consider using the tools on smaller, less complex projects and gradually implementing the tools to allow more training time.
- ***Weak repository controls:*** Failure to adequately control access to CASE repositories may result in security breaches or damage to the work documents, system designs, or code modules stored in the repository. Controls include protecting the repositories with appropriate access, version, and backup controls.

Workbenches: Workbenches integrate two or more CASE tools and support specific software-process activities. Hence they achieve:

- A homogeneous and consistent interface (presentation integration).

- Seamless integration of tools and tool chains (control and data integration).

An example workbench is Microsoft's Visual Basic programming environment. It incorporates several development tools: a GUI builder, smart code editor, debugger, etc. Most commercial CASE products tended to be such workbenches that seamlessly integrated two or more tools. Workbenches also can be classified in the same manner as tools; as focusing on Analysis, Development, Verification, etc. as well as being focused on upper case, lower case, or processes such as configuration management that span the complete life-cycle.

Environments: An environment is a collection of CASE tools or workbenches that attempts to support the complete software process. This contrasts with tools that focus on one specific task or a specific part of the life-cycle.

CASE environments are classified as follows:

- **Toolkits.** Loosely coupled collections of tools. These typically build on operating system workbenches such as the Unix Programmer's Workbench or the VMS VAX set. They typically perform integration via piping or some other basic mechanism to share data and pass control. The strength of easy integration is also one of the drawbacks. Simple passing of parameters via technologies such as shell scripting can't provide the kind of sophisticated integration that a common repository database can.
- **Fourth generation.** These environments are also known as 4GL standing for fourth generation language environments due to the fact that the early environments were designed around specific languages such as Visual Basic. They were the first environments to provide deep integration of multiple tools. Typically these environments were focused on specific types of applications. For example, user-interface driven applications that did standard atomic transactions to a relational database. Examples are Informix 4GL, and Focus.
- **Language-centred.** Environments based on a single often object-oriented language such as the Symbolic Lisp Genera environment or Visual Works Smalltalk from Parcplace. In these environments all the operating system resources were objects in the object-oriented language. This provides powerful debugging and graphical opportunities but the code developed is mostly limited to the specific language. For this reason, these environments were mostly a niche within CASE. Their use was mostly for prototyping and R&D projects. A common core idea for these environments was the model-view-controller user interface that facilitated keeping multiple presentations of the same design consistent with the underlying model. The MVC architecture was adopted by the other types of CASE environments as well as many of the applications that were built with them.
- **Integrated.** These environments are an example of what most IT people tend to think of first when they think of CASE. Environments such as IBM's AD/Cycle, Andersen Consulting's FOUNDATION, the ICL CADES system, and DEC Cohesion. These environments attempt to cover the complete life-cycle from analysis to maintenance and provide an integrated database repository for storing all artefacts of the software process. The integrated software repository was the defining feature for these kinds of tools. They provided multiple different design models as well as support for code in heterogeneous languages. One of the main goals for these types of environments was "round trip engineering": being able to make changes at the design level and have

those automatically be reflected in the code and vice versa. These environments were also typically associated with a particular methodology for software development.

- ***Process-centred.*** This is the most ambitious type of integration. These environments attempt to not just formally specify the analysis and design objects of the software process but the actual process itself and to use that formal process to control and guide software projects. Examples are East, Enterprise II, Process Wise, Process Weaver, and Arcadia. These environments were by definition tied to some methodology since the software process itself is part of the environment and can control many aspects of tool invocation.

In practice, the distinction between workbenches and environments was flexible. Visual Basic for example was a programming workbench but was also considered a 4GL environment by many. The features that distinguished workbenches from environments were deep integration via a shared repository or common language and some kind of methodology (integrated and process-centred environments) or domain (4GL) specificity.

Components of CASE Tools

CASE tools can be broadly divided into the following parts based on their use at a particular SDLC stage as depicted in figure 5.1:

- ***Central Repository*** - CASE tools require a central repository, which can serve as a source of common, integrated and consistent information. Central repository is a central place of storage where product specifications, requirement documents, related reports and diagrams, other useful information regarding management are stored. Central repository also serves as data dictionary.
- ***Upper Case Tools*** - Upper CASE tools are used in planning, analysis and design stages of SDLC.
- ***Lower Case Tools*** - Lower CASE tools are used in implementation, testing and maintenance.
- ***Integrated Case Tools*** - Integrated CASE tools are helpful in all the stages of SDLC, from Requirement gathering to Testing and documentation.

CASE tools can be grouped together if they have similar functionality, process activities and capability of getting integrated with other tools.

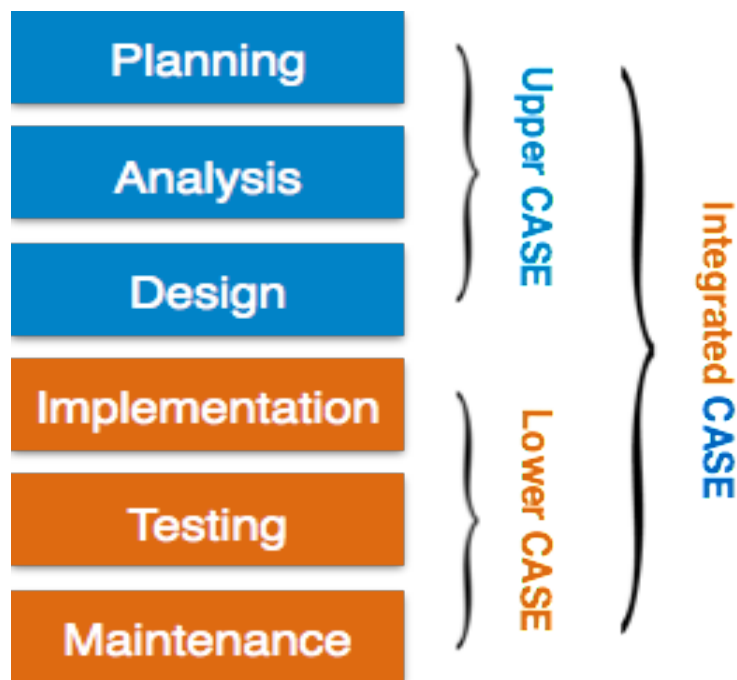


Figure 5.1: CASE Tool components

Check Your Progress 1.

How CASE tools are useful?

5.3 BUILDING BLOCKS OF CASE

Computer aided software engineering can be as simple as a single tool that supports a specific software engineering activity or as complex as a complete "environment" that encompasses tools, a database, people, hardware, a network, operating systems, standards, and myriad other components. The building blocks for CASE are illustrated in figure. Each building block forms a foundation for the next, with tools sitting at the top of the heap. It is interesting to note that the foundation for effective CASE environments has relatively little to do with software engineering tools themselves. Rather, successful environments for software engineering are built on an environment architecture that encompasses appropriate hardware and systems software. In addition, the environment architecture must consider the human work patterns that are applied during the software engineering process.

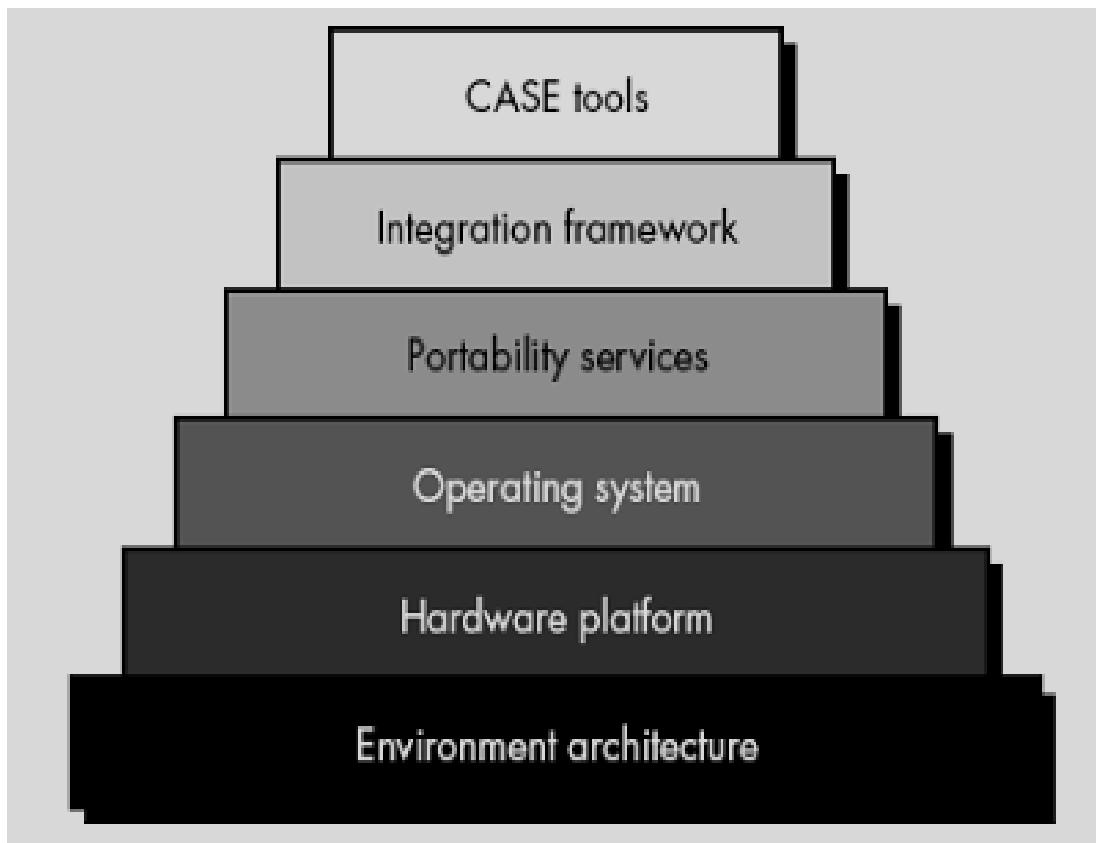


Figure 5.2: Building blocks of CASE

The building blocks depicted in figure 5.2 represent a comprehensive foundation for the integration of CASE tools. However, most CASE tools in use today have not been constructed using all these building blocks. In fact, some CASE tools remain "point solutions." That is, a tool is used to assist in a particular software engineering activity (e.g., analysis modelling) but does not directly communicate with other tools, is not tied into a project database, is not part of an integrated CASE environment (ICASE). Although this situation is not ideal, a CASE tool can be used quite effectively, even if it is a point solution.

Here are the Building blocks of CASE:

- **Environment Architecture.** The environment architecture, composed of the hardware platform and operating system support including networking and database management software, lays the groundwork for CASE but the CASE environment itself demands other building blocks.
- **Portability Services.** A set of portability services provides a bridge between CASE tools and their integration framework and the environment architecture. These portability services allow the CASE tools and their integration framework to migrate

across different hardware platforms and operating systems without significant adaptive maintenance.

■ **Integration Framework.** It is a collection of specialized programs that enables individual CASE tools to communicate with one another and to create a project database.

■ **Case Tools.** Case tools are used to assist software-engineering activities (such as analysis modelling, code generation, etc.) by either communicating with other tools, the project database (integrated CASE environment), or as point solutions.

■ Operating system

■ Hardware platform

There are relative levels of CASE integration as depicted in figure 5.3.

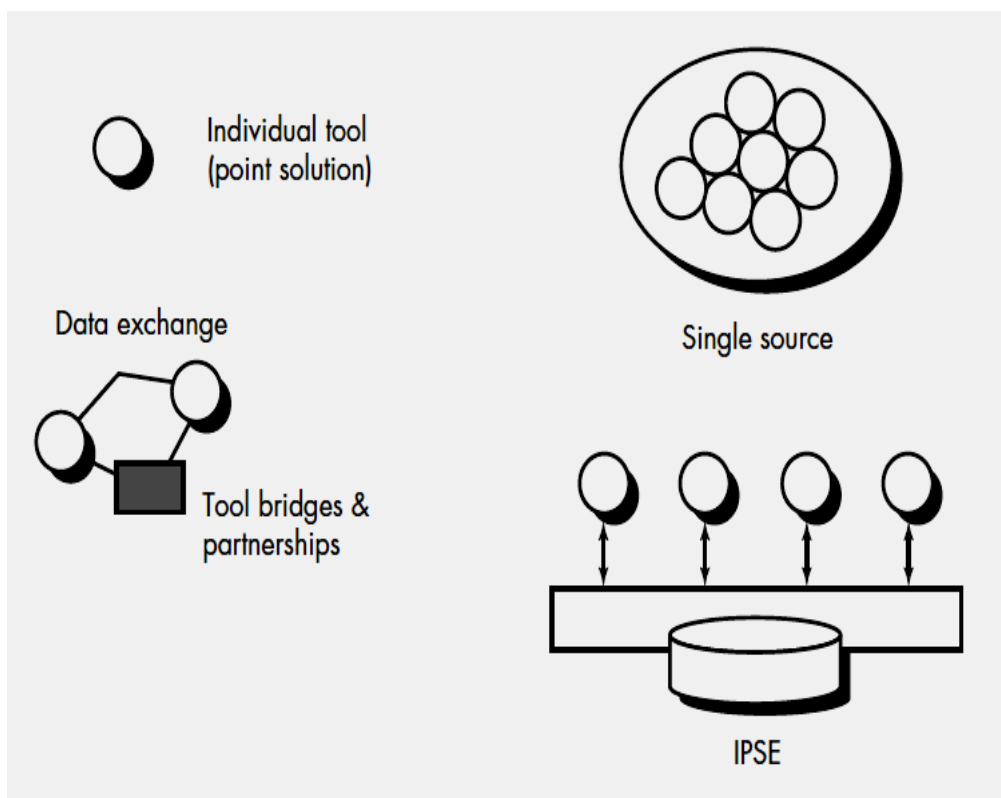


Figure 5.3: Integration options

At the low end of the integration spectrum is the individual tool. When individual tools provide facilities for data exchange, the integration level is improved slightly. Such tools produce output in a standard format that should be compatible with other tools that can read the format. In some cases, the builders of complementary CASE tools work together to form a bridge between the tools. Using this approach, the synergy between the tools can produce end products that would be difficult to create using either tool separately. Single-source

integration occurs when a single CASE tools vendor integrates a number of different tools and sells them as a package. Although this approach is quite effective, the closed architecture of most single-source environments precludes easy addition of tools from other vendors.

At the high end of the integration spectrum is the integrated project support environment (IPSE). Standards for each of the building blocks described previously have been created. CASE tool vendors use IPSE standards to build tools that will be compatible with the IPSE and therefore compatible with one another.

Check Your Progress 2.

How portability services works?

5.4 A TAXONOMY OF CASE TOOLS

A number of risks are inherent whenever we attempt to categorize CASE tools. There is a subtle implication that to create an effective CASE environment, one must implement all categories of tools—this is simply not true. Confusion can be created by placing a specific tool within one category when others might believe it belongs in another category. In addition, simple categorization tends to be flat—that is, we do not show the hierarchical interaction of tools or the relationships among them. But even with these risks, it is necessary to create taxonomy of CASE tools—to better understand the breadth of CASE and to better appreciate where such tools can be applied in the software engineering process.

CASE tools can be classified by function, by their role as instruments for managers or technical people, by their use in the various steps of the software engineering process, by the environment architecture (hardware and software) that supports them, or even by their origin or cost.

- ***Business process engineering tools-*** By modeling the strategic information requirements of an organization, business process engineering tools provide a "meta-model" from which specific information systems are derived. Rather than focusing on the requirements of a specific application, business information is modeled as it moves between various organizational entities within a company. The primary objective for tools in this category is to represent business data objects, their relationships, and how these data objects flow between different business areas within a company.

- ***Process modeling and management tools-*** If an organization works to improve a business (or software) process, it must first understand it. Process modeling tools (also called *process technology* tools) are used to represent the key elements of a process so that it can be better understood. Such tools can also provide links to process descriptions that help those involved in the process to understand the work tasks that are required to perform it. Process management tools provide links to other tools that provide support to defined process activities.
- ***Project planning tools-*** Tools in this category focus on two primary areas: software project effort and cost estimation and project scheduling. Estimation tools compute estimated effort, project duration, and recommended number of people for a project. Project scheduling tools enable the manager to define all project tasks (the work breakdown structure), create a task network (usually using graphical input), represent task interdependencies, and model the amount of parallelism possible for the project.
- ***Risk analysis tools-*** Identifying potential risks and developing a plan to mitigate, monitor, and manage them is of paramount importance in large projects. Risk analysis tools enable a project manager to build a risk table by providing detailed guidance in the identification and analysis of risks.
- ***Project management tools-*** The project schedule and project plan must be tracked and monitored on a continuing basis. In addition, a manager should use tools to collect metrics that will ultimately provide an indication of software product quality. Tools in the category are often extensions to project planning tools.
- ***Requirements tracing tools-*** When large systems are developed, things "fall into the cracks." That is, the delivered system does not fully meet customer specified requirements. The objective of requirements tracing tools is to provide a systematic approach to the isolation of requirements, beginning with the customer request for proposal or specification. The typical requirements tracing tool combines human interactive text evaluation with a database management system that stores and categorizes each system requirement that is "parsed" from the original RFP or specification.
- ***Metrics and management tools-*** Software metrics improve a manager's ability to control and coordinate the software engineering process and a practitioner's ability to improve the quality of the software that is produced. Today's metrics or measurement tools focus on process and product characteristics. Management-oriented tools capture project specific metrics that provide an overall indication of productivity or quality.

Technically oriented tools determine technical metrics that provide greater insight into the quality of design or code.

- **Documentation tools-** Document production and desktop publishing tools support nearly every aspect of software engineering and represent a substantial "leverage" opportunity for all software developers. Most software development organizations spend a substantial amount of time developing documents, and in many cases the documentation process itself is quite inefficient. It is not unusual for a software development organization to spend as much as 20 or 30 percent of all software development effort on documentation. For this reason, documentation tools provide an important opportunity to improve productivity.
- **System software tools-** CASE is a workstation technology. Therefore, the CASE environment must accommodate high-quality network system software, object management services, distributed component support, electronic mail, bulletin boards, and other communication capabilities.
- **Quality assurance tools-** The majority of CASE tools that claim to focus on quality assurance are actually metrics tools that audit source code to determine compliance with language standards. Other tools extract technical metrics in an effort to project the quality of the software that is being built.
- **Database management tools-** Database management software serves as a foundation for the establishment of a CASE database (repository) that we have called the *project database*. Given the emphasis on configuration objects, database management tools for CASE are evolving from relational database management systems to object oriented database management systems.
- **Software configuration management tools-** Software configuration management lies at the kernel of every CASE environment. Tools can assist in all five major SCM tasks—identification, version control, change control, auditing, and status accounting. The CASE database provides a mechanism for identifying each configuration item and relating it to other items; the change control process can be implemented with the aid of specialized tools; easy access to individual configuration items facilitates the auditing process; and CASE communication tools can greatly improve status accounting.
- **Analysis and design tools-** Analysis and design tools enable a software engineer to create models of the system to be built. The models contain a representation of data, function, and behavior and characterizations of data, architectural, component-level,

and interface design. One By performing consistency and validity checking on the models, analysis and design tools provide a software engineer with some degree of insight into the analysis representation and help to eliminate errors before they propagate into the design, or worse, into implementation itself.

- ***PRO/SIM tools-*** PRO/SIM (prototyping and simulation) tools provide the software engineer with the ability to predict the behavior of a real-time system prior to the time that it is built. In addition, these tools enable the software engineer to develop mock-ups of the real-time system, allowing the customer to gain insight into the function, operation and response prior to actual implementation.
- ***Interface design and development tools-*** Interface design and development tools are actually a tool kit of software components (classes) such as menus, buttons, window structures, icons, scrolling mechanisms, device drivers, and so forth. However, these tool kits are being replaced by interface prototyping tools that enable rapid onscreen creation of sophisticated user interfaces that conform to the interfacing standard that has been adopted for the software.
- ***Prototyping tools-*** A variety of different prototyping tools can be used. *Screen painters* enable a software engineer to define screen layout rapidly for interactive applications. More sophisticated CASE prototyping tools enable the creation of a data design, coupled with both screen and report layouts. Many analysis and design tools have extensions that provide a prototyping option. PRO/SIM tools generate skeleton Ada and C source code for engineering (real-time) applications. Finally, a variety of fourth generation tools have prototyping features.
- ***Programming tools-*** The programming tools category encompasses the compilers, editors, and debuggers that are available to support most conventional programming languages. In addition, object-oriented programming environments, fourth generation languages, graphical programming environments, application generators, and database query languages also reside within this category.
- ***Web development tools*** - The activities associated with Web engineering are supported by a variety of tools for Web App development. These include tools that assist in the generation of text, graphics, forms, scripts, applets, and other elements of a Web page.
- ***Integration and testing tools-*** In their directory of software testing tools, Software Quality Engineering defines the following testing tools categories:
 - *Data acquisition*—tools that acquire data to be used during testing.

- *Static measurement*—tools that analyze source code without executing test cases.
 - *Dynamic measurement*—tools that analyze source code during execution.
 - *Simulation*—tools that simulate function of hardware or other externals.
 - *Test management*—tools that assist in the planning, development, and control of testing.
 - *Cross-functional tools*—tools that cross the bounds of the preceding categories.
 - It should be noted that many testing tools have features that span two or more of the categories.
- ***Static analysis tools-*** Static testing tools assist the software engineer in deriving test cases. Three different types of static testing tools are used in the industry: code based testing tools, specialized testing languages, and requirements-based testing tools. *Code-based testing tools* accept source code (or PDL) as input and perform a number of analyses that result in the generation of test cases. *Specialized testing languages* (e.g., ATLAS) enable a software engineer to write detailed test specifications that describe each test case and the logistics for its execution. *Requirements-based testing tools* isolate specific user requirements and suggest test cases (or classes of tests) that will exercise the requirements.
 - ***Dynamic analysis tools-*** Dynamic testing tools interact with an executing program, checking path coverage, testing assertions about the value of specific variables, and otherwise instrumenting the execution flow of the program. Dynamic tools can be either intrusive or nonintrusive. An *intrusive tool* changes the software to be tested by inserting probes (extra instructions) that perform the activities just mentioned. *Nonintrusive testing tools* use a separate hardware processor that runs in parallel with the processor containing the program that is being tested.
 - ***Test management tools-*** Test management tools are used to control and coordinate software testing for each of the major testing steps. Tools in this category manage and coordinate regression testing, perform comparisons that ascertain differences between actual and expected output, and conduct batch testing of programs with interactive human/computer interfaces. In addition to the functions noted, many test management tools also serve as generic test drivers. A test driver reads one or more test cases from a testing file, formats the test data to conform to the needs of the software under test, and then invokes the software to be tested.

- **Client/server testing tools-** The c/s environment demands specialized testing tools that exercise the graphical user interface and the network communications requirements for client and server.
- **Reengineering tools-** Tools for legacy software address a set of maintenance activities that currently absorb a significant percentage of all software-related effort.

These tools are limited to specific programming languages and require some degree of interaction with the software engineer.

Check Your Progress 3.

Name some CASE tools useful during Testing.

5.5 INTEGRATED CASE ENVIRONMENTS

Although benefits can be derived from individual CASE tools that address separate software engineering activities, the real power of CASE can be achieved only through integration. The benefits of integrated CASE (I-CASE) include

- Smooth transfer of information (models, programs, documents, data) from one tool to another and one software engineering step to the next;
- A reduction in the effort required to perform umbrella activities such as software configuration management, quality assurance, and document production;
- An increase in project control that is achieved through better planning, monitoring, and communication; and
- Improved coordination among staff members who are working on a large software Project.

But I-CASE also poses significant challenges. Integration demands consistent representations of software engineering information, standardized interfaces between tools, a homogeneous mechanism for communication between the software engineer and each tool, and an effective approach that will enable I-CASE to move among various hardware platforms and operating systems. Comprehensive I-CASE environments have emerged more slowly than originally expected. However, integrated environments do exist and are becoming more powerful as the years pass.

The term integration implies both combination and closure. I-CASE combines a variety of different tools and a spectrum of information in a way that enables closure of communication among tools, between people, and across the software process. Tools are integrated so that

software engineering information is available to each tool that needs it; usage is integrated so that a common look and feel is provided for all tools; a development philosophy is integrated, implying a standardized software engineering approach that applies modern practice and proven methods.

To define integration in the context of the software engineering process, it is necessary to establish a set of requirements for I-CASE: An integrated CASE environment should

- Provide a mechanism for sharing software engineering information among all tools contained in the environment.
- Enable a change to one item of information to be tracked to other related information items.
- Provide version control and overall configuration management for all software engineering information.
- Allow direct, non-sequential access to any tool contained in the environment.
- Establish automated support for the software process model that has been chosen, integrating CASE tools and software configuration items (SCIs) into a standard work breakdown structure.
- Enable the users of each tool to experience a consistent look and feel at the human/computer interface.
- Support communication among software engineers.
- Collect both management and technical metrics that can be used to improve the process and the product.

To achieve these requirements, each of the building blocks of a CASE architecture must fit together in a seamless fashion. The foundation building blocks—environment architecture, hardware platform, and operating system—must be "joined" through a set of portability services to an integration framework that achieves these requirements.

A CASE environment facilitates the automation of the step-by-step methodologies for software development. In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of software development

Check Your Progress 4.

Describe the importance of an integrated environment.

5.6 THE INTEGRATION ARCHITECTURE

A software engineering team uses CASE tools, corresponding methods, and a process framework to create a pool of software engineering information. The integration framework facilitates transfer of information into and out of the pool. To accomplish this, the following architectural components must exist: a database must be created; an object management system must be built; a tools control mechanism must be constructed; a user interface must provide a consistent pathway between actions made by the user and the tools contained in the environment. Most models of the integration framework represent these components as layers as depicted in figure 5.4.

The user interface layer incorporates a standardized interface tool kit with a common presentation protocol. The interface tool kit contains software for human/computer interface management and a library of display objects. Both provide consistent mechanisms for communication between the interface and individual CASE tools. The presentation protocol is the set of guidelines that gives all CASE tools the same look and feel. Screen layout conventions, menu names and organization, icons, object names, the use of the keyboard and mouse, and the mechanism for tools access are all defined as part of the presentation protocol.

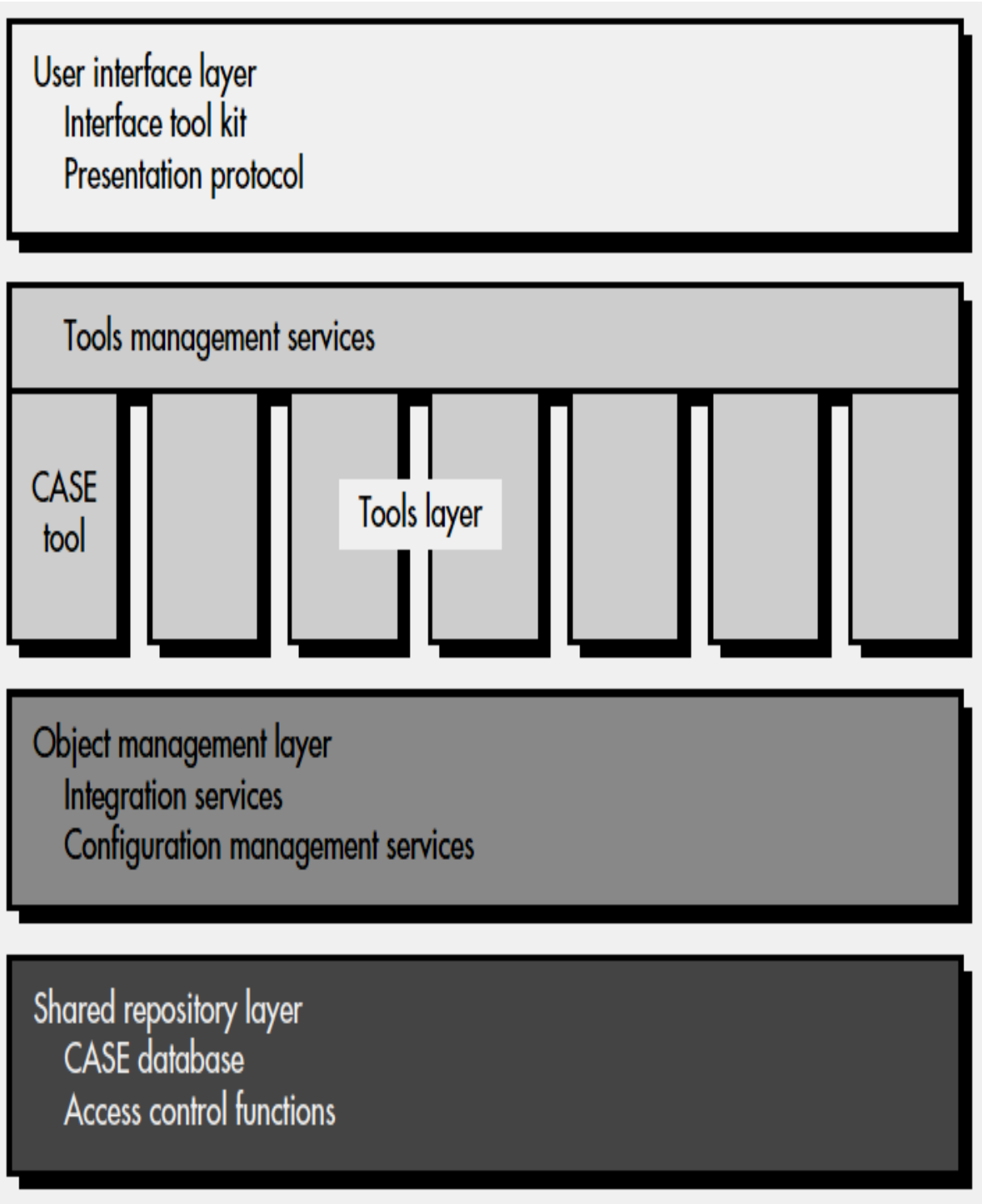


Figure 5.4: Architectural model for the integration framework

The tools layer incorporates a set of tools management services with the CASE tools themselves. Tools management services (TMS) control the behaviour of tools with in the environment. If multitasking is used during the execution of one or more tools, TMS performs multitask synchronization and communication, coordinates the flow of information

from the repository and object management system into the tools, accomplishes security and auditing functions, and collects metrics on tool usage.

In essence, software in this layer of the framework architecture provides the mechanism for tools integration. Every CASE tool is "plugged in to" the object management layer. Working in conjunction with the CASE repository, the OML provides integration services—a set of standard modules that couple tools with the repository. In addition, the OML provides configuration management services by enabling the identification of all configuration objects, performing version control, and providing support for change control, audits, and status accounting. The shared repository layer is the CASE database and the access control functions that enable the object management layer to interact with the database.

Check Your Progress 5.

How user interface layer works?

5.7 THE CASE REPOSITORY

Webster's Dictionary defines the word repository as "anything or person thought of as a centre of accumulation or storage." During the early history of software development, the repository was indeed a person—the programmer who had to remember the location of all information relevant to software project, who had to recall information that was never written down and reconstruct information that had been lost. Sadly, using a person as "the centre for accumulation and storage", does not work very well. Today, the repository is a "thing"-a database that acts as the centre for both accumulation and storage of software engineering information. The role of the software engineer is to interact with the repository using CASE tools that are integrated with it.

The Role of the Repository in CASE

The repository for a CASE environment is the set of mechanisms and data structures that achieve data/tool and data/data integration. It provides the obvious functions of a database management system, but in addition, the repository performs or precipitates the following functions:

- Data integrity includes functions to validate entries to the repository, ensure consistency am

- ong related objects, and automatically perform "cascading" modifications when a change to one object demands some change to objects related to it.
- Information sharing provides a mechanism for sharing information among multiple developers and between multiple tools, manages and controls multiuser access to data and locks or unlocks objects so that changes are not inadvertently overlaid on one another.
- Data/tool integration establishes a data model that can be accessed by all tools in the I-CASE environment, controls access to the data, and performs appropriate configuration management functions.
- Data/data integration is the database management system that relates data objects so that other functions can be achieved.
- Methodology enforcement defines an entity-relationship model stored in the repository that implies a specific paradigm for software engineering; at a minimum, the relationships and objects define a set of steps that must be conducted to build the contents of the repository.
- Document standardization is the definition of objects in the database that leads directly to a standard approach for the creation of software engineering documents.

Types of Things to be stored:

The types of things to be stored in the repository include:

- The problem to be solved.
- Information about the problem domain.
- The system solution as it emerges.
- Rules and instructions pertaining to the software process (methodology) being followed.
- The project plan, resources, and history.
- Information about the organizational context.

Case Repository Contents:

- ***Enterprise information***
 - Organizational structure
 - Business area analyses System
 - Business functions
 - Business rules

- Process models (scenarios)
- Information architecture
- ***Application design***
 - Methodology rules
 - Graphical representations
 - System diagrams
 - Naming standards
 - Referential integrity rules
 - Data structures
 - Process definitions
 - Class definitions
 - Menu trees Estimates;
 - Performance criteria
 - Timing constraints
 - Screen definitions
 - Report definitions
 - Logic definitions
 - Behavioural logic
 - Algorithms
 - Transformation rules
- ***Construction***
 - Source code; Object code
 - build instructions
 - Binary images
 - Configuration dependencies
 - Change information
- ***Validation and verification***
 - Test plan; Test data cases
 - Regression test scripts
 - Test results
 - Statistical analyses
 - Software quality metrics
- ***Project management information***
 - Project plans

- Work breakdown structure
- Schedules
- Resource loading; Problem reports
- Change requests; Status reports
- Audit information
- ***System documentation***
 - Requirements documents
 - External/internal designs
 - User manuals

The DBMS features in CASE:

- Non-redundant data storage
- High-level access
- Data independence
- Transaction control
- Security
- Ad hoc data queries and reports
- Openness
- Multiuser support

The special features of CASE:

- Storage of sophisticated data structures.
- Integrity enforcement.
- Semantics-rich tool interface.
- Process/project management.

Check Your Progress 6.

What are the functions performed by repository in an integrated CASE environment.

5.8 SUMMARY

In this chapter we discuss about the CASE tools, uses and application area of CASE tools, CASE environment, CASE Repository. All aspects of the software development life cycle can be supported by software tools, and so the use of tools from across the spectrum can,

arguably, be described as CASE; from project management software through tools for business and functional analysis, system design, code storage, compilers, translation tools, test software, and so on.

5.9 EXERCISE

- 1) What is CASE and CASE tools? Why we use CASE tool.
- 2) What is CASE environment? Differentiate CASE environment and a programming environment.
- 3) Explain the Benefits of CASE.
- 4) Write short notes on CASE Repository.
- 5) How the CASE tools are classified.
- 6) What are the advantages and drawbacks of CASE?