BCA-08

Vardhaman Mahaveer Open University, Kota

Software Engineering

Course Development Committee

Chairman

Prof. (Dr.) Naresh Dadhich
Vice-Chancellor
Vardhaman Mahaveer Open University, Kota

Co-ordinator/Convener and Members

Convener

Dr. Anuradha Sharma
Assistant Professor,
Department of Botany, Vardhaman Mahaveer Open University, Kota

Members :

1. **Prof. (Dr.) D.S.Chauhan**
   Department of Mathematics
   University of Rajasthan, Jaipur
2. **Prof. (Dr.) M.C.Govil**
   Govt. Mahila Engineering College, Ajmer
3. **Prof. (Dr.) A.K.Nagawat**
   University of Rajasthan, Jaipur
4. **Dr. (Mrs.) Madhavi Sinha**
   Birla Institute of Technology & Science, Jaipur
5. **Dr. Rajeev Srivastava**
   LBS College, Jaipur

Editing and Course Writing

Editor

Dr. (Mrs.) Reena Dadhich
Associate Professor
Head, Department of Master of Computer Applications
Govt. Engineering College, Ajmer

Unit Writers	Unit No.	Unit Writers	Unit No.
1. **Dr. Anubha Jain** Department of Computer Science ICG College, Jaipur	(1, 2)	5. **Sh. Sanjeev Agnihotri** Department of Computer Science Modi Institute of Technology, Kota	(9, 10)
2. **Sh. Rajeev Srivastava** Department of Computer Science LBS PG College, Jaipur	(3, 4)	6. **Sh. Bharat Gupta** Department of Computer Science RN Modi Engineering College, Kota	(11, 12)
3. **Sh. Bright Keswani** Department of Computer Science S. Gyan Vihar University, Jaipur	(5, 6)	7. **Dr. O.P. Rishi** Department of Computer Science Central University of Rajasthan, Jaipur	(13, 14)
4. **Mrs. Poonam Keswani** Department of Computer Science Senior Software Consultant, Mapple Technology, Jaipur	(7, 8)		

Academic and Administrative Management

Prof. (Dr.) Naresh Dadhich Vice-Chancellor Vardhaman Mahveer Open University, Kota	**Prof. B.K. Sharma** Director (Academic) Vardhaman Mahveer Open University, Kota	**Mr. Yogendra Goyal** Incharge Material Production and Distribution Department

Course Material Production

Mr. Yogendra Goyal
Assistant Production Officer
Vardhaman Mahaveer Open University, Kota

PREFACE

The major component of software development process is a **“Software Engineering ”,** which carries the most important role in software development life cycle. It is necessary for a computer science student to have knowledge about these principles and practices to be a skilled software den elopes. This book is an edited collection of different phases of software development life cycle. This book includes all formal steps of software development life cycle such as Software Project Management, Planning & Estimation, Software Requirements Analysis and Software Design as well as Software Testing Techniques and Quality Assurance. It also introduces about the process of Software Re-engineering and concept of computerised Software Engineering. The book covers all course contents. Each chapter is designed to cover individual unit.

The book has been compiled for the beginners with a coherent view of the state of the art and practice and provides developers and managers with technical and organizational approaches to push the state of the art.

Vardhaman Mahaveer Open University, Kota

Software Engineering

Unit - 1 : Introduction to Software Engineering

Structure of the Unit

1.0 Objectives

This chapter provides a general overview of

- Software Engineering Emergence
- Aims of Software Engineering
- Software Characteristics, Crisis and Myths
- Software Qualities
- Principles of Software Engineering

1.1 Introduction

The nature and complexity of software have changed significantly in the last 30 years. In the 1970s, applications ran on a single processor, produced alphanumeric output, and received their input from a linear source. Today's applications are far more complex; typically have graphical user interface and client-server architecture. They frequently run on two or more processors, under different operating systems, and on geographically distributed machines.

Rarely, in history has a field of endeavor evolved as rapidly as software development. The struggle to stay, abreast of new technology, deal with accumulated development backlogs, and cope with people issues has become a treadmill race, as software groups work as hard as they can, just to stay in place. The initial concept of one "guru", indispensable to a project and hostage to its continued maintenance has changed. The Software Engineering Institute (SEI) and group of "gurus" advise us

to improve our development process. Improvement means "ready to change". Not every member of an organization feels the need to change. It is too easy to dismiss process improvement efforts as just the latest management fad.

Computers are fast becoming our way of life and one cannot imagine life without computers in today's world. You go to a railway station for reservation, you want to explore a web site to book a ticket for a cinema, you go to a library, or you go to a bank, you will find computers at all places. Since computers are used in every possible field today, it becomes an important issue to understand and build these computerized systems in an effective way. Building such systems is not an easy process but requires certain skills and capabilities to understand and follow a systematic procedure towards making of any information system. For this, experts in the field have devised various methodologies. Waterfall model is one of the oldest methodologies. Later Prototype Model, Object Oriented Model, Dynamic Systems Development Model, and many other models became very popular for system development.

Therefore, there is an urgent need to adopt software engineering concepts, strategies, practices to avoid conflict, and to improve the software development process in order to deliver good quality maintainable software in time and within budget.

1.2 The Evolving Role of Software

The software has seen many changes since its inception. After all, it has evolved over the period of time against all odds and adverse circumstances. Computer industry has also progressed at a break-neck speed through the computer revolution, and recently, the network revolution triggered and/or accelerated by the explosive spread of the internet and most recently the web. Computer industry has been delivering exponential improvement in price performance, but the problems with software have not been decreasing. Software still come late, exceed budget and are full of residual faults. As per the latest IBM report, "31% of the projects get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% and for every 100 projects, there are 94 restarts" .

1.3 What is Software Engineering?

Software is the set of instructions encompasses programs that execute within a computer of any size and architecture, documents that encompass hard-copy and virtual forms, and data that combine numbers and text. It also includes representations of pictorial, video, and audio information. Software engineers can build the software and virtually everyone in the industrialized world uses it either directly or indirectly. It is so important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities.

The steps to build the computer software is as the user would like to build any successful product, by applying a process that leads to a high-quality result that meets the needs of the people who will use the product. From the software engineer's view, the product is may be the programs, documents, and data that are computer software. But from the user's viewpoint, the product is the resultant information that somehow makes the user's world better. Software's impact on the society and culture continues to be profound. As its importance grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build high-quality computer programs. Some of these technologies are targeted at a specific application domain like web-site design and implementation; others focus on a technology domain such as object oriented systems and still others are broad-based like operating systems such as LINUX.

However, a software technology has to develop useful information. The technology encompasses a process, a set of methods, and an array of tools called as software engineering.

Software Engineering is the systematic approach to the development, operation and maintenance of

software. Software Engineering is concerned with development and maintenance of software products. Software has become critical to advancement in almost all areas of human endeavor. The art of programming only is no longer sufficient to construct large programs. There are serious problems in the cost, timeliness, maintenance and quality of many software products.

The primary goal of software engineering is to provide the quality of software with low cost. Software Engineering involves project planning, project management, systematic analysis, design, validations and maintenance activities.

Software engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget. To achieve this objective, we have to focus in a disciplined manner on both the quality of the product and on the process used to develop the product.

1.3.1 Definition

At the first conference on software engineering in 1968, Fritz Bauer defined software engineering as "The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines". Stephen Schach defined the same as "A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements".

Both the definitions are popular and acceptable to majority. However, due to increase in cost of maintaining software, objective is now shifting to produce quality software that is maintainable, delivered on time, within budget, and also satisfies its requirements.

A popular definition will be : development of procedures and systematic applications that are used on electronic machines. Software engineering incorporates various accepted methodologies to design software. This particular type of engineering has to take into consideration what type of machine the software will be used on, how the software will work with the machine, and what elements need to be put in place to ensure reliability.

1.4 Software Characteristics

To make the difference from other things or product, it is important to examine the characteristics of software. When hardware is built, the human creative process may be analysis, design, construction, testing is ultimately translated into a physical form where as build a new computer, the initial sketches, formal design drawings, and bread boarded prototype evolve into a physical product such as chips, circuit boards, power supplies, etc.

Software is a logical related rather than a physical system. So that the software have distinct characteristics but the hardware is not so, it is only the peripherals or devices or components.

Character 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 the activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent or easily corrected for software. Both the activities are dependent on people, but the relationship between people is totally varying. These two activities require the construction of a "product" but the approaches are different. Software costs are concentrated in engineering which means that software projects cannot be managed as if they were manufacturing.

Character 2: Software does not wear out

The figure 1 shows the relationship between failure rate and time. Consider the failure rate as a function of time for hardware. The relationship is called the bathtub curve, 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. So, stated simply, the hardware begins to wear out.

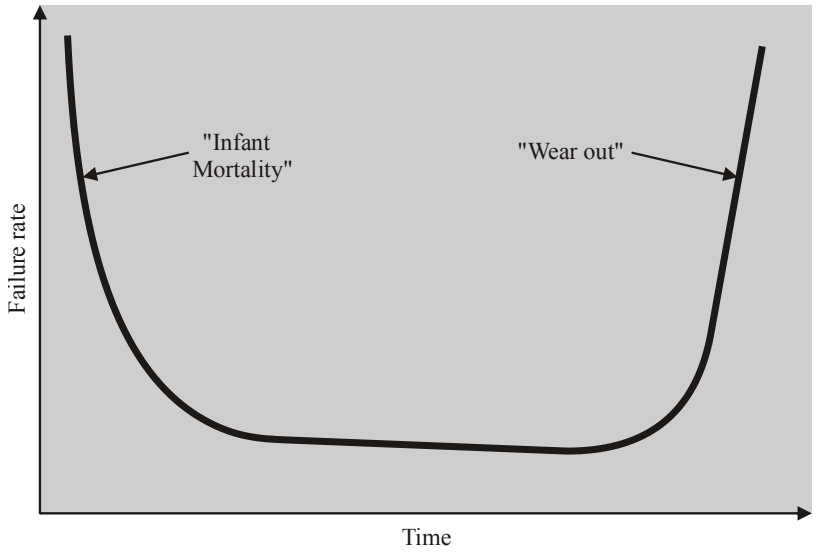


Fig 1: Relationship between failure rate and time

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" like a zig-zag form. Undiscovered defects will cause high failure rates early in the life of a program. However, the implication is clear software doesn't wear out. But it does deteriorate. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

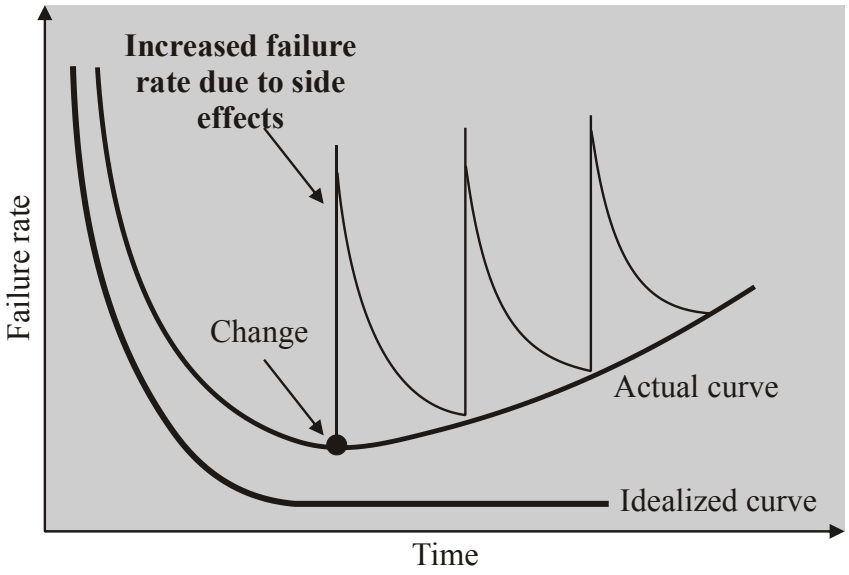


Fig 2: Idealized and actual failure curves for software

When a hardware component wears out, it is replaced by a spare part unlike the software spare parts. The software failure indicates an error in design or in the process through which design as translated into machine executable code. Therefore, software maintenance involves more complexity than hardware maintenance.

Character 3 : Although the industry is moving toward component-based assembly, most software continues to be custom built

Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of

digital components exist. Each integrated circuit (called an *IC* or a *chip*) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines.

After each component is selected, it can be ordered off the shelf. As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are standard components that are used by mechanical and electrical engineers to design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process.

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. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application and not extended algorithm only but included data structure too. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts.

1.5 Software Crisis

Many industry observers have characterized the problems associated with software development as a crisis. The experts have recounted the impact of some of the more spectacular software failures that have occurred over the past decade. Yet, the great successes achieved by the software industry have led many to question whether the term software crisis is still appropriate. Software crisis was a term used in the early days of computing science. The term was used to describe the impact of rapid increases in computer power and the complexity of the problems that could be tackled. In essence, it refers to the difficulty of writing correct, understandable, and verifiable computer programs. The roots of the software crisis are complexity, expectations, and change.

Robert Glass states that the failure stories and exception reporting, spectacular failures in the midst of many successes. From the Webster's Dictionary, crisis is defined as a turning point in the course of anything; decisive or crucial time, stage or event. In terms of overall software quality and the speed with which computer-based systems and products are developed, there has been no "turning point," no "decisive time," only slow, evolutionary change, punctuated by explosive technological changes in disciplines associated with software.

Affliction is defined as anything causing pain or distress. It is far more accurate to describe the problems we have endured in the software business as a chronic affliction than a crisis. Regardless of what we call it, the set of problems that are encountered in the development of computer software is not limited to software that doesn't function properly. Rather, the affliction encompasses problems associated with how we develop software, how we support a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software.

By the end of the 1960s, hardware costs had fallen exponentially, and were continuing to do so, while the cost of software development was rising at a similar rate. The apparent problem of incomplete, poorly performing software became referred to as "the software crisis". Until this time, the attitude of many of the people involved in the computer industry had been to treat computers as almost a craft industry.

"Programming started out as a craft which was practiced intuitively. By 1968, it began to be generally acknowledged that the methods of program development followed so far were inadequate... Hence, a style of design was developed in which the program and its correctness ... were designed hand in hand. This was a dramatic step forward."

With the emergence of large and powerful general purpose mainframe computers (such as the IBM 360) large and complex systems became possible. People began to ask about project failures, cost overruns and so on and, in the late 1960s a new discipline of Software Engineering was born. Now, for the first time, the creation of software was treated as a discipline in its own right, demanding as

much reasoning and consideration as the hardware on which it was to run.

As programs and computers became part of the business world, so their development moved out of the world of bespoke craft work and became a commercial venture; the buyers of software increasingly demanded a product that was built to a high quality, on time and within budget. Many large systems of that time were seen as absolute failures - either they were abandoned, or did not deliver any of the anticipated benefits.

Frequently, software was never completed, even after further significant investment had been made. The amount of work involved in removing flaws and bugs from "completed" software, to make it usable, often took a considerable amount of time - often more than what had been spent in the writing it in the first place. The functionality of the software seldom matched the requirements of the end-users. Once created, software was almost impossible to maintain; the developer's ability to understand what they had written appeared to decrease rapidly over time.

The causes of the software crisis were linked to the overall complexity of hardware and the software development process. The crisis manifested itself in several ways:

- Projects running over-budget.
- Projects running over-time.
- Software was very inefficient.
- Software was of low quality.
- Software often did not meet requirements.
- Projects were unmanageable and code difficult to maintain.
- Software was never delivered.

Many of the software problems were caused by increasingly complex hardware. In his essay, Dijkstra noted that the newer computers in his day "embodied such serious flaws that [he] felt that with a single stroke the progress of computing science had been retarded by at least ten years". He also believed that the influence of hardware on software was too frequently overlooked.

1.6 Software Myths

Many causes of a software affliction can be traced to a mythology during the development of software. Software myths propagated misinformation and confusion. Software myths had a number of attributes that made them insidious. Today, most knowledgeable professionals recognize myths for what they are misleading attitudes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify, and remnants of software myths are still believed.

1.6.1 Management Myths

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure.

Myth : We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?

Reality : The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is no.

Myth : My people have state-of-the-art software development tools, after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality

software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

Myth : If we get behind schedule, we can add more programmers and catch up is called the Mongolian horde concept.

Reality : Software development is not a mechanistic process like manufacturing. In the words of Brooks: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth : If I decide to out source the software project to a third party, I can just relax and let them to firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

1.6.2 Customer Myths

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

Myth : A general statement of objectives is sufficient to begin writing programs, we can fill the details later.

Reality : A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth : Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality : It is true that software requirements change, but the impact of change varies with the time at which it is introduced. Figure 3 illustrates the impact of change. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly.

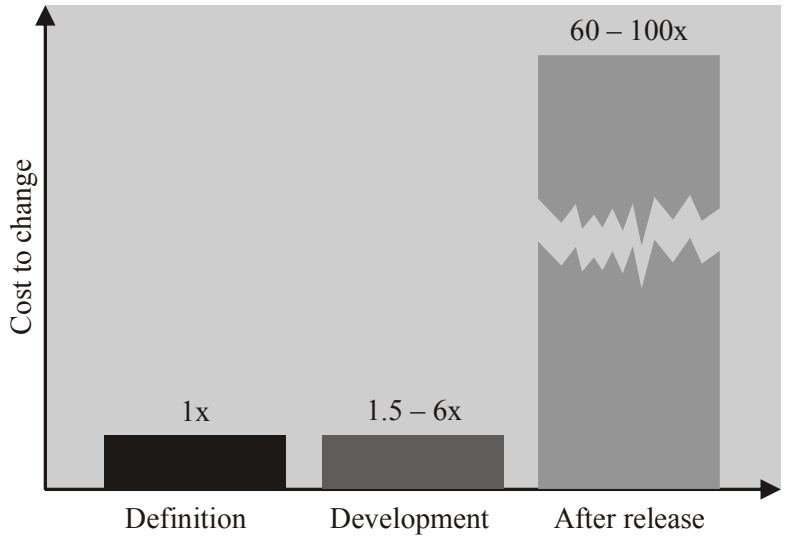


Fig 3: The impact of change

Resources have been committed and a design framework has been established. Change Can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier.

1.6.3 Practitioner's Myths

Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

Myth : Once we write the program and get it to work, our job is done.

Reality : Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth : Until I get the program "running" I have no way of assessing its quality.

Reality : One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the formal technical review. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth : The only deliverable work product for a successful project is the working program.

Reality : A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

Myth : Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality : Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

1.7 Qualities of Software Product

Product quality is an international standard for the evaluation of software quality. The fundamental objective of this standard is to address some of the well known human biases that can adversely affect the delivery and perception of a software development project. These biases include changing priorities after the start of a project or not having any clear definitions of "success". By clarifying, then agreeing on the project priorities and subsequently converting abstract priorities (compliance) to measurable values.

In the context of software engineering, software quality refers to two related but distinct notions that exist wherever quality is defined in a business context:

- Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product;
- Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

The qualities are as follows:

1. **Functionality** - *A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.*
   - ***Suitability*** - is the process and procedures used to establish the suitability of a system - that is, the ability of a system to meet the needs of a stakeholder or other user.
   - ***Accuracy***
   - ***Interoperability*** - is a property referring to the ability of diverse systems and organizations to work together (inter-operate). The term is often used in a technical systems engineering sense, or alternatively in a broad sense, taking into account social, political, and organizational factors that impact system to system performance.
   - ***Security*** - is a branch of computer technology known as Information Security as applied to computers and networks. The objective of computer security includes protection of information and property from theft, corruption, or natural disaster, while allowing the information and property to remain accessible and productive to its intended users. The term computer system security means the collective processes and mechanisms by which sensitive and valuable information and services are protected from publication, tampering or collapse by unauthorized activities or untrustworthy individuals and unplanned events respectively. The strategies and methodologies of computer security often differ from most other computer technologies because of its somewhat elusive objective of preventing unwanted computer behavior instead of enabling wanted computer behavior.
   - ***Functionality Compliance*** - The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functionality.
2. **Reliability** - *A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.*
   - ***Maturity***
   - ***Fault Tolerance*** – It is the property that enables a system (often computer-based) to continue operating properly in the event of the failure of (or one or more faults within) some of its components. A newer approach is progressive enhancement. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naïvely-designed system in which even a small failure can cause total breakdown. Fault-tolerance is particularly sought-after in high-availability or life-critical systems.
   - ***Recoverability***
   - ***Reliability Compliance***
3. **Usability** - *A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.*
   - ***Understandability***
   - ***Learnability*** - In software testing learnability, according to ISO/IEC 9126, is the capability of a software product to enable the user to learn how to use it. Learnability may be considered as an aspect of usability, and is of major concern in the design of complex software applications.
   - ***Operability*** – It is the ability to keep an equipment, a system or a whole industrial installation in a safe and reliable functioning condition, according to pre-defined operational requirements. In a computing systems environment with multiple systems this includes the ability of products, systems and business processes to work together to accomplish a common task such as finding and returning availability of inventory for flight.
   - ***Attractiveness***
   - ***Usability Compliance***

4. **Efficiency** - *A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.*
   - ***Time Behaviour***
   - ***Resource Utilisation***
   - ***Efficiency Compliance***
5. **Maintainability** - *A set of attributes that bear on the effort needed to make specified modifications.*
   - ***Analyzability***
   - ***Changeability***
   - ***Stability***
   - ***Testability*** – It a property applying to an empirical hypothesis, involves two components:
     (1) the logical property that is variously described as contingency, defeasibility, or falsifiability, which means that counterexamples to the hypothesis are logically possible, and
     (2) the practical feasibility of observing a reproducible series of such counterexamples if they do exist. In short, a hypothesis is testable if there is some real hope of deciding whether it is true or false of real experience. Upon this property of its constituent hypotheses rests the ability to decide whether a theory can be supported or falsified by the data of actual experience. If hypotheses are tested, initial results may also be labeled inconclusive. In engineering, this refers to the capability of an equipment or system to be tested.
   - ***Maintainability Compliance***
6. **Portability** - *A set of attributes that bear on the ability of software to be transferred from one environment to another.*
   - ***Adaptability***
   - ***Installability***
   - ***Co-Existence***
   - ***Replaceability***
   - ***Portability Compliance***

1.8 Principles of Software Engineering

1.8.1 Separation of Concerns

Separation of concerns is recognition of the need for human beings to work within a limited context. As described by G. A. Miller, the human mind is limited to dealing with approximately seven units of data at a time. A unit is something that a person has learned to deal with as a whole - a single abstraction or concept. Although human capacity for forming abstractions appears to be unlimited, it takes time and repetitive use for an abstraction to become a useful tool; that is, to serve as a unit.

When specifying the behavior of a data structure component, there are often two concerns that need to be dealt with: basic functionality and support for data integrity. A data structure component is often easier to use if these two concerns are divided as much as possible into separate sets of client functions. It is certainly helpful to clients if the client documentation treats the two concerns separately. Further, implementation documentation and algorithm descriptions can profit from separate treatment of basic algorithms and modifications for data integrity and exception handling.

There is another reason for the importance of separation of concerns. Software engineers must deal with complex values in attempting to optimize the quality of a product. From the study of algorithmic complexity, we can learn an important lesson. There are often efficient algorithms for optimizing a single measurable quantity, but problems requiring optimization of a combination of quantities are almost always NP-complete. Although it is not a proven fact, most experts in complexity theory

believe that NP-complete problems cannot be solved by algorithms that run in polynomial time.

In view of this, it makes sense to separate handling of different values. This can be done either by dealing with different values at different times in the software development process, or by structuring the design so that responsibility for achieving different values is assigned to different components.

As an example of this, run-time efficiency is one value that frequently conflicts with other software values. For this reason, most software engineers recommend dealing with efficiency as a separate concern. After the software is design to meet other criteria, it's run time can be checked and analyzed to see where the time is being spent. If necessary, the portions of code that are using the greatest part of the runtime can be modified to improve the runtime.

1.8.2 Modularity

The principle of modularity is a specialization of the principle of separation of concerns. Following the principle of modularity implies separating software into components according to functionality and responsibility. Parnas wrote one of the earliest papers discussing the considerations involved in modularization. A more recent work, describes a responsibility-driven methodology for modularization in an object-oriented context.

1.8.3 Abstraction

The principle of abstraction is another specialization of the principle of separation of concerns. Following the principle of abstraction implies separating the behavior of software components from their implementation. It requires learning to look at software and software components from two points of view: what it does, and how it does it.

Failure to separate behavior from implementation is a common cause of unnecessary coupling. For example, it is common in recursive algorithms to introduce extra parameters to make the recursion work. When this is done, the recursion should be called through a non-recursive shell that provides the proper initial values for the extra parameters. Otherwise, the caller must deal with a more complex behavior that requires specifying the extra parameters. If the implementation is later converted to a non-recursive algorithm then the client code will also need to be changed.

Design by contract is an important methodology for dealing with abstraction. The basic ideas of design by contract are sketched by Fowler and Scott . The most complete treatment of the methodology is given by Meyer.

1.8.4 Anticipation of Change

Computer software is an automated solution to a problem. The problem arises in some context, or domain that is familiar to the users of the software. The domain defines the types of data that the users need to work with and relationships between the types of data.

Software developers, on the other hand, are familiar with a technology that deals with data in an abstract way. They deal with structures and algorithms without regard for the meaning or importance of the data that is involved. A software developer can think in terms of graphs and graph algorithms without attaching concrete meaning to vertices and edges.

Working out an automated solution to a problem is thus a learning experience for both software developers and their clients. Software developers are learning the domain that the clients work in. They are also learning the values of the client: what form of data presentation is most useful to the client, what kinds of data are crucial and require special protective measures.

The clients are learning to see the range of possible solutions that software technology can provide. They are also learning to evaluate the possible solutions with regard to their effectiveness in meeting the clients needs.

If the problem to be solved is complex then it is not reasonable to assume that the best solution will be worked out in a short period of time. The clients do, however, want a timely solution. In most cases, they are not willing to wait until the perfect solution is worked out. They want a reasonable solution soon; perfection can come later. To develop a timely solution, software developers need to

know the requirements: how the software should behave. The principle of anticipation of change recognizes the complexity of the learning process for both software developers and their clients. Preliminary requirements need to be worked out early, but it should be possible to make changes in the requirements as learning progresses.

Coupling is a major obstacle to change. If two components are strongly coupled then it is likely that changing one will not work without changing the other.

Cohesiveness has a positive effect on ease of change. Cohesive components are easier to reuse when requirements change. If a component has several tasks rolled up into one package, it is likely that it will need to be split up when changes are made.

1.8.5 Generality

The principle of generality is closely related to the principle of anticipation of change. It is important in designing software that is free from unnatural restrictions and limitations. One excellent example of an unnatural restriction or limitation is the use of two digit year numbers, which has led to the "year 2000" problem: software that will garble record keeping at the turn of the century. Although the two-digit limitation appeared reasonable at the time, good software frequently survives beyond its expected lifetime.

For another example where the principle of generality applies, consider a customer who is converting business practices into automated software. They are often trying to satisfy general needs, but they understand and present their needs in terms of their current practices. As they become more familiar with the possibilities of automated solutions, they begin seeing what they need, rather than what they are currently doing to satisfy those needs. This distinction is similar to the distinction in the principle of abstraction, but its effects are felt earlier in the software development process.

1.8.6 Incremental Development

Fowler and Scott give a brief, but thoughtful, description of an incremental software development process. In this process, you build the software in small increments; for example, adding one use case at a time. An incremental software development process simplifies verification. If you develop software by adding small increments of functionality then, for verification, you only need to deal with the added portion. If there are any errors detected then they are already partly isolated so they are much easier to correct.

A carefully planned incremental development process can also ease the handling of changes in requirements. To do this, the planning must identify use cases that are most likely to be changed and put them towards the end of the development process.

1.8.7 Consistency

The principle of consistency is a recognition of the fact that it is easier to do things in a familiar context. For example, coding style is a consistent manner of laying out code text. This serves two purposes. First, it makes reading the code easier. Second, it allows programmers to automate part of the skills required in code entry, freeing the programmer's mind to deal with more important issues.

Consistency serves two purposes in designing graphical user interfaces. First, a consistent look and feel makes it easier for users to learn to use software. Once the basic elements of dealing with an interface are learned, they do not have to be relearned for a different software application. Second, a consistent user interface promotes reuse of the interface components. Graphical user interface systems have a collection of frames, panes, and other view components that support the common look. They also have a collection of controllers for responding to user input, supporting the common feel. Often, both look and feel are combined, as in pop-up menus and buttons. These components can be used by any program.

Meyer applies the principle of consistency to object-oriented class libraries. As the available libraries grow more and more complex it is essential that they be designed to present a consistent interface to the client. For example, most data collection structures support adding new data items. It is much easier to learn to use the collections if the name that adds is always used for this kind of operation.

1.9 Summary

Software Engineering (SE) is a profession dedicated to designing, implementing, and modifying software so that it is of high quality, affordable, maintainable, and fast to build. It is a "systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software, that is, the application of engineering to software. The goal of software engineering is, of course, to design and develop better software. However, what exactly does "better software" mean? In order to answer this question, this lesson introduces some common software quality characteristics. Six of the most important quality characteristics are maintainability, correctness, reusability, reliability, portability, and efficiency. Software crisis was a term used in the early days of computing science. The term was used to describe the impact of rapid increases in computer power and the complexity of the problems that could be tackled. In essence, it refers to the difficulty of writing correct, understandable, and verifiable computer programs. The roots of the software crisis are complexity, expectations, and change. Many causes of a software affliction can be traced to a mythology during the development of software. Software myths propagated misinformation and confusion. Software myths had a number of attributes that made them insidious. Today, most knowledgeable professionals recognize myths for what they are misleading attitudes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify, and remnants of software myths are still believed. The quality of software is assessed by a number of variables. These variables can be divided into external and internal quality criteria. External quality is what a user experiences when running the software in its operational mode. Internal quality refers to aspects that are code-dependent, and that are not visible to the end-user. External quality is critical to the user, while internal quality is meaningful to the developer only.

1.10 Self Assessment Questions

1. What do you mean by Software engineering?
2. Explain the aim of software engineering?
3. What are the characteristics of good software? Explain.
4. What do you understand by software Crisis?
5. What is a software myth? Explain the Management myth.
6. Explain the practitioner's software myth?
7. Explain the customer's software myth?
8. What do you mean by Software quality? Explain.
9. What are various principles of Software engineering?
10. What do you mean by modularity and abstraction?

1.11 References

- Introduction to Software Engineering - www.newagepublishers.com/
- Software Engineering By K.K. Aggarwal
- Bauer, Fritz et al., "Software Engineering: A Report on a Conference Sponsored by NATO Science Committee", NATO, 1968
- Schach, Stephen, "Software Engineering", Vanderbilt University, 1990
- Bertrand Meyer, *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice-Hall, 1994.
- Martin Fowler with Kendall Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley Object Technology Series, 1997
- D. L. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, 15(12):1053-1058, December 1972.
- John M. Vlissides and James O. Coplien and Norman L. Kerth, eds., *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.
- Sommerville, I (2000), Software Engineering, 6th Edition, Pearson Education Ltd., New Delhi
- Roger S. Pressman, Software Engineering : A Practioners Approach., 6th Edition, The McGraw Hill Companies, New York

Unit - 2 : Software Process Models

Structure of the Unit

2.0 Objectives

This chapter provides a general overview of

- Software Process
- Linear Sequence Model
- Spiral Model
- Prototyping Model
- RAD Model

2.1 Introduction

A software development process, also known as a software development life cycle (SDLC), is a structure imposed on the development of a software product. Similar terms include software life cycle and software process. It is often considered a subset of systems development life cycle. There are several models for such processes, each describing approaches to a variety of tasks or activities

that take place during the process. Some people consider a life cycle model a more general term and a software development process a more specific term. For example, there are many specific software development processes that 'fit' the spiral life cycle model. ISO 12207 is an ISO standard for software life cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software.

The software life cycle is composed of three types of processes:

- Planning
- Development
- Software Processes.

2.1.1 The Planning Process

It is the first step in the cycle. Its goal is to define the activities of the development and supporting processes, which includes their interactions and sequencing. The data produced by the processes and the design development environments, with definitions of the methods and tools to be used, are also determined during the planning process. The planning process scopes the complexity of the software and estimates the resources needed in the development activities.

2.1.2 The Development Process

Software development standards and methods are selected during the planning activity. Standards exist for all the processes in the software life cycle and are used as part of the quality assurance strategy. For better performance in terms of development time and quality of the final product, the plans should be generated at a point in time that provides timely direction to those involved in the life cycle process. The planning process should provide mechanisms for further refinement of the plans as the project advances. The strategy used to develop the software, also known as the **process model or software engineering paradigm**, is chosen based on the particular characteristics of the application to be developed. Further consideration in selecting a process model is given to project elements like the maturity level of the developers and the organization, tools to be used, existent process control, and the expected deliverables. Various process models have been proposed in the software engineering literature.

2.1.3 A Software Process

It outlines all steps or practices used to create a software application. From the customer's requirements to the finished product, the software process that is used determines the organization and flexibility of the project. There are several different software processes, and each describes their own solution to developing valid software.

This module will investigate several different software practices, including: waterfall, spiral, extreme programming (XP), and agile.

The waterfall software process is an almost linear path through software development. The waterfall model contains six major stages of development: requirements, design, code, test, integrate, maintain. Each stage is done one at a time with some little flexibility in repeating a stage. The waterfall model should be used on projects with few requirements changes.

The spiral model is a phased model that allows a program to go through several iterations of the development. The software goes through determination, evaluation, development, and planning phases until the final release of the product.

Agile methodologies, including XP, are more flexible programming processes that allow for more iterations and quicker prototype releases. Agile and XP practices are best used in small development groups with dynamic requirements.

2.2 Linear Sequential Model or Waterfall Model

The waterfall model is a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation and Maintenance.

The waterfall development model originates in the manufacturing and construction industries: highly structured physical environments in which after-the-fact changes are prohibitively costly, if not impossible. Since no formal software development methodologies existed at the time, this hardware-oriented model was simply adapted for software development.

The waterfall model derives its name due to the cascading effect from one phase to the other as is illustrated in Figure 1. In this model each phase has a well defined starting and ending point, with identifiable deliveries to the next phase.

Note that this model is sometimes referred to as the linear sequential model or the classic software life cycle.

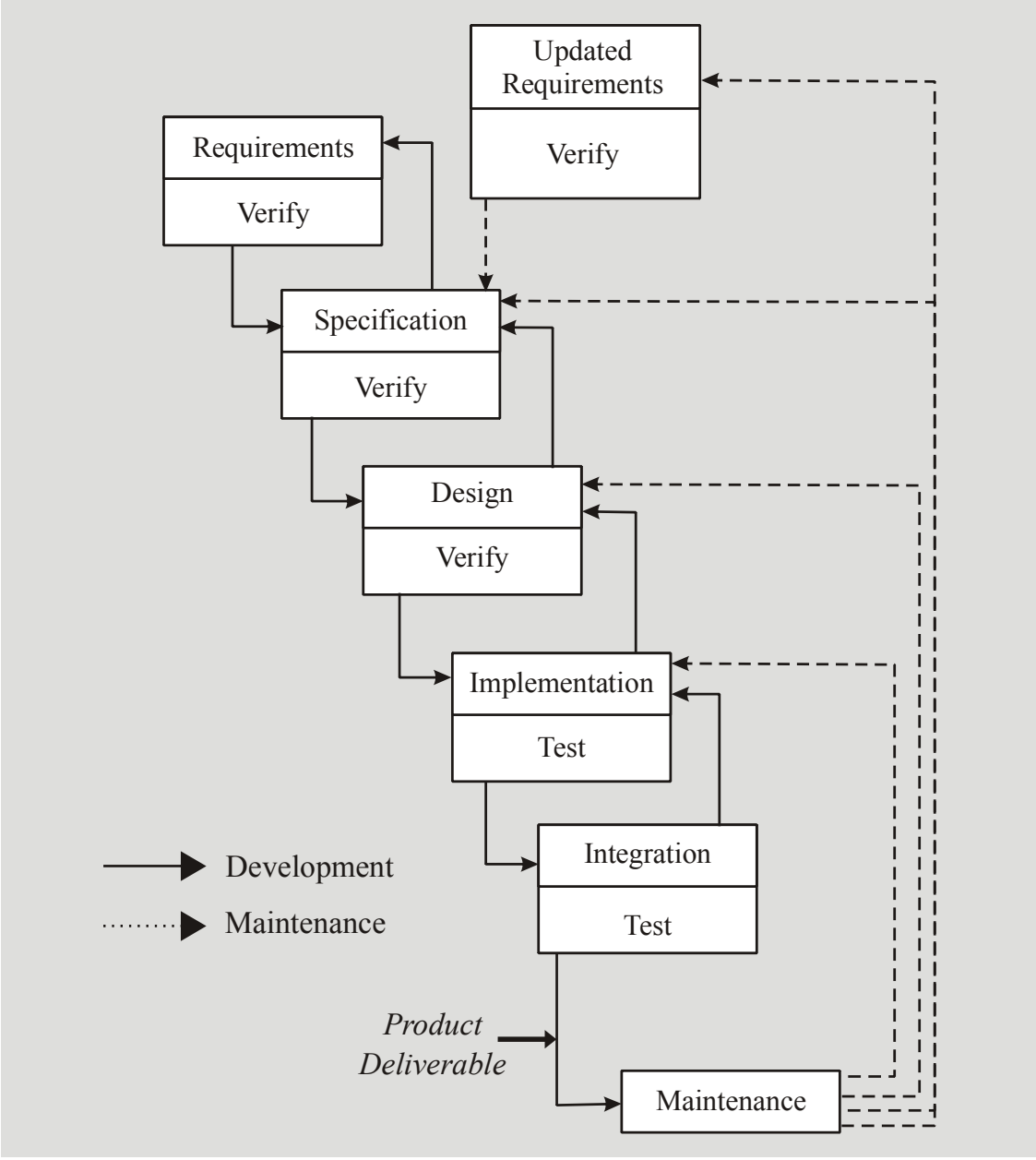


Fig. 2.1: Waterfall Model

The model consists of six distinct stages, namely:

2.2.1 Requirements Analysis Phase

As software is always of a large system (or business), work begins by establishing the requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when the software must have interface with other elements such as hardware, people and other resources. System is the basic and very critical requirement for the existence of software in any entity. So if the system is not in place, it should be engineered and put in place. In some cases, to extract the maximum output, the system should be re-engineered and spruced up. Once the ideal system is engineered or tuned, the development team studies the software requirement for the system.

In the requirements analysis phase :

- The problem is specified along with the desired service objectives (goals)
- The constraints are identified

2.2.2 Specification phase

This process is also known as feasibility study. In this phase, the development team visits the customer and studies their system. They investigate the need for possible software automation in the given system. By the end of the feasibility study, the team furnishes a document that holds the different specific recommendations for the candidate system. It also includes the personnel assignments, costs, project schedule, target dates etc.... The requirement gathering process is intensified and focused specially on software. To understand the nature of the program(s) to be built, the system engineer or "Analyst" must understand the information domain for the software, as well as required function, behaviour, performance and interfacing. The essential purpose of this phase is to find the need and to define the problem that needs to be solved .

In the specification phase the system specification is produced from the detailed definitions of the two points above. This document should clearly define the product function.

Note that in some text, the requirements analysis and specifications phases are combined and represented as a single phase.

2.2.3 System and software design phase

In this phase of the software development process, the software's overall structure and its nuances are defined. In terms of the client/server technology, the number of tiers needed for the package architecture, the database design, the data structure design etc... are all defined in this phase. A software development model is thus created. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

In the system and software design phase, the system specifications are translated into a software representation. The software engineer at this stage is concerned with:

- Data structure
- Software architecture
- Algorithmic detail and
- Interface representations

The hardware requirements are also determined at this stage along with a picture of the overall system architecture. By the end of this stage the software engineer should be able to identify the relationship between the hardware, software and the associated interfaces. Any faults in the specification should ideally not be passed 'down stream'

2.2.4 Implementation and testing phase

The design must be translated into a machine-readable form. The code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Programming tools like compilers, interpreters, debuggers etc... are used to generate the code. Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.

In the implementation and testing phase stage the designs are translated into the software domain

- Detailed documentation from the design phase can significantly reduce the coding effort.
- Testing at this stage focuses on making sure that any errors are identified and that the software meets its required specification.

Once the code is generated, the software program testing begins. Different testing methodologies are available to unravel the bugs that were committed during the previous phases. Different testing tools and methodologies are already available. Some companies build their own testing tools that are 'tailor made' for their own development operations.

2.2.5 Integration and system testing phase

In the integration and system testing phase all the program units are integrated and tested to ensure that the complete system meets the software requirements. After this stage the software is delivered to the customer [Deliverable – The software product is delivered to the client for acceptance testing.

2.2.6 Maintenance phase

The software will definitely undergo change once it is delivered to the customer. There can be many reasons for this change to occur. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period.

The maintenance phase is usually the longest stage of the software development. In this phase the software is updated to:

- Meet the changing customer needs
- Adapted to accommodate changes in the external environment
- Correct errors and oversights previously undetected in the testing phases
- Enhancing the efficiency of the software

Observe that feed back loops allow for corrections to be incorporated into the model. For example a problem/update in the design phase requires a 'revisit' to the specifications phase. When changes are made at any phase, the relevant documentation should be updated to reflect that change.

Advantages

- Testing is inherent to every phase of the waterfall model
- It is an enforced disciplined approach
- It is documentation driven, that is, documentation is produced at every stage

Disadvantages

The waterfall model is the oldest and the most widely used paradigm. However, many projects rarely follow its sequential flow. This is due to the inherent problems associated with its rigid format. Namely:

- It only incorporates iteration indirectly, thus changes may cause considerable confusion as the project progresses.
- As The client usually only has a vague idea of exactly what is required from the software product, this WM has difficulty accommodating the natural uncertainty that exists at the beginning of the project.
- The customer only sees a working version of the product after it has been coded. This may result in disaster if any undetected problems are precipitated to this stage.

2.3 Prototype Model

A prototype is a working model that is functionally equivalent to a component of the product. A prototype typically simulates only a few aspects of the features of the eventual program, and may be completely different from the actual implementation.

The conventional purpose of a prototype is to allow users of the software to evaluate developers' proposals for the design of the eventual product by actually trying them out, rather than having to interpret and evaluate the design based on descriptions. Prototyping can also be used by end users to describe and prove requirements that developers have not considered, so "controlling the prototype" can be a key factor in the commercial relationship between solution providers and their clients.

In many instances the client only has a general view of what is expected from the software product. In such a scenario where there is an absence of detailed information regarding the input to the system, the processing needs and the output requirements, the prototyping model may be employed. This model reflects an attempt to increase the flexibility of the development process by allowing the client to interact and experiment with a working representation of the product. The developmental process only continues once the client is satisfied with the functioning of the prototype. At that stage the developer determines the specifications of the client's real needs.

The process of prototyping involves the following steps

- **Identify basic requirements**

  Determine basic requirements including the input and output information desired. Details, such as security, can typically be ignored.

- **Develop Initial Prototype**

  The initial prototype is developed that includes only user interfaces.

- **Review**

  The customers, including end-users, examine the prototype and provide feedback on additions or changes.

- **Revise and Enhance the Prototype**

  Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps #3 and #4 may be needed.

The following section examines two version of the prototyping model:

2.3.1 Version I

This approach, as illustrated in Fig 2.2, uses the prototype as a means of quickly determining the needs of the client; it is discarded once the specifications have been agreed on. The emphasis of the prototype is on representing those aspects of the software that will be visible to the client/user (e.g. input approaches and output formats). Thus it does not matter if the prototype hardly works.

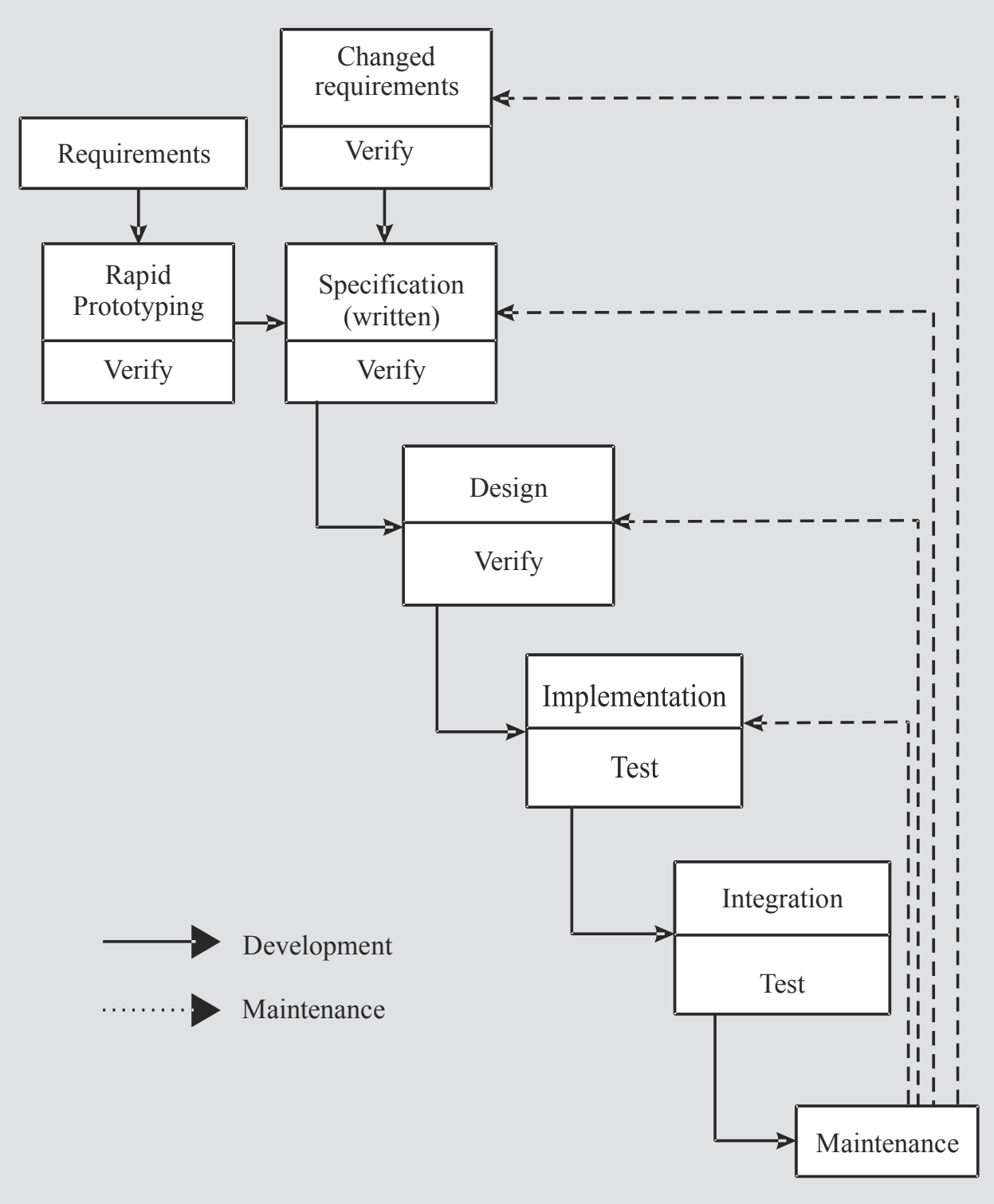


Fig. 2.2 Prototype development Version I

Note that if the first version of the prototype does not meet the client's needs, then it must be rapidly converted into a second version.

2.3.2 Version II

In this approach, as illustrated in Fig 2.3, the prototype is actually used as the specifications for the design phase. This advantage of this approach is speed and accuracy, as not time is spent on drawing up written specifications. The inherent difficulties associated with that phase (i.e. incompleteness, ontradictions and ambiguities) are then avoided.

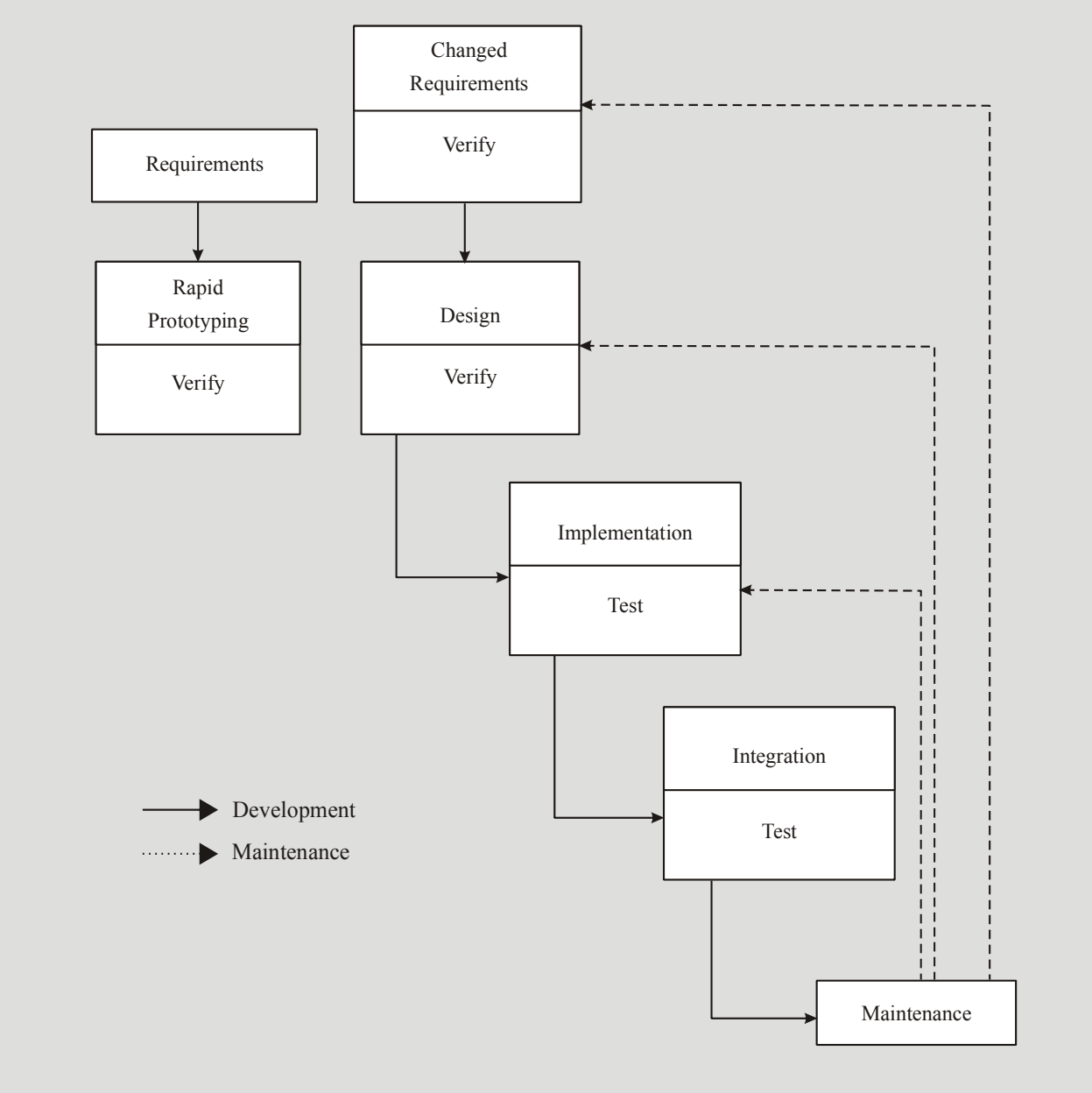


Fig. 2.3 Prototype development Version II

Advantages of prototyping

There are many advantages to using prototyping in software development – some tangible, some abstract.

- **Reduced time and costs**: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of what the user really wants can result in faster and less expensive software.
- **Improved and increased user involvement**: Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality. The final product is more likely to satisfy the users desire for look, feel and performance.

Disadvantages of prototyping

Using, or perhaps misusing, prototyping can also have disadvantages.

- **Insufficient analysis**: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.
- **User confusion of prototype and finished system**: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error-checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to conflict.
- **Developer misunderstanding of user objectives**: Developers may assume that users share their objectives (e.g. to deliver core functionality on time and within budget), without understanding wider commercial issues. For example, user representatives attending Enterprise software (e.g. People Soft) events may have seen demonstrations of "transaction auditing" (where changes are logged and displayed in a difference grid view) without being told that this feature demands additional coding and often requires more hardware to handle extra database accesses. Users might believe they can demand auditing on every field, whereas developers might think this is feature creep because they have made assumptions about the extent of user requirements. If the solution provider has committed delivery before the user requirements were reviewed, developers are between a rock and a hard place, particularly if user management derives some advantage from their failure to implement requirements.
- **Developer attachment to prototype**: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. (This may suggest that throwaway prototyping, rather than evolutionary prototyping, should be used.)
- **Excessive development time of the prototype**: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.
- **Expense of implementing prototyping**: the start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling, or both. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.

  In both versions the prototype is discarded early in the life cycle. However, one way of ensuring that the product is properly designed and implemented is to implement the prototype in a different programming language from that of the product.

2.4 RAD Model

RAD is a linear sequential software development process model that emphasis an extremely short development cycle using a component based construction approach. If the requirements are well understood and defines, and the project scope is constraint, the RAD process enables a development team to create a fully functional system with in very short time period.

One of the conveniences of developing software is that it is not a physical tool that can be lost once it gets developed or manufactured. Codes are used to implement the software and it does not disappear together with the product. We can re-use the code all over again in another software. We just make a little change in the interface to fit the requirement of the client and we have a brand new program. To make our lives even better in developing software, there are tools coming out of the marking that serves as code generators. No need to create complicated codes, we just run the system through our preferences and we have a fully functional program.

The idea of reusing codes and tools is what defines another form of Software Development Life Cycle called Rapid Application Development, or RAD (see fig. 2.4). This form of software development constantly refers to the objective. Instead of creating original coding, developers use other tools such as software development kits and other graphic user interfaces to create programs. RAD also relies heavily on the available codes and reuses it to fit the intended program. Since RAD uses GUI, development kits and existing codes, software development will be faster.

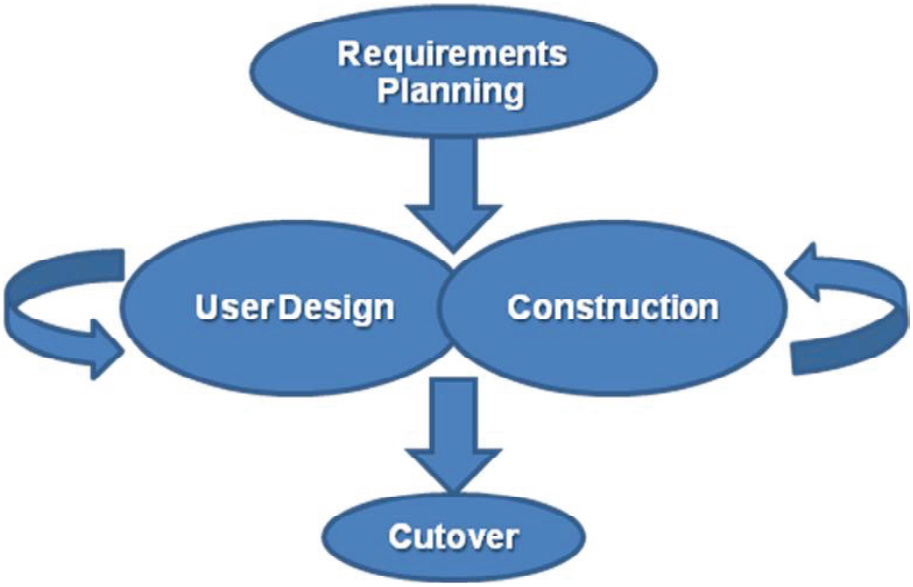


Fig. 2.4 RAD Model

The operational development of the software also works like the Prototype version. Instead of planning out and creating a single documentation, developers only have to look at the objective and use it as a "motivation" to create a program. It will be constantly tested and checked as the new "prototype" is released. Developers constantly create screens and show a workflow until it's approved by users. It is a very simple process since it doesn't have to use so many original ideas. It's basically an aggregate of different tools and cleverly using them to create completely different software.

RAD focuses more on the visual instead of the coding or software development. Since it already uses tools for generating codes, developers will have more time setting up the GUI of the software rather than focusing on the root structure of the program. The development cycle also enlists the help of the users to make sure the product being developed could match their taste. This will ensure a good user feedback since their ideas was acknowledged and had an actual impact on the product development. RAD reduces stress in creating GUI, coding and structure since it has a remote possibility of being trashed by users.

RAD (rapid application development) is a concept that products can be developed faster and of higher quality through :

- Gathering requirements using workshops or focus groups
- Prototyping and early reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that defers design improvements to the next product version
- Less formality in reviews and other team communication

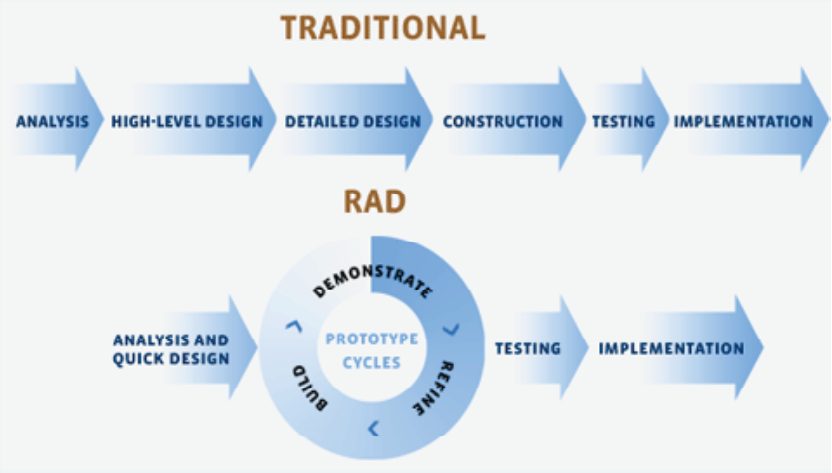


Fig. 2.5 Traditional model V/s RAD model

Some companies offer products that provide some or all of the tools for RAD software development. (The concept can be applied to hardware development as well.) These products include requirements gathering tools, prototyping tools, computer-aided software engineering tools, language development environments such as those for the Java platform, groupware for communication among development members, and testing tools. RAD usually embraces object-oriented programming methodology, which inherently fosters software re-use. The most popular object-oriented programming languages, C++ and Java, are offered in visual programming packages often described as providing rapid application development.

2.4.1 Development Methodology

The traditional software development cycle follows a rigid sequence of steps with a formal sign-off at the completion of each. A complete, detailed requirements analysis is done that attempts to capture the system requirements in a Requirements Specification. Users are forced to "sign-off" on the specification before development proceeds to the next step. This is followed by a complete system design and then development and testing.

But, what if the design phase uncovers requirements that are technically unfeasible, or extremely expensive to implement? What if errors in the design are encountered during the build phase? The elapsed time between the initial analysis and testing is usually a period of several months. What if business requirements or priorities change or the users realize they overlooked critical needs during the analysis phase? These are many of the reasons why software development projects either fail or don't meet the user's expectations when delivered.

RAD is a methodology for compressing the analysis, design, build, and test phases into a series of short, iterative development cycles. This has a number of distinct advantages over the traditional sequential development model.RAD projects are typically staffed with small integrated teams comprised of developers, end users, and IT technical resources. Small teams, combined with short iterative development cycles, optimizes speed, unity of vision and purpose, effective informal communication and simple project management.

2.4.2 RAD Model Phases

RAD model has the following phases:

2.4.2.1 Business Modeling

The information flow among business functions is defined by answering questions like what information drives the business process, what information is generated, who generates it, where does the information go, who process it and so on.

2.4.2.2 Data Modeling

The information collected from business modeling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified and the relation between these data objects (entities) is defined.

2.4.2.3 Process Modeling

The data object defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting or retrieving a data object.

2.4.2.4 Application Generation

Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques.

2.4.2.5 Testing and Turn over

Many of the programming components have already been tested since RAD emphasis reuse. This reduces overall testing time. But new components must be tested and all interfaces must be fully exercised.

Advantage and Disadvantages

RAD reduces the development time and reusability of components help to speed up development. All functions are modularized so it is easy to work with.

For large projects RAD require highly skilled engineers in the team. Both end customer and developer should be committed to complete the system in a much abbreviated time frame. If commitment is lacking, RAD will fail. RAD is based on Object Oriented approach and if it is difficult to modularize the project the RAD may not work well.

2.5 Incremental Model

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 utilization 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.

This model, illustrated in Fig. 2.6, derives its name from the way in which the software is built. More specifically, the model is designed, implemented and tested as a series of incremental builds until the product is finished. A build consists of pieces of code from various modules that interact together to provide a specific function.

At each stage of the IM a new build is coded and then integrated into the structure, which is tested as a whole. Note that the product is only 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.

However, unlike prototyping the IM focuses on the delivery of an operational product at the end of each increment.

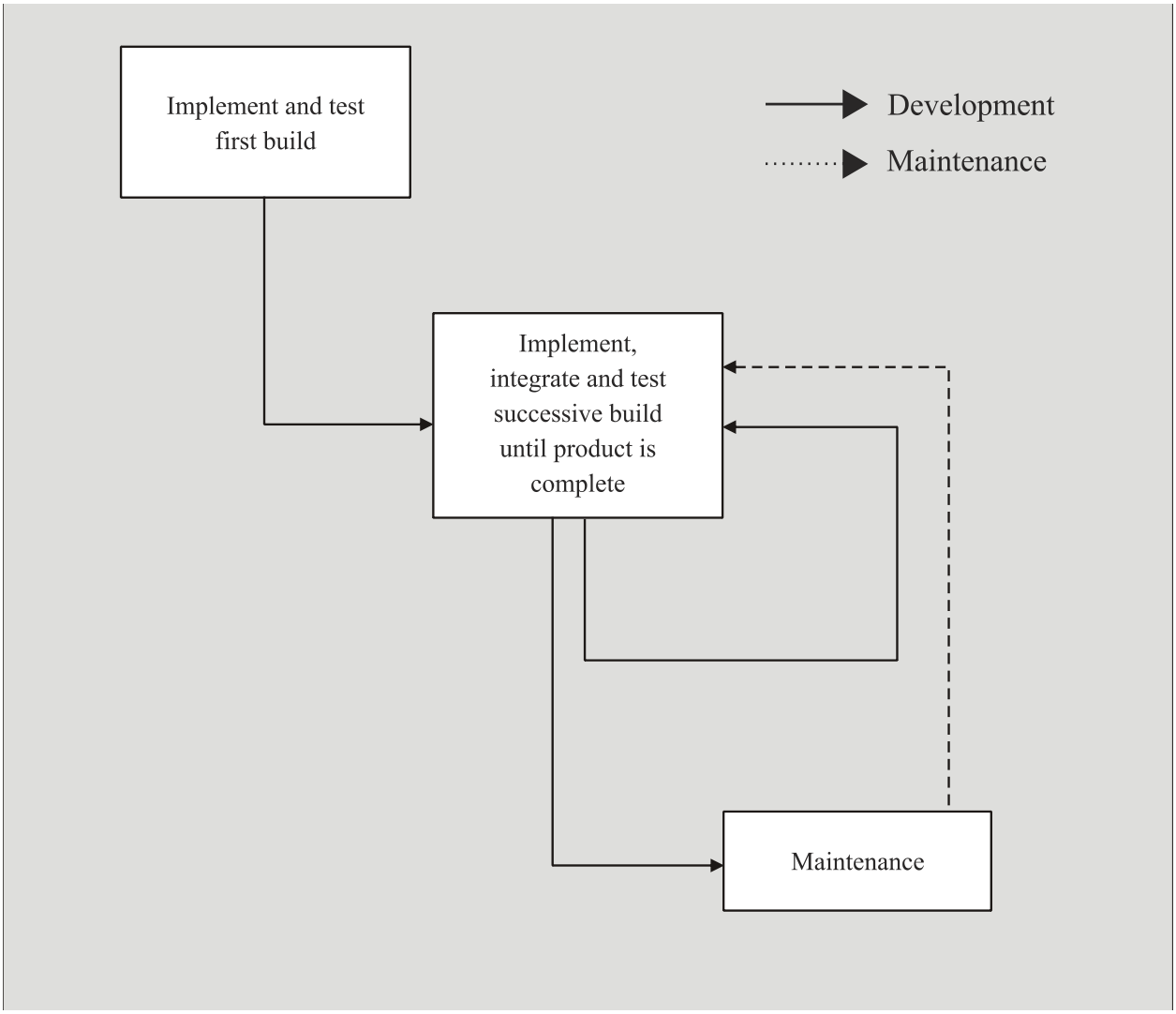


Fig. 2.6 Incremental Model

An example of this incremental approach is observed in the development of word processing applications where the following services are provided on subsequent builds:

1. Basic file management, editing and document production functions
2. Advanced editing and document production functions
3. Spell and grammar checking
4. Advance page layout

The first increment is usually the core product which addresses the basic requirements of the system. This maybe either be used by the client or subjected to detailed review to develop a plan for the next increment. This plan addresses the modification of the core product to better meet the needs of the customer, and the delivery of additionally functionality. More specifically, at each stage

- The client assigns a value to each build not yet implemented
- The developer estimates cost of developing each build
- The resulting value-to-cost ratio is the criterion used for selecting which build is delivered next

Essentially the build with the highest value-to-cost ratio is the one that provides the client with the most functionality (value) for the least cost. Using this method the client has a usable product at all of the development stages.

2.6 Summary

A **software development process**, also known as a software development life cycle (SDLC), is a structure imposed on the development of a software product. Similar terms include software life cycle and software process. It is often considered a subset of systems development life cycle. The **waterfall model** is a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation and Maintenance. A **prototype** is a working model that is functionally equivalent to a component of the product. A prototype typically simulates only a few aspects of the features of the eventual program, and may be completely different from the eventual implementation. **RAD** is a linear sequential software development process model that emphasis an extremely short development cycle using a component based construction approach. If the requirements are well understood and defines, and the project scope is constraint, the RAD process enables a development team to create a fully functional system with in very short time period. 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.

2.7 Self Assessment Questions

1. What do you mean by a Software process?
2. What do you understand by software models?
3. What is Linear Sequential model? Explain its stages.
4. What is RAD? Explain its stages.
5. What is a Prototype model?
6. Compare Waterfall model and Spiral model.
7. What is a spiral model? What are its advantages and disadvantages?
8. Explain advantages and disadvantages of a waterfall model.
9. What are advantages and disadvantages of a prototyping model?

2.8 References

- Roger S. Pressman, Software Engineering : Practioners approach, 6th Edition, The McGrawhill Companies, New York.
- Sommerville, I (2000), Software Engineering, 6th Edition, Pearson
- Pratap K.J. Mohpatra, Software Engineering, A lifecycle Approach, Ist Edition, New age Informational Publishers.

Unit - 3 : Software Process Evolutionary Models

Structure of the Unit

3.0 Objectives

After going through this unit student will be able to:

- To understand the evolutionary model concept
- Understand various evolutionary process models
- Understand about fourth generation techniques.
- Able to compare various software process models

3.1 Introduction

Evolutionary Models take the concept of "evolution" into the engineering paradigm. Therefore Evolutionary Models are iterative. They are built in a manner that enables software engineers to develop increasingly more complex and complete versions of the software. There is growing recognition that software, like all complex systems, evolves over a period of time. Product requirements often change as development proceeds, building a straight path to an end product unrealistic; market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined.

Evolutionary Software Process Models

As business and product requirements often change during the development phase, a limited version must be introduced to meet competitive or business pressure. 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.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop more complete versions of the software.

3.2 The Incremental Model

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable "increment" of the software.

In this approach of software development the model is designed, implemented and tested incrementally that is, functionality is delivered in small increments and added till the product is completed. It involves both development and maintenance. The product is considered to be complete when all of its requirements are fulfilled. This model combines the elements of the waterfall model along with the iterative way of prototyping. This model is also known as **Iterative Enhancement model.**

The incremental model applies linear sequences in a staggered fashion as calendar time progresses.

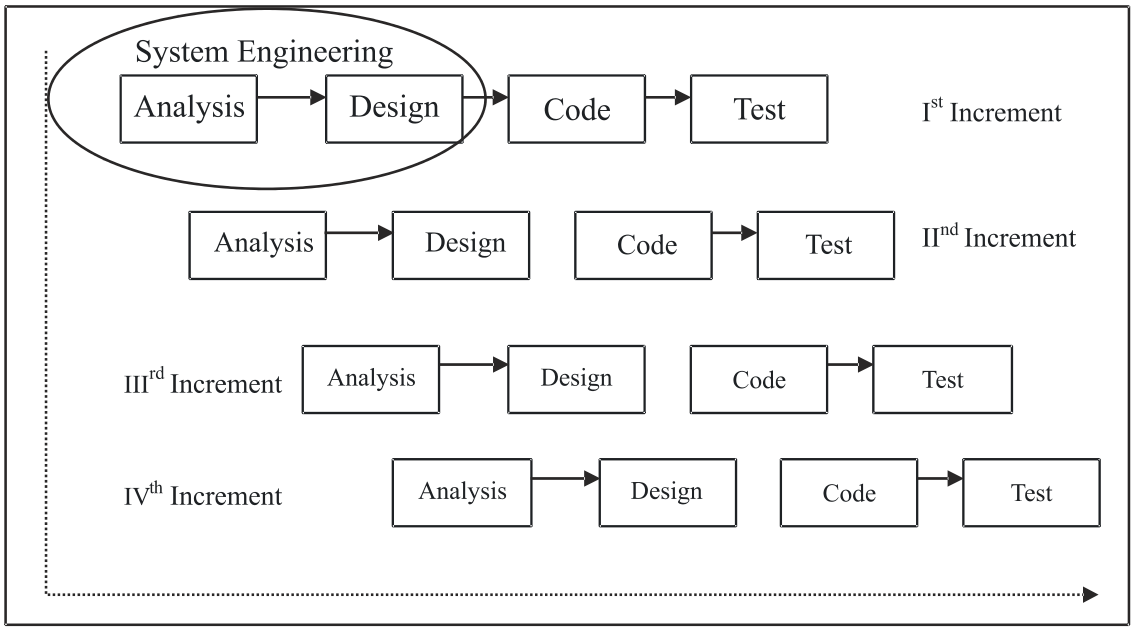


Fig. 3.1 Calendar Time

For example : In Document setting, software development produces

Ist phase – Basic file management, editing and document production

- **IInd phase –** More sophisticated editing and document production capabilities
- **IIIrd Phase –** Spelling and Grammar checking
- **IVth phase –** Advanced page layout capability

When an incremental model is used, the first increment is often **a core product** (Basic requirements provided). That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered.

But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user. **The difference between prototype and incremental model is that incremental model focuses on the delivery of an operational product.** It is particularly useful when staffing is unavailable for a complete implementation by the business deadline established for the project.

In this model the product is decomposed into a number of components. Each of the components is designed and built separately and delivered to the client when it is complete. This eliminates a long development time. Overall functionality of the product is delivered in small increments. Each increment uses the waterfall approach.

Iterative development segments the deliverable system functionality into iterations. Each linear sequence produces a deliverable product of the software. The first increment is often a core product in which basic requirements are addressed but the supplementary features are not delivered. The core product is used and evaluated by the customer and reported, based upon which plan is developed for the next increment. This process is repeated until the complete product is produced.

Advantages of the Incremental Model

- User defines the essential requirements. As focus is on necessary features, the amount of unnecessary functionality is reduced.
- Enhancements from initial increments help to develop later stages. User is benefited at an early stage. User can plan without knowing all the details
- As the components are delivered as soon as the design is made, the changes in the requirements are reduced.
- Smaller projects are easier to control and manage. Requirements can be added in later stages.
- Costs are reduced as iteration is planned and lessons learned can be incorporated through increments
- Early increments act as a prototype and deliver value early
- Prioritization leads to robustness
- Lower risk of project failure

Disadvantages of the Incremental Model

- Working on a large product can help to achieve creative work than working with small project modules.
- Some modules are difficult to divide into functional units. Increments at later stage may require modifications to be made in the previous stages.
- Needs client involvement throughout project which leads to project delay
- Informal requests may lead to confusion and hard to identify common facilities needed in all increments

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people.

3.3 The Spiral Model

The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software.

In the spiral model, software is developed in a series of incremental releases. During initial iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

The spiral model is a software development process which combines elements of design and prototyping, to unite advantages of top-down and bottom-up concepts. It is also known as the spiral lifecycle model. It is a systems development method (SDM) used in information technology. This model combines the features of the prototyping model and the waterfall model. The spiral model is intended for large, expensive and complicated projects.

Barry Boehm defined the spiral model in his 1988 article "A Spiral Model of Software Development and Enhancement". This model was the first model to explain iterative matters of development. The iterations were typically 6 months to 2 years long. Each phase starts with a design goal and ends with the client reviewing the progress. Analysis and engineering efforts are applied at each phase of the project, keeping the goal of the project in mind.

The steps in the spiral model can be generalized as follows:

- New system requirements are defined in detail. This usually involves interviewing a number of users representing all the external or internal users and other aspects of the existing system.
- A preliminary design is created for the new system.
- First prototype of the new system is constructed from the preliminary design.
- Second prototype is evolved as follows:

- evaluating the first prototype for its strengths, weaknesses, and risks;
- defining the requirements for the second prototype
- planning and designing the second prototype
- constructing and testing the second prototype

The spiral model is divided into a number of framework activities also called task regions. Spiral model contains **six task regions**:-

- **Customer Communication** – Tasks required to establish effective communication between developer and customer.
- **Planning** – Tasks required to define resources, timeliness and other project related information.
- **Risk Analysis** – Tasks required to assess both technical and management risks.
- **Engineering** – Tasks required to build one or more representations of the application.
- **Construction & Release** – Tasks required to construct, test, install and provide user support (i.e. documentation and training)
- **Customer Evaluation** – Tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

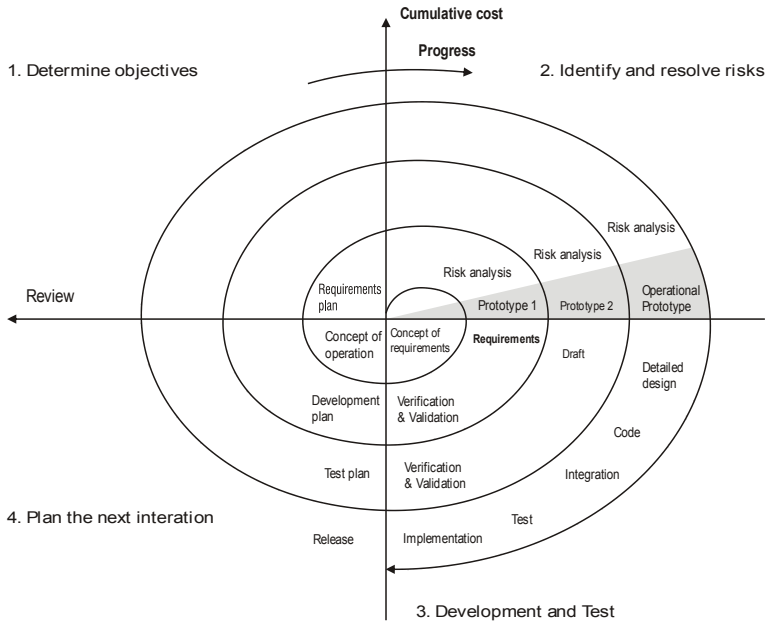


Fig.3.2

In the above diagram the radial dimension is cumulative cost to complete the cycle. Angular dimension is progress within this cycle. Four quadrants represent four standard periods in each cycle. These are Goal setting, Risk minimization, Development and Planning.

- Activities of the First and Second Quadrants are to set objectives, discover alternatives and to discover constraints. It also helps to gather information to assess risks like, prototyping, simulation, benchmarking, reference checking, administering user surveys and modeling. It helps in choosing the best alternative.
- Activities of the Third and Fourth Quadrants are to develop the product of the cycle based on risks like performance or user interfaces, program development or interface-control and Validation of the work product at the end of this period, to decide whether to continue, to review for planning next cycle, to allocating resources and to gather commitment.

  The spiral model is often used in large projects. For smaller projects, the concept of agile software development is becoming an alternative. The US military has adopted the spiral model for its Future Combat Systems program.

The spiral model is a realistic approach to the development of large scale systems and software. It uses prototyping as a risk reduction mechanism, but more important it enables the developer to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle, but incorporates it into an iterative framework that more realistically reflects the real world.

Advantages of the Spiral Model

- The spiral model promotes quality assurance through prototyping at each stage in system development.
- It provides flexibility
- It is developmental and focuses on component reuse
- It provides cohesion between stages and focuses on outputs
- It provides repeated research training and minimizes many risks
- It encourages prototyping and minimizes unnecessary elaborative specification
- It enables rework when needed
- It incorporates existing models
  - Low risk of making mistakes – waterfall
  - Stable requirements – transform
  - Good customer support, management control, integration, decision making – evolutionary
  - Tool availability – rapid prototyping or transform
- It accommodates maintenance as well as initial development
- It focuses on quality assurance and can integrate hardware and software development

Disadvantages of the Spiral Model

- Customers may feel insecure using this model
- Good risk-assessment skills are required
- Spiral model was considered to be immature
- It is not agile

The spiral model demands a direct consideration of technical risks during all stages of the project and, if properly applied, should reduce risks before they become problematic. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur. Finally, the model has not been used as widely as the linear sequential or prototyping paradigms.

3.4 The component Assembly Model

In the majority of software projects, there is some software reuse. This usually happens informally when people working on the project know of designs or code which is similar to that required. This informal reuse takes place irrespective of the development process that is used.

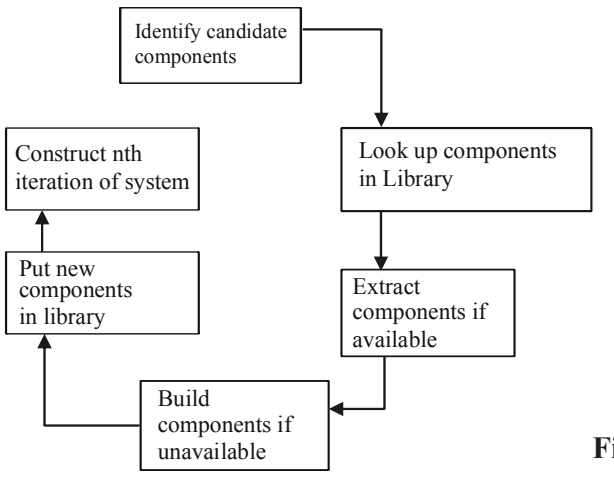


Fig. 3.3

This model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an interactive approach to the creation of software. However, this model composes applications from prepackaged software components.

- The activity begins with the identification of candidate classes by examining the data that are to be manipulated by the application and the algorithms that will be applied to accomplish the manipulation.
- The classes (called components) are stored in a class library.
- The class library is searched to determine if the desired classes already exist, if exists than they are extracted from the library and reused.
- If does not exists, it is engineered using object oriented techniques.
- When new component is engineered, it is stored in the library.

Process flow & then returns to the spiral and will re-enter the component assembly iteration during subsequent passes through the engineering activity.

This reuse-oriented approach relies on a large base of reusable software components and some integrating framework for these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as text formatting or numeric calculation.

Component-based software engineering has the obvious advantage of reducing the amount of software to be developed and so reducing the cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable and this may lead to a system that does not meet the real needs of users.

This model leads to software reuse and reusability provides number of benefits to software engineers.

3.5 The Concurrent Development Model

This model can be represented schematically as a series of major technical activities, tasks and their associated states. It makes use of state charts to represents the concurrent relationship among tasks associated within a framework of activities. It is represented schematically by a series of major technical tasks, and associated states. The user's need, management decisions and review results drive the over-all progression of the development.

The concurrent model is often more appropriate for system engineering projects where different engineering teams are involved. The concurrent process model defines a series of events that will trigger transitions from state to state for each of software engineering activities, actions or tasks. This generates the event analysis model correction which will trigger the analysis action from done state to awaiting changes state.

The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. It defines a network of activities. Each activity on the network exists simultaneously with other activities, actions or tasks.

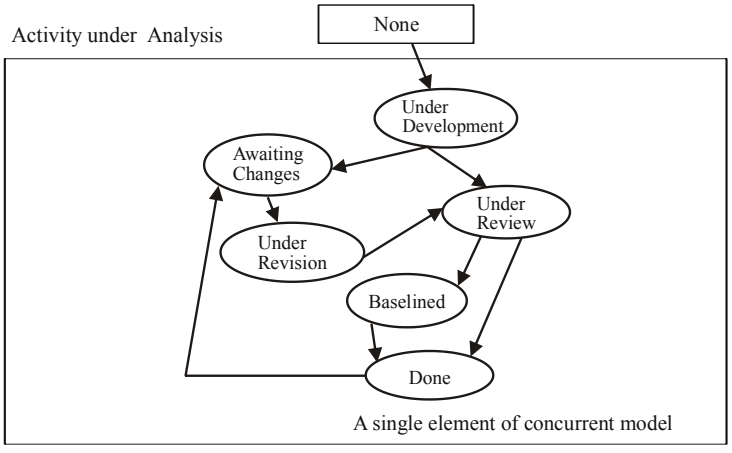


Fig. 3.4

As per the above figure, it is a schematic representation of one activity within the concurrent process model. *The activity analysis may be in any one of the states at any given time.* Similarly, other activities can be represented in the same manner. All activities exist concurrently but reside in different states. For example, 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.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities. The concurrent process model is often used as the paradigm for the development of client/server applications.

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.

3.6 The Formal Methods Model

This model encompasses a set of activities that lead to mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called *cleanroom software engineering*, is currently applied by some software development organizations. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design. However, the high cost of using formal methods means that they are usually only used in the development of high-integrity systems, where safety or security is of utmost importance.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Incompleteness and inconsistency can be discovered and corrected more easily not through adhoc review, but through the application of mathematical analysis.

Although it is not a mainstream approach, the formal methods model offers the promise of defect-free software.

Drawbacks

- The development of this model is currently quite time-consuming and expensive.
- Few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

3.7 Fourth Generation Techniques model

The term 4GT is a broad array of software tools that have one thing in common - Each enables the software engineer to specify some characteristic of software at a high level. The 4GT paradigm for software engineer focuses on the ability to specify software using specialized language forms or a graphic notation that describes the problems to be solved in terms that the customers can understand.

The Fourth Generation Technique (4GT) is based on NPL that is the Non-Procedural Language techniques. Depending upon the specifications made, the 4GT approaches uses various tools for the automatic generation of source codes. It is the vital tool which uses the non-procedural language for Report generation, Database query, Manipulation of data, Interaction of screen, Definition, Generation of code, Spread Sheet capabilities, High level graphical capacity, etc.

Like any other models used, the 4GT approach requires the requirement analysis step. Once the requirement analysis is done upto the expectations, its translation into the operational prototype begins. The most important phase in the 4GT approach is the "customer developer approach", all the major

decisions regarding the implementations, costs and functioning of the system is taken in this phase.

The Fourth Generation Technique (4GT) is usually successful in implementing smaller applications as we can easily switch from the requirement analysis phase to the implementation phase. Prototypes of any applications can be easily developed with the help of the 4GT approach. This prototype helps the clients to give a rough idea of how the system will look when it is done. The 4GT model is very useful approach in small projects but it is not a dominant approach for large software development.

To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other solution integration activities that are required in other software engineering paradigms. In addition the 4GT designed software must be built in a manner that enables maintenance to be performed expeditiously.

For small applications, it may be possible to move directly from the requirement specification to implementation using a non-procedural fourth generation languages (4GL).

Advantages

- The use of 4GT has broadened considerably over the past decade and is now a viable approach for many different application areas.
- The time required to produce software is greatly reduced for small and intermediate applications and that the amount of design and analysis for small applications is also reduced.

Disadvantages

- Not much easier to use as compared to programming languages.
- The maintainability of large software systems built using 4GT is open to question.

3.8 Comparison of Software Process Models

Table 3.1

Strengths	Weaknesses	Types of Projects
Waterfall • Simple • Easy to execute • Intuitive and logical	• All or nothing approach • Requirements frozen early • Disallows changes • Cycle time too long • May choose outdated hardware technology • User feedback not allowed • Encourages req. bloating	For well understood problems, short duration project, automation of existing manual systems
Prototyping • Helps in requirements elicitation • Reduces risk • Leads to a better system	• Front heavy process • Possibly higher cost • Disallows later changes	Systems with novice users When There are uncertainties in requirements When UI is very important
Iterative • Regular/quick deliveries • Reduces risk • Accommodates changes • Allows user feedback • Allows reasonable exit points • Avoids req. bloating • Prioritizes requirements	• Each iteration can have planning overhead • Cost may increase as work done in one iteration may have to be undone later • System architecture and structure may suffer as frequent changes are made	For businesses where time is of essence Where risk of a long project cannot be taken Where requirements are not known and will be known only with time

3.9 Summary

Evolutionary Models take the concept of "evolution" into the engineering paradigm. Therefore Evolutionary Models are iterative. There is growing recognition that software, like all complex systems, evolves over a period of time.

The Incremental Model - The incremental model combines elements of the linear sequential model with the iterative philosophy of prototyping. The model is designed, implemented and tested incrementally that is, functionality is delivered in small increments and added till the product is completed.

The Spiral Model - Software is developed in a series of incremental releases. During initial iterations, the incremental release might be a paper model or prototype. The spiral model is a software development process which combines elements of design and prototyping, to unite advantages of top-down and bottom-up concepts. The spiral model is a realistic approach to the development of large scale systems and software.

The Component Assembly Model - This model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an interactive approach to the creation of software. However, this model composes applications from prepackaged software components. This model leads to software reuse and reusability provides number of benefits to software engineers.

The Concurrent Development Model - The concurrent model is often more appropriate for system engineering projects where different engineering teams are involved. The concurrent process model defines a series of events that will trigger transitions from state to state for each of software engineering activities, actions or tasks. The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project.

The Formal Methods Model - This model encompasses a set of activities that lead to mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation.

Fourth Generation Techniques model - The 4GT paradigm for software engineer focuses on the ability to specify software using specialized language forms or a graphic notation that describes the problems to be solved in terms that the customers can understand. The 4GT approach uses various tools for the automatic generation of source codes. It is the vital tool which uses the non-procedural language for Report generation, Database query, Manipulation of data, Interaction of screen, Definition, Generation of code, Spread Sheet capabilities, High level graphical capacity etc.

3.10 Self Assessment Questions

1. Explain various evolutionary software models.
2. Explain the advantages and disadvantages of spiral model.
3. What do you mean by fourth generation techniques ?
4. Explain formal methods model.
5. Compare various software process models.

3.11 References

- Roger, S. Pressman, " Software Engineering-A Practitioner's Approach", Third Edition, McGraw Hill.
- R.E. Fairley, 'Software Engineering Concepts", McGraw Hill.
- Jalota Pankaj "An Integrated Approach to Software Engineering", Narosa Publishing House.

Unit - 4 : Software Project Management

Structure of the unit

4.0 Objectives

After going through this unit student will be able to:

- To understand the project management concept
- Understand various software management activities
- Understand about project management approaches.
- Understand about project management life cycle.

4.1 Introduction

Project Management uses a systematic and disciplined approach to develop software. Project Management activities include: determining the scope of the project, project planning, implementation of project components in a timely manner, review of project activities, and a listing of lessons learned during the project. These activities follow the typical life cycle of a software project.

A project has specific start and completion date to create a product or service. Processes and operations are permanent functional work for repetitively producing the same product or service. Main job of project management is to achieve project goals and objectives while considering the project constraints. Constraints could be scope, time and budget. The challenge is to optimize the allocation and integration of inputs.

Software project management is a sub-discipline of project management in which software projects are planned, monitored and controlled.

A software development process or software life cycle or software process is a structure imposed on the development of a software product. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. A software development process is concerned primarily with the production aspect of software development. These processes mainly support the management of software development, and are generally addressing the business concerns.

Requirements analysis is an important part of the software engineering process, in which software developers identify the needs or requirements of a client, and depending upon the requirements they design a solution.

Risk management is the process of measuring risk and then developing strategies to manage the risk. The strategies used are transferring the risk to other party, avoiding risk, reducing negative effect of risk, and accepting some or all consequences of a particular risk.

Software project management is an essential part of software engineering. Software managers are responsible for planning and scheduling project development. They ensure that the work is being carried out as per the required standards and monitor progress to check that the development is on time and within the budget.

Software project management is concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software.

Project management is needed because software development is always the focus to budget and schedule constraints that are set by the organisation developing the software.

4.2 Software management activities

- Project planning and scheduling
- Project cost estimation and Project monitoring
- Personnel selection and evaluation
- Report writing and presentations

4.2.1 Project planning, monitoring and control

Software project management begins with set of activities which are collectively known as project planning. The function of project planning is to identify the scope of the project, estimate the work involved, and create a project schedule. Project planning begins with requirements that define the software to be developed. The project plan is then developed to describe the tasks that will lead to conclusion.

The idea of project monitoring and control is to keep the team and management updated on the project's development. If the project deviates from the plan, then the project manager can take action to correct the problem. Project monitoring and control involves status meetings. Software project planning encompasses estimation of project cost, estimation of effort, how many resources will be used and how much time it will take to build a specific software system or product. Software managers and software engineers use the information collected from the clients and metrics data collected from the old projects.

4.2.2 Project Cost Estimation

Estimation starts with the product description. The problem is decomposed into smaller problems and estimation is done using the past historical data and experience. Making estimation of cost, schedule and resources need experience and access to past historical data. Problem complexity and risk are also considered before making final estimate.

Project complexity has strong effect on planning. The experience of the similar work decreases the complexity. With the growth of project size the interdependency among the software elements increases. Experience is really useful at the design or code level but difficult to implement during software planning. Project size can affect the precision and usefulness of estimates.

4.2.3 Problems in software projects

The problems in software projects can be faced by developers, project managers as well as customers/clients.

- Problems that "developers" come across are due to lack of knowledge about developing standards, lack of updated documentations, deadlines, and changes of application necessities.

- Problems that "project managers" come across are because of inappropriate definition of roles, lack of estimation and planning, lack of decision-making. They also face constraints with schedule, budget and quality.
- Problems that "clients" come across are financial constraints due to receiving products after the due date.

4.2.4 Project Estimation

Estimation is the approximate calculation made of a result that is in working condition even if input data is incomplete or doubtful. The ability to estimate the time and/or cost taken for a project is severe problem for software engineers. Sampling more frequently and minimizing the constraints between parts of a project, helps in making more accurate estimation and more rapid development.

Popular methods for estimation in software engineering include:

- Parametric Estimating
- COCOMO
- SEER-SEM Parametric Estimation of Effort, Schedule, Cost, Risk. Mimimum time and staffing concepts based on Brooks's law
- Function Point Analysis
- Proxy-based estimating (PROBE) (from the Personal Software Process)
- The Planning Game (from Extreme Programming)
- Analysis Effort method
- Program Evaluation and Review Technique (PERT)
- True Planning Software Model Parametric model that estimates the scope, cost, effort and schedule for software projects.
- Evidence-based Scheduling Refinement of typical agile estimating techniques using minimal measurement and total time accounting.

Project planning is part of project management; it uses Gantt charts to plan and report progress within the project. Initially, the project scope is defined and suitable methods for completing the project are determined. Then, duration for various tasks are planned and grouped into a working arrangement. The logical dependencies between tasks are defined using activity network diagram that facilitates identification of the critical path. Slack time in the schedule can be calculated using project management software. Then necessary resources are estimated and cost for each activity can be allocated to each resource, giving the total project cost. Once the plan is established and agreed it is known as the baseline. Progress will be measured against the **baseline** during the life of the project. Analyzing the progress compared to the baseline is known as **earned value management.**

4.3 Project management approaches

To manage project activities including agile, interactive, incremental, and phased approaches, several approaches are used.

The traditional approach

In this approach a series of steps are to be completed. There are five components of a project in the development of a project.

- Project initiation
- Project planning or design
- Project execution or production
- Project monitoring and controlling systems
- Project completion

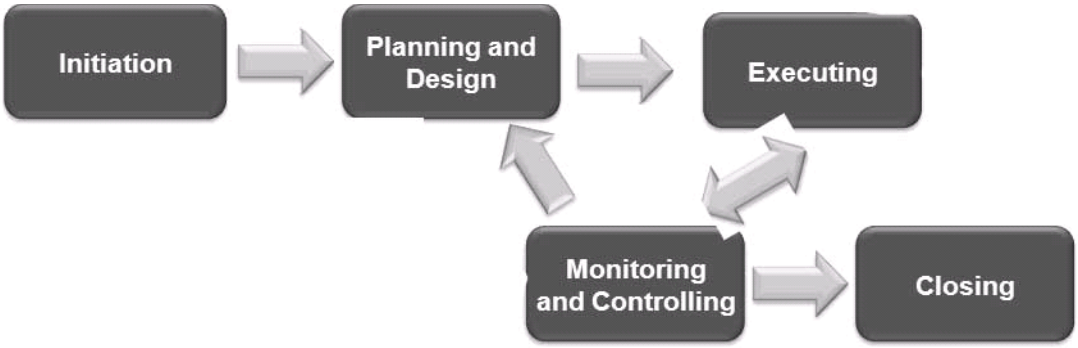


Fig. 4.1 Development Phases of A Project

4.3.1 Critical Chain Project Management

Critical Chain Project Management (CCPM) is a method of planning and managing projects, which emphasizes on the resources required to carry out project tasks. Aim is to increase the rate of throughput of projects in an organization. The system constraint for all projects is identified as resources. To utilize the constraint, tasks on the critical chain are given priority over all other activities. Finally, projects are planned and managed to ensure that the critical chain tasks are ready to start as soon as the required resources are available, sub-ordinating all other resources to the critical chain.

4.3.2 Extreme Project Management

In vital studies of Project Management, it has been noted that in today's situation the PERT-based models are not well suited for the multi-project company environment. Most of them are aimed at very large-scale, one-time, non-routine projects, and now a days all kinds of management are expressed in terms of projects.

4.3.3 Event chain methodology

Event chain methodology is an uncertainty modeling and schedule network analysis technique that is focused on identifying and managing events and event chains that affect project schedules. This methodology helps to lessen the negative impact of psychological heuristics which allows for easy modeling of uncertainties in the project schedules.

4.3.4 PRINCE2

PRINCE2 is a structured approach to project management, released in 1996 as a generic project management method. It provides a method for managing projects within a defined framework. It explains procedures to coordinate people and activities in a project, how to design and supervise the project, and what to do if the project has to be adjusted if it doesn't develop as planned.

4.4 Process-based management

The incorporation of process-based management has been driven by the use of Maturity models such as the CMMI (Capability Maturity Model Integration) and ISO/IEC15504 (SPICE - Software Process Improvement and Capability Determination).

Agile Project Management approaches are based on the principles of human interaction Management is founded on a process view of human collaboration. It is in sharp contrast with conventional approach. In the agile software development or flexible product development approach, the project is seen as a series of relatively small tasks developed and executed as the situation demands in an adaptive manner, rather than as a completely pre-planned process.

4.5 Rational Unified Process

The Rational Unified Process (RUP) is an iterative software development process framework created by the Rational Software Corporation, a division of IBM since 2003. It is an adaptable process framework, planned to be customized by the development organizations and software project teams that select the elements of the process that are appropriate for their needs. The following are phases

of RUP, which align to business activities to successful delivery and deployment projects. It also provides the classification for blue printing and producing enterprise architecture artifacts across different domains.

4.6 Project development stages or Project Management Life Cycle

Traditionally project development includes various elements: few stages, and a control system. Regardless of the methodology used, the project development process has the following major stages:

- Initiation
- Planning
- Execution [Production]
- Monitoring and Controlling
- Closure

4.6.1 Initiation

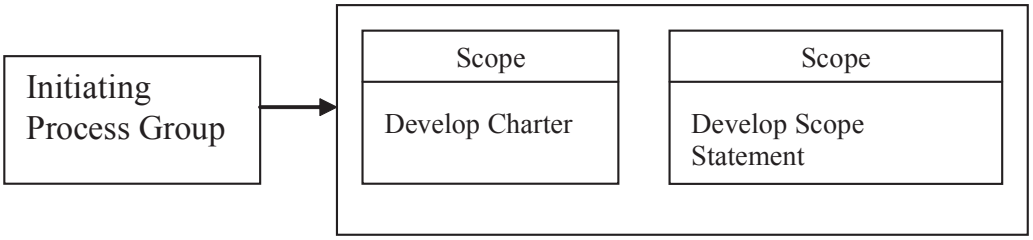


Fig. 4.2 Initiating Process Group Processes.

This stage determines the nature and scope of the development. If this stage is not performed well, then the project will be unsuccessful in meeting the business's needs. The main work here is to understand the business environment and make sure that all necessary controls are incorporated into the project. Any deficit should be reported and a proposal should be made to fix them.

The initiation stage should include a cohesive plan that includes the following areas:

- Study and analyzing the industry needs
- Review of current operations
- Conceptual design of the operation of the final product
- Equipment and contracting requirements including assessment of items
- Financial analysis of the costs and benefits including a budget
- Stakeholder analysis, including users, and support staff for the project
- Project costs, tasks, deliverables, and schedule

4.6.2 Planning and design

After the initiation stage, the system design takes place. Testing is generally performed by the testers and end users, and can take place after the prototype is built or concurrently. Controls should be in place that ensures that the final product will meet the specifications of the project. The output of the design stage should include a design which :

- Satisfies the end users and business requirements
- Works as planned and Produced as per quality standards
- Produced as per time and budget constraints

4.6.3 Execution

It consists of the processes, which are used to complete the work defined in the project management plan and it involves coordinating people and resources, as well as integrating and performing the activities of the project in accordance with the project management plan.

4.6.4 Monitoring and Controlling

It consists of processes, which are performed to examine project execution so that likely problems can be identified timely and corrective action can be taken, to control the execution of the project. Benefit is that project performance is observed and measured regularly to identify inconsistency from the project management plan.

Monitoring and Controlling includes :-

- Measuring the current project activities
- Monitoring the project variables (cost, effort, etc.)
- Identifying corrective actions to tackle issues and risks
- Implementing only approved changes

Project Maintenance includes:-

- Nonstop support of end users
- Correction of errors
- Updating software with time regularly

4.6.5 Closure

Closing includes the formal acceptance of the project followed by the ending thereof. Activities include the archiving of the files and documenting. Closing phase consists of two parts:

- Close project: to complete all the activities to formally close the project or a project phase
- Contract closure: for completing and settling each contract, closing each contract applicable to the project or a project phase.

4.7 Project control systems ?

Project control keeps track of the system that it is being made as required, is on time and within budget. It begins with planning and ends with review after implementation. Each project is assessed for the appropriate level of control needed: too much control is too time-consuming and too little control is very uncertain.

Auditors or program inspectors should review the development process and procedures for how they are implemented. The process of development and the quality of the final product may also be assessed if needed or requested. A business may want the auditing firm or program inspectors to be involved throughout the process to grab problems earlier so that they can be fixed more easily. Sometimes businesses use formal systems development processes, which assure that systems are developed successfully. Outlines for a good formal systems development plan includes :

- Strategy to align development with the organization's objectives
- Standards for new systems
- Project management policies for timing and budgeting
- Procedures describing the process

4.8 Summary

Software project management is said to be an umbrella activity. The main element in software projects is people. Before any technical activity starts it takes place first it and continues throughout the various stages of definition, development and support activities. Software engineers are organized in different team structures. The work is supported by various coordination and communication techniques. Project management activities include metrics, estimation, risk analysis, schedules, tracking and control.

The process of development and the quality of the final product may also be assessed if needed or requested. A business may want the auditing firm or program inspectors to be involved throughout the process to grab problems earlier so that they can be fixed more easily.

4.9 Self Assessment Questions

1. What is a software project management?
2. Explain Software management activities.
3. Explain various project management approaches.
4. Compare various project management approaches.
5. Differentiate between process and product.
6. What is Process-based management? Explain.
7. Describe the Critical Chain Project Management.
8. Explain Extreme Project Management.
9. Explain briefly Project development stages or Project Management Life Cycle.
10. What is the importance of planning and design? Explain.

4.10 References

- Introduction to Software Engineering - www.newagepublishers.com/
- Software Engineering By K.K. Aggarwal
- Bauer, Fritz et al., "Software Engineering: A Report on a Conference Sponsored by NATO Science Committee", NATO, 1968
- Schach, Stephen, "Software Engineering", Vanderbilt University, 1990
- Bertrand Meyer, *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice-Hall, 1994.
- Martin Fowler with Kendall Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley Object Technology Series, 1997
- D. L. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, 15(12):1053-1058, December 1972.
- John M. Vlissides and James O. Coplien and Norman L. Kerth, eds., *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.
- Sommerville, I (2000), Software Engineering, 6th Edition, Pearson Education Ltd., New Delhi
- Roger S. Pressman, Software Engineering : A Practioners Approach., 6th Edition, The McGraw Hill Companies, New York

Unit - 5 : Software Project Metrics and Software Project Planning

Structure of the Unit

5.0 Objective

Planning of the software project includes all aspects related to the Cost, Schedule and Quality of a software system and its development process. In this chapter we will look at a variety of issues concerned with Software Measurement, Software Project Planning and Identification of available Resources.

5.1 Introduction

Measurements can be categorized in two ways: 'direct' and 'indirect' measures. *Direct measures* of the product include cost and effort applied. It includes lines of code (LOC) produced, execution speed, memory size, and defects reported over pre-specified period of time. *Indirect measures* of the product include quality, functionality, complexity, efficiency, reliability, maintainability, etc.

The cost and effort required in building a software and the number of lines of code produced are relatively easy to measure, but, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

5.2 Software Measurement

5.2.1 Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and productivity measures by considering the *size* of the software product. If a software development organization maintain records in the form of a table that contain size-oriented measures, organization have to create a table as shown in Figure 5.1. The table lists each software development project that has been completed over the past few years and corresponding measures for that project.

As per the table entry (table 5.1) for project **Alpha:** 11,999 lines of code were developed with 23 person-months of effort at a cost of $160,000. It should be notified that the effort and cost recorded in the table represent all software engineering activities such as analysis, design, code, and test, not just coding.

Table 5.1: Size Oriented Metrics

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
Alpha	11,999	23	160	360	120	24	3
Beta	26,900	60	422	1222	320	83	4
Gamma.....	20,120	42	311	1040	254	63	5

Rest information for project Alpha indicates that 360 pages of documentation were developed, 120 errors were recorded before the software was released, and 24 defects were encountered after release to the customer within the first year of operation. Three people worked on the development of software for the aforesaid project. In order to develop metrics that can be incorporated with similar metrics from other projects, we choose lines of code as our normalization value. From the basic data contained in the table, a set of simple size-oriented metrics can be developed for each project:

(a) Errors per KLOC (thousand lines of code).

(b) Defects per KLOC.

(c) $ per LOC.

(d) Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

(a) Errors per person-month.

(b) LOC per person-month.

(c) $ per page of documentation.

5.2.2 Function-Oriented Metrics

This type of metrics uses a measure of the functionality delivered by the application as a normalization value. As 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed as a measure called the 'Function Point'. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity. Function points are computed by completing the given table. (table no. 5.2)

Five information domain characteristics are determined and counts are provided in the appropriate columns of the table. 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. Output refers to reports and screens, error messages, etc.
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.

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.

Table 5.2: Computing function points

Measurement parameter	Count		Weighting factor				
			Simple	Average	Complex		
Number of user inputs		x	3	4	6	=	
Number of user outputs		x	4	5	7	=	
Number of user inquiries		x	3	4	6	=	
Number of files		x	7	10	15	=	
Number of external interfaces		x	5	7	10	=	
Count total							

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

FP = count total x [0.65 + 0.01 $\times$ sum [Fi]]

Where count total is the sum of all FP entries obtained from the above table.

The Fi (i = 1 to 14) are ***'complexity adjustment values'*** based on responses to the questions like,

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 re-usable?
12. Are conversions and installations 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 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 SLOC, as a way to normalize measures for software productivity, quality, and other attributes:

(a) Errors and defects per Function Point.

(b) Cost per Function Point.

(c) Pages of documentation per Function Point.

(d) Function Point per person-month.

5.3 Project Planning

For a successful project proper management and technical skilled staff is essential. Lack of only one can cause a project to fail. Proper management control and checkpoints are required for effective monitoring of the project. Controlling over development, ensuring quality, satisfying the needs of the selected process model- all requires careful management of the project.

5.3.1 Objective

Productivity and quality oriented software design cannot be accomplished without adequate project planning. A realistic project plan must be developed at the beginning of a project. It must be monitored during the project and modified, if necessary. Ultimately, a project plan is a vehicle and focal point that enables the project manager to carry out all sorts of project management activities. It provides a roadmap for a successful software project.

During Planning, all activities that management needs to perform are planned, while during Project Control, the plan is executed and updated. Without a proper plan, no real monitoring/controlling over the project is possible. The main goal of planning is to look into the future, identify the activities that need to be done to complete the project successfully and plan the scheduling and resource allocation for these activities.

5.3.2 Software Scope

The very first activity in software project planning is the determination of software scope. Function as well as performance allocated to software during system engineering should be assessed to establish a project scope that is understandable at the technical as well as management levels. A statement of software scope must be *bounded*. The *Software scope* describes the data and control to be processed, function, interfaces, reliability, performance and constraints etc. Functions when described in the statement of scope are evaluated and refined to provide more detail information prior to the beginning of estimation. Some degree of decomposition is useful because cost and schedule estimates are functionally oriented. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

Project management is an essential part of software development. A large software development project involves many people working together for a long period. To meet the cost, quality, and schedule objectives, resources have to be properly allocated to each activity for the project, and progress of different activities has to be monitored and corrective actions taken, if needed. For a large project, a proper management process is essential for success. The Management Process component specifies all activities that need to be done by the Project Management to ensure low cost and high quality objectives. The Management Process can be grouped into three main categories:

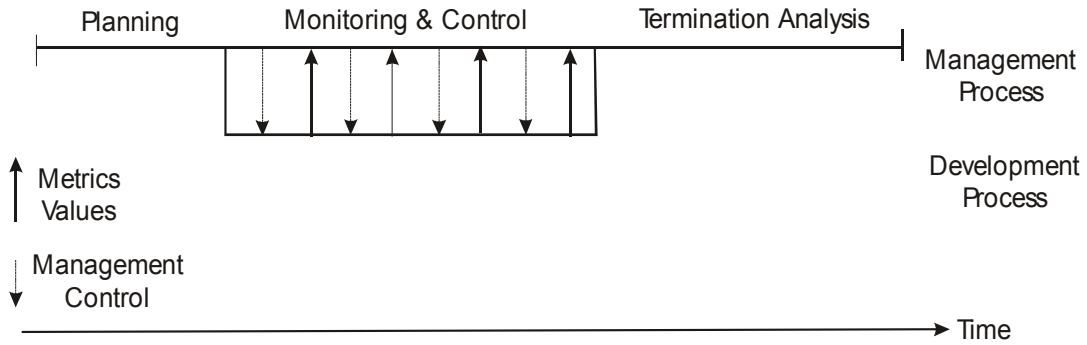


Figure 5.1: Relationship between development process and management process

Project management begins with planning. The goal of this phase is to develop a plan for software development, following which the objectives of the project can be met successfully and efficiently. Proper planning is essential for a successful project. The project plan provides the fundamental basis for project management. A software plan is usually produced before the development activity begins and is updated as

development proceeds and data about progress of the project becomes available. During planning, the major activities are 'cost estimation', 'schedule and milestone determination', 'staffing', 'quality control plans, and 'controlling and monitoring plans'.

In cost and schedule estimation, the total cost and time needed for successfully completing the project are estimated. In addition, cost and schedule for the different activities of the development process to be used are also estimated, as the development process only specifies the activities, not the cost and time requirement for them. A plan also provides methods for handling change and methods for monitoring a project. Project planning is undoubtedly the single most important management activity, and output of this forms the basis of monitoring and control.

The basic template to be used is derived from IEEE Standard 1058-1998, IEEE Standard for Software Project Management Plans. The SPMP begins with a cover page that contains the version control and release information. Each section has a description of the information contained within. The document produced in this phase is the '**Software Project Management Plan'** (SPMP), which describes the following –

1. Cost Estimation;
2. Schedule Estimation;
3. Personnel/Staff/Manpower Plan;
4. Software Quality Assurance Plan;
5. Configuration Management Plan;
6. Risk Management Plan; and
7. Monitoring Plan, etc.

5.3.3 Resources

The second software planning task is estimation of the resources required to accomplish the software development effort. Figure 5.2 illustrated pyramid shows the Software Development Resources.

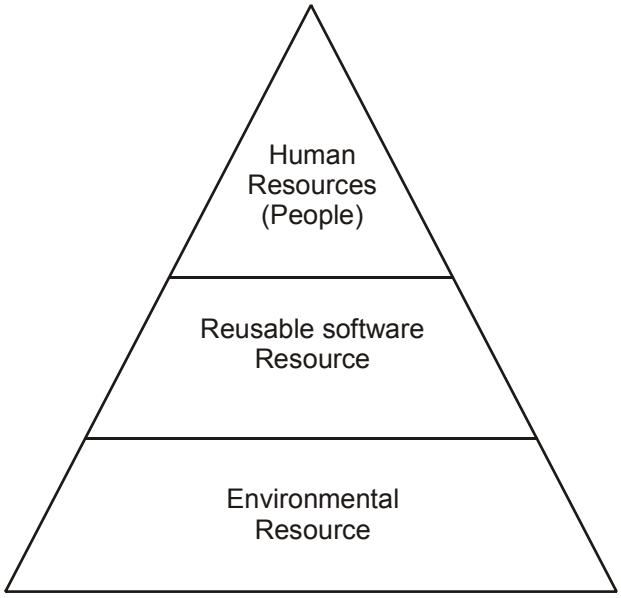


Figure 5.2: Software Project Resources

In the *Development Environment,* the hardware and software tools, sits at the bottom of the resources pyramid and provide the foundation for infrastructure to support the software development effort.

R*eusable* S*oftware Components* are basically software building blocks which can dramatically reduce development costs and accelerate delivery. These Reusable Software Components resides at the middle of the Pyramid.

At the top of the pyramid is the primary resource *'Human (People)'*.

Each resource is specified with four characteristics:

1. Description of the resource,
2. A statement of availability,
3. Time when the resource will be required; and
4. Duration of time that resource will be applied.

The last two characteristics can also be viewed as a time window.

(a) Human Resources

The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and speciality (e.g., telecommunications, database, client/server) are specified. For small projects (one person per year or less), a single individual may perform all software engineering tasks, consulting with specialists as per the requirement of the porject. The number of people required for a software project can be determined only after an estimate of development effort (e.g., person per months) is made.

(b) Reusable Software Resources

'Reusability' is the core concept of the Component-based software engineering (CBSE), which means the creation and reuse of software building blocks. Such building blocks, often called *components,* must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

According to Bennatan, following four software resource categories should be considered as planning proceeds:

(a) Off-the-shelf components: Existing software that can be acquired from a third party or that has been developed internally for a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

(b) Full-experience components: Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be of relatively low-risk.

(c) Partial-experience components: Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

(d) New components: Software components that must be built by the software team specifically for the needs of the current project.

The following guidelines should be considered by the software planner when reusable components are specified as a resource:

1. If off-the-shelf components meet project requirements, acquire them. The cost for acquisition and integration of off-the-shelf components will almost always be less than the cost to develop equivalent software. Risk is relatively low.

2. If full-experience components are available, the risks associated with modification and integration is generally acceptable. The project plan should reflect the use of these components.

3. If partial-experience components are available, their use for the current project must be analyzed. If extensive modification is required before the components can be properly integrated with other

elements of the software, proceed carefully risk is high. The cost to modify partial-experience components can sometimes be greater than the cost to develop new components.

Ironically, reusable software components are often neglected during planning, only to become a paramount concern during the development phase of the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.

(c) Environmental Resources

The environment that supports the software project, often called the *software engineering environment* (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice. Because most software organizations have multiple constituencies that require access to the SEE, a project planner must prescribe the time window required for hardware and software and verify that these resources will be available.

When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a numerical control (NC) used on a class of machine tools may require a specific machine tool (e.g., an NC lathe) as part of the validation test step; a software project for advanced page layout may need a digital-typesetting system at some point during development. Each hardware element must be specified by the software project planner.

5.4 Summary

Planning of the software project includes all aspects related to the Cost, Schedule and Quality of a software system and its development process. Measurements can be categorized in two ways: 'direct' and 'indirect' measures. Size-oriented software metrics are derived by normalizing quality and productivity measures by considering the *size* of the software product. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity. Proper management control and checkpoints are required for effective monitoring of the project. Controlling over development, ensuring quality, satisfying the needs of the selected process model- all requires careful management of the project. During Planning, all activities that management needs to perform are planned, while during Project Control, the plan is executed and updated. Component-based software engineering emphasizes reusability that is, the creation and reuse of software building blocks. The environment that supports the software project, often called the Software Engineering Environment (SEE), incorporates hardware and software. When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams.

5.5 Self Assessment Questions

1. What do you underst and by the measurement of Software?
2. Differentiate between Direct measures and Indirect measures.
3. Write a short note on Reusable Software Components.
4. What is 'Function Point'? Discuss with suitable example.
5. Explain 'Size-Oriented Metrics' with suitable example.
6. Define Off-the-shelf component and its importance.
7. What are the elements of Software Project Management Plan (SPMP)?

8. Define 'Software Scope' in brief.
9. Discuss the role of 'Planning' in the Software Project Development.
10. Explain 'Function Oriented Metrics' in brief.

5.6 References

- Pressman R.S., 'Software Engineering: A Practitioner's Approach', (Fifth Edition), McGraw-Hill, 2001.
- Keswani Bright, 'Software Engineering', (First Edition), Genius Publications, 2009.
- Jalote Pankaj, 'An Integrated Approach to Software Engineering', (Second Edition), Narosa Publishing House, 1997.
- Puri S. and Singh G., 'Software Engineering Vol. 1', (Fourth Edition), Genius Publications, 2008.
- Sommerville I., 'Software Engineering', (Sixth Edition), Pearson Education, 2002.
- Schach S.R., 'Software Engineering', (Seventh Edition), Tata McGraw-Hill, New Delhi, 2007.
- Mall R., 'Fundamentals of Software Engineering', (Sixth Edition), Prentice-Hall of India, 2002.
- Gill N.S., 'Software Engineering: Software Reliability, Testing and Quality Assurance', Khanna Book Publishing Co (P) Ltd, New Delhi, 2002
- Sabharwal S., 'Software Engineering: Principles, Tools and Techniques', (Second Edition), Umesh Publications, Delhi, 2005.

Unit - 6 : Software Project Estimation

Structure of the Unit

6.0 Objective

Many elements are involved in determining the structure of a project, including requirements, architecture, quality provisions and staffing. Perhaps the most important element in the success or failure of a project is the initial estimate of its scope, in terms of both the time and cost that will be required. The initial estimate drives every aspect of the project, constrains the actions that can be taken in the development or upgrade of a product, and limits available options. In this chapter we will discuss a variety of issues related with the software size, cost and schedule estimation. A very important issue i.e. Risk Management is also being discussed in this chapter.

6.1 Introduction

A software project estimate is the most knowledgeable statement that can be made at a particular point in time regarding effort, cost, schedule, and risk. A complete estimate covers definitions, uncertainties, ground rules, and assumptions. Too often an analyst is requested to prepare an estimate on short notice and with limited information. Lack of proper estimation is the main cause of the failure of any software project.

6.2 Software Decomposition and Decomposition Techniques

In today's scenario, the level of complexity of software projects is very high. The expectations of the end user are also very high. That's why the development of a software product according to the customer's specifications is a difficult task and this needs a lot of expertise in the field and a wide view towards the problem.

As software products become larger and more complex, it is increasingly more difficult to create new products based on existing software products and to service these products as demanded by market and competitive needs. An operating system (OS) is an example of a relatively large, complex software product. The operating system manages and schedules the resources of a computer or device in which it resides, and provides various function modules and interfaces that may be used by applications to accomplish various tasks. A conventional computer generally executes its operating system to manage various aspects of the computer as it is running. For example, the operating system is typically responsible for managing access to storage devices as well as input and/or output devices, and controlling the execution of one or more additional applications. Installation usually occurs before the computer executes the operating system.

A conventional operating system has a large number of files (e.g., thousands) for instructions and/or data. Such instructions, when executed by the computer, provide the operating system's functionality. In addition to the large number of files, the operating system usually has a large number of dependencies among files. For instance, many files may require the installation of one or more other files for their intended functionality to be carried out. Although the operating system's manufacturer may know of such dependencies at the time of installation, it can be difficult for a user, administrator, or other software developer to learn about these dependencies. This can prove troublesome, as software developers may not know what data or instructions in other files affect the particular file(s) with which they are concerned. It becomes more difficult for developers and others to develop and keep track of all functions of a software product when it is bigger and more complicated.

So, the **'Software Decomposition'** is usually defined as breaking up a system into its component parts. One can then design (or purchase), build, test, and deploy these separately.

A componentization schema representing files from which a software product is built. According to the schema, a component object represents one or more of the files. The component object has a manifest that identifies the component and specifies any dependencies between the component and other objects. Grouping objects according to the dependencies specified in manifests permits building the software product. A feature object defined by at least one component object represents a feature of the software product and a product object defined by at least one feature object represents the software product.

The componentization architecture defines the concepts, component repository, and programming model for managing components both during design-time and run-time. There are several strategic benefits of componentization including agility, speed, supportability, and increased revenue opportunities.

Decomposition techniques are used to estimate the cost and effort by decomposing the complete project problem into subsequent parts. It is a good method to be used, because sometimes it is quite not possible to evaluate the estimation of the software estimation at once. So, if the problem to be solved is too complicated, it is subdivided into parts. So, first these subdivisions are solved one-by-one and after that their solutions can be combined to form a whole.

The software decomposition techniques are used to estimate the size, cost and effort by following ways,

a. Software Sizing;
b. Problem based estimation and;
c. Process- based estimation.

6.2.1 Software Sizing

Without software, computer hardware is useless. Software development or purchase of software has a major share of the computing budget of any organization. So, software systems are very important and valuable for not only developers but also its users.

Single software has many attributes which one can measure, like,

1. The Size (Line of Code);
2. The Cost of Development;
3. The cost of Maintenance;
4. The time for development (in person-months);
5. The size of memory required (in bytes); and so on.

For the measurement of above factors, it is quite obvious that different observers of the same computer program may get different results. For example, to measure the line of code of a computer program, one observer may count all the lines present in the program including blank lines and comments; where another observer may not count blank lines and comment lines appear in the program. So, different persons get different count.

Here, a standard and precise definition and guideline of the line of code metric is required so that different persons may get identical counts for the same program.

From time-to-time different metrics have been developed to quantify various attributes of software. Here, we can grouped all of them into two broad categories –

1. ***Product Metrics*** –As earlier stated, the software metrics, like Size, which can be derived from the software itself, are called 'Product Metrics'.
2. ***Process Metrics*** – All those measurements of a software product which depend upon the development environment are called 'Process Metrics'.

Computer Programs were developed for solving different problem and having different objectives. So, they are written in different languages. Some programs are of good quality because they have good documentation and written with the use of latest tools and techniques of Software Engineering. While others having no planning, comments and documentation. Despite all these, all have size and it is one common feature which all programs share.

Measurement of size is a very important metric for software industry and it has some characteristics like –

1. It is very simple and easily calculated when the program is completed.
2. Cost and effort estimation is based on the size estimation so it plays an important role in a software development project.
3. Memory requirement can be decided on the basis of size measurement.
4. Software productivity can be also predicted by the estimation of size.

The software size depends upon the following factors –

1. The degree to which the size is to estimated,
2. The ability to translate these estimates into effort, time and cost, and
3. Necessity of stability of the requirements and the environment in which project is built.

Some of the size measures are –

a. *Source Line of Code* (LOC based estimation)
b. *Token Count*
c. *Function Point Metric* (FP based estimation)

a). Source Line of Code (LOC based estimation)

SLOC is a software metric used to measure the amount of code in a program. SLOC is typically used to estimate the amount of effort that will be required to develop a program as well as to quantify productivity or effort once the software is produced. One of the problems with using SLOC has been the lack of a standardized deûnition. Physical SLOC measures (counting physical line endings like lines on a page) are sensitive to the formatting and style conventions of the language used to develop the code. Logical SLOC measures the number of 'statements'. Logical SLOC deûnitions are tied to particular computer languages. Logical SLOC is less sensitive to formatting and style conventions, but is sensitive to programming language and technology. SLOC measures are often stated without a deûnition, which constitutes a problem because the SLOC represents a count of unknown units. Additionally, SLOC that are hand-generated should be separately identiûed from auto-generated SLOC, and new SLOC must be separated from reused SLOC. Despite these problems, SLOC counts have been and will continue to be successfully used by many companies, organizations, and projects as units of software size.

In other words, we can say that this is the simplest metric to calculating the size of a computer program. The most striking feature of this metric is its precise definition. There is a general agreement among researchers that SLOC measure should not include documentation and blank lines also because blank lines are also a part of internal documentation of the program. Their absence does not affect the efficiency and functionality of the program. According to this measure, only executable statements should be included in the count because only they support the functions of the program.

This metric still continues to be popular and useful in software industry because of its simplicity.

The SLOC measure gives an equal weight to each line of code. But, infect some statements of a program are more difficult to code and comprehend than other.

This is the major drawback in SLOC size measure of treating all lines alike can be solved by giving more weight to those lines, which are difficult to code and have more 'stuff".

b). Token Count

'Token Count' measure is the solution of the drawback of SLOC size measure. According to the Token Count – Count the basic symbols used in a line instead of lines themselves. These basic symbols are called 'Tokens'. And according to the 'Halstead Theory', a computer program is considered to be a collection of tokens, which may be classified as either 'operators' or 'operands'.

An operator can be defined as a symbol or a keyword which specifies an action like arithmetic, relational, special symbols, punctuation marks, reserve words (e.g. Do, READ, WHILE etc.) and function names (e.g. printf() etc.).

All Software Science metrics can be defined in terms of these basic symbols. The basic measures are –

n_1 = Count of unique operators.

n_2 = Count of unique operands.

N_1 = Count of total occurrence of operators.

N_2 = Count of total occurrence of operands.

A token which receives the action and is used to represent the data is called an 'operand'. An 'operand' includes variables, constants or labels.

In terms of the total tokens used, the size of the program can be expressed as

$$N = N_{1+} N_2$$

c). Function Point Metric (FP based estimation)

In a large and complex software product, size can be estimated in a better way though a larger unit called 'Module'. A 'Module' is a segment of code which may be compiled separately and independently.

In a large and complex project it is easier to predict the number of modules than the lines of code. For example, if we require n modules and assume that each module have 90-100 line of code; then the size estimate of this product is about n x 100 SLOC. But, this metric requires a strict rule for dividing a program into modules. Due to the absence of a state forward rule, this metric may not be so useful.

A module may consist of many functions and a function may be defined as a group of executable statements which performs a defined task. The number of SLOC for a function should not be very large. So, when we count the number of functions in a program, we can easily estimate the size of the computer program.

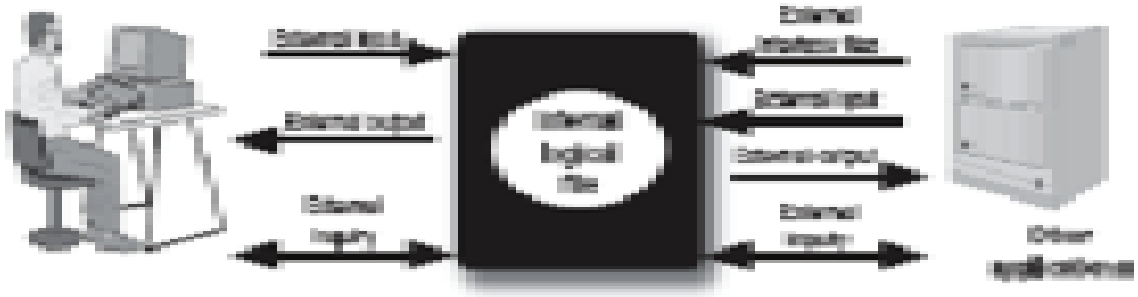
Function points measure delivered functionality in a way that is independent of the technology used to develop the system. Function points compute size by counting functional components (inputs, outputs, external interfaces, files, and inquiries). As illustrated in Figure given below, function points project what will be provided by the system rather than how big the end product will be.

This approach is similar to projecting the cost of a car by anticipating what capabilities it will have rather than by its size. When the size of a proposed software product is projected from requirements, the project has taken the first essential step toward estimating the cost and time that will be needed for its development. The problem of project management, like that of most management, is to find an acceptable balance among time, cost, and performance. Schedule performance is most important due to customer pressures, so cost and product performance lose emphasis.

Often the product takes center stage due to a customer review so cost and schedule performance focus drifts out of the shadows. What was once well controlled now becomes less well managed, resulting in risk.

The means by which software projects establish such balance is through the size projections used to estimate the work, develop the schedule, and monitor the changing size projections, which alter the balance point. If these projections are flawed, either through poor application of process, incorrect use of historical information, or management malpractice, this essential balance cannot be reached.

Table given below, illustrates some advantages and disadvantages of SLOC counts and function points.



Function Point is a unit of measure

- Functionality as viewed from the user's perspective
- A user is one who writes the system requirements, not just a software operator

Figure 6.1: Function Points – a user's perspective

Advantages and Disadvantages of SLOC and Function Point Counts

	Advantages	**Disadvantages**
Lines of Code	Easy to Compute Intuitive	No Incentive to optimize code Multiple definitions for what constitutes a line of code; autogenerated code may artificially inflate a SLOC count.
	Granular Natural by-product of process	Varies dramatically based on technology used Difficult to relate to requirements early in the project
		Some languages difficult to count automatically.
Function Point	Measures system from user perspective	User perspective must be available for detailed count
	Better defined counting rules	Not all information available early in project
	Relatively independent of Technology	Not intuitive to managers
	Maintains link to Functionality	Counters must be trained

6.2.2 Problem Based Estimation

During software project estimation the SLOC and FP data are used in two ways –

1. As an estimation variable to 'size' each element of the software
2. As baseline metrics collected from various past projects and used in combination with estimation variables to develop cost and effort projections.

Here, SLOC and FP both are different estimation techniques but both have a number of characteristics in common. The project planner identify the statement of software scope and from this statement attempts to decompose software into problem functions that can each be estimated individually. SLOC or FP is then estimated for each function. Alternatively, the planner may choose another component for sizing.

Baseline productivity metrics like SLOC per Person Month 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. 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. So, it is always remember that projects should be grouped by the application area, team size, complexity of the application and other relevant parameters. Local domain averages should then be computed. The SLOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning.

- When SLOC is used as the estimation variable, composition is absolutely essential and is often taken to considerable levels of detail.
- For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics- inputs, outputs, data files, inquires, and external interfaces etc. The resultant estimates can then be used to derive a FP value that can be tied to past data and used to generate an estimate.

The project planner begins by estimating a range of values for each function or information domain value. Using historical data, the planner estimates an optimistic 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 points or expected values can then be computed by the project planner. The expected value for the estimation variable (size):S, can be computed as a weighted average of the optimistic (S_{opt}), most likely (S_m), and pessimistic (S_{pess}) estimates. For example,

$$S = (S_{opt} + 4S_m + S_{pess}) / 6$$

Gives heaviest weight to the 'most likely' estimate and follows a beta probability distribution. We assume that there is a vary small probability the actual size result will fall outside the optimistic or pessimistic values. The historical SLOC and FP productivity data can be applied after the determination of the expected value for the estimation variable. It is suggested that any estimation technique must be cross checked with another approach.

6.2.3 Process Based Estimation

To base the estimate on the process that will be used is the most common technique for estimating the project. In the process based estimation technique, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated. Process-based estimation begins with a description of software functions obtained from the project scope and a series of software process activities must be performed for each function.

Once problem functions and process activities are decided, the planner estimates the effort in person-months that will be required to accomplish each software process activity for each software function. Functions and related software process activities may be represented as part of a table -6.1 as given below. These data constitute the central matrix of the table. Average labour rates or cost per unit

efforts are then applied to the effort estimated for each process activity. It is very likely that 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. The table is given below-

Table 6.1 Process Based Estimation

Activity	Costomer communication	Planning	Risk Analysis	Engineering		Construction		Customer evaluation	Total
Task →				Analysis	Design	Code	Test		
Function ↓									
UICF				.50	2.50	.40	5.00	n/a	8.40
2DGA				.75	4.00	.60	2.00	n/a	7.35
3DGA				.50	4.00	1.00	3.00	n/a	8.50
CGDF				.50	3.00	1.00	1.50	n/a	6.00
DBM				.50	3.00	.75	1.50	n/a	5.75
PCF				.25	2.00	.50	1.50	n/a	4.25
DAM				.50	2.00	.50	2.00	n/a	5.00
Total	0.25	0.25	0.25	3.50	20.50	40.5	16.50	n/a	46.00
%Effort	1%	1%	1%	8%	45%	10%	36%		

UICF: User Interface and Control Facilities **2DGA:** Two Dimensional Geometric Analysis
3DGA: Three Dimensional Geometric Analysis **DBM:** Data Base Management
CGDF: Computer Graphics Display Facilities **PCF:** Peripheral Control Function
DAM: Design Analysis Module

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. Otherwise, the results of these decomposition techniques show little agreement; further investigation and analysis must be conducted by the estimator.

6.2.4 COCOMO (COnstructive COst estimation MOdel) – An Empirical Estimation Model for Effort

The Constructive Cost Model (COCOMO) was developed by 'Boehm' in 1981 and is based on study of 63 projects. It estimates the total effort in terms of 'person-months' of the technical project staff. The effort estimate includes 'development', 'management', and 'support tasks'.

The estimation through this model starts with to obtain an initial estimate of the development effort from the estimate of thousands of delivered lines of source code (KDLOC). After this step determine a set of 15 multiplying factors from different attributes of the project; and then adjust the effort estimate by multiplying the initial estimate with all the multiplying factors.

According to the COCOMO, Software projects are categorized into three types –

1. ***Organic :*** Project area in which the organization has considerable experience and requirements are less. Such systems usually developed by a small team. For example, 'Simple business system'.
2. ***Embedded :*** The organization has little experience. System is highly affected from the environment like software, people or hardware. For example, Real time systems.
3. ***Semidetached :*** This is the combination of both systems. For example, developing a new operating system and a simple business management system.

The basic features of these three types of project are shown in the given table-

Table 6.2 Types of Project

	Organic	**Embedded**	**Semidetached**
Project Size	Small	Any Size	Large
Type of Project	Independent	Product embedded in environment with several constraints	Independent
Development Experience	Extensive	Moderate	Moderate
Environment Knowledge	Extensive	Normal	Considerable
Constant Values	a = 2.4, b=1.05	a = 3.6, b=1.20	a = 3.0, b=1.12

The 'initial estimate' which is also called 'nominal estimate' is determined by an equation, using KDLOC as the measurement of size. To determine the effort E in person-months, the following equation is used -

$$\mathbf{E = a * (SIZE)\ b}$$

Where, E = effort in 'person-month'
a, b = Constants, values depends on the type of the project
SIZE = in KDLOC

So, effort formulas for three types of project are-

Organic : $E = 2.4 * (KDLOC)^{1.05}$
Embedded : $E = 3.6 * (KDLOC)^{1.20}$
Semidetached : $E = 3.0 * (KDLOC)^{1.12}$

According to the 'Intermediate COCOMO' and 'Detailed COCOMO', Boehm proposed fifteen different attributes which are used to measure cost called 'Cost driven attributes'. All these factors depend on '*Product attributes*', '*Computer attributes*', '*Personne*l *attributes*" and *'Project attributes*' attributes.

The 'Cost Driven Attributes' are shown below,

Product Attributes	Software Reliability Size of application database Complexity
Computer Attributes	Performance Requirements Memory Constraints Volatility of Virtual Machine Environment Turn Around Time
Personnel Attributes	Analyst Capability Software Engineering Capability Applications Experience Virtual Machine Experience Programming Language Expertise
Project Attributes	Use of Software Tools Application of Software Engineering Methods Required Development Schedule

Each of these cost drivers for a given project is rated at a scale:

- Very Low
- High
- Very High
- Extra High

The scale describes the proportion in which cost driver applies to a project. Values corresponding to each of the six ratings are also defined. The multiplying factors for all 15 cost drivers are multiplied to get the 'Effort Adjustment Factor' (EAF). The 'Final Effort Estimate' (E) is obtained by multiplying the initial estimate by the EAF:

$$\mathbf{E = a\ (KDLOC)^{b} * EAF}$$

Boehm also proposed standard development time (D) based on the project type and project size in months. For different types of projects, D is computed using formulas given as under –

For 'Organic Project' : $D = 2.5 * (E)^{0.38}$

For 'Semidetached' : $D = 2.5 * (E)^{0.35}$

For 'Embedded' : $D = 2.5 * (E)^{0.32}$

Exercise -

The values of size in KDLOC and different cost drivers for a project are given below-

Size = 200 KDLOC

Cost driver:

Software Reliability: 1.15

Use of Software Tools: 0.91

Product Complexity: 0.85

Execution Time Constraint: 1.00

Calculate the effort for three types of projects i.e. Organic, Semidetached, and Embedded using COCOMO Model.

Solution –

Effort Adjustment Factor (EAF) is:

EAF = 1.15 * 0.91*0.85*1.00 = 0.8895

Effort for three types of projects is:

Organic Project

$E = 3.2 * (200)^{1.05} * 0.8895 = 742$ person months

Semidetached Project

$E = 3.0 * (200)^{1.12} * 0.8895 = 1012$ person months

Embedded Project

$E = 2.8 * (200)^{1.2} * 0.8895 = 1437$ person months

6.3 Risk Management

Software Engineering is the technical and managerial discipline and it is concerned with the systematic invention, production, and maintenance of high quality software products delivered on time, at minimum cost. The global IT software industry stands to lose billions each year in project overruns, and reworking software. According to survey reports, as much as 40% of software development costs alone can be spent reworking software. Furthermore, on average only one in six software projects will be delivered on time and to budget, and in some large organizations only 10% of their projects will come in on time and to budget.

Given figure outlines some major causes of project failure.

Managerial Issues - account for 65 percent of failure

- Inappropriate project structure and channels of communication
- Inappropriate resources (poor skill and knowledge mix)
- Inappropriate estimates
- Inappropriate planning methods (poor scheduling and tracking)
- Inappropriate user buy-in
- Inappropriate risk management

Technical Issues - account for 35 per cent of failure

- Inappropriate software requirements
- Inappropriate technical design
- Inappropriate development and testing tools
- Inappropriate technical documentation
- Inappropriate technical support
- Inappropriate technical reviews (and quality audits)

Figure 6.2: Major causes of project failure

The term software risk management can mean different things to many different people, in considering what is meant by risk we need to understand the qualitative, quantitative and philosophical aspects of what constitutes risk. There are numerous definitions on what constitutes risk; an appropriate definition is as follows:

Risk is a combination of an abnormal event or failure and the consequences of that event or failure to a system's operators, users or environment. A risk can range from catastrophic (loss of an entire system; loss of life or permanent disability) to negligible (no system damage or injury). (Glutch, 1994)

If not examined at the outset the threats listed below, usually end up downstream as major causes of project failure.

Business	Technological
• Pricing strategy	• Production limitations
• Legal restrictions (litigation)	• Design compatibility
• Potential for copying	• Production limitations
• Payments and royalties	• Degree of innovation
• Potential for customer participation	• Proprietary technology
• Timing of introduction	• Patents (s)
• Stakeholder influence	• Technology uniqueness

Figure 6.3: Some threats to industry-wide software projects

Regardless of size and scope, every project is threatened by risk. Software projects are especially susceptible to risk due to the speed of development in the field and its rapidly changing environment. 'Barry Boehm' identified a 'Top-10' list of major software development areas where risk must be addressed:

1. Skill shortfalls
2. Unrealistic schedules and budgets
3. Stability of external or off-the-shelf software
4. Requirements mismatch
5. User interface mismatch

6. Architecture, performance, quality
7. Requirement changes
8. Legacy software
9. Externally performed tasks
10. Straining computer science capabilities.

According to 'McManus' other key risk area includes the following-

1. Infrastructure limitations
2. Procurement of infrastructure
3. Multi-site software development
4. Untested development methodologies
5. Regulatory standards (for example, health and safety)
6. Inadequate test strategy
7. Development team involved in multiple projects or other activities
8. Communication problems
9. Availability of test sites or personnel.

Software development is a very complex activity. The software problem has numerous elements with extremely complicated interrelationships. Problem element relationships can be multidimensional. The laws of proportionality do not govern changes in elements. It is well documented that adding more people to a project that is behind schedule, in many instances, will make it even later. Software problem elements are unstable and changeable. Although cost and schedule may be fixed, actual costs in labour and time to complete are difficult to project. The development process is dynamic. Conditions ceaselessly change; thus, project equilibrium is rarely achieved. The environment is never static – hardware malfunctions, personnel quit, and contractors do not deliver. People are an essential software development element and a major source of risk.

Additionally, there are other interrelated factors that contribute to software risk. These factors are:

- Communication about risk is one of the most difficult, yet important. People do not want to talk about potential problems. Effective risk planning only occurs when people are willing to talk about risks in a non-threatening, constructive environment.
- Software size can affect the accuracy and efficacy of estimates. Interdependence amongst software elements increases exponentially as size increases. With extremely large software systems, handling complexity through decomposition becomes increasingly difficult because even decomposed elements may be unmanageable.
- Software architecture also affects software risk. Architectural structure is the ease with which functions can be modularized and the hierarchical nature of information to be processed. It is also development team structure, its relationship with the user and to one another, and the ease with which the human structure can develop the software architecture.

Identification and prioritization of risks enable project managers and project staff to focus on the areas with the most impact to their project. Appropriate risk mitigation actions reduce overall project risk thus accelerating project completion. Projects that finish sooner cost less, plus risk mitigation actions can further reduce project cost. Projects using software risk management have more predictable schedules; they experience fewer surprises, since they have identified risks before they can become problems. Risk management doesn't necessarily mean avoiding projects that could incur a high level of risk. Preparing a detailed risk assessment for a software project is costly and time consuming; however, in the long run the benefits that accrue normally outweigh the cost involved. Software risk management experts agree that the costs associated with taking a few preventative measures early on are negligible when compared to the dramatic costs that can be incurred when proper risk management techniques are neglected.

6.4 Summary

Cost and Schedule estimation are the fundamental requirements to the project management. A software project estimate is the most knowledgeable statement that can be made at a particular point in time regarding effort, cost, schedule, and risk. A complete estimate covers definitions, uncertainties, ground rules, and assumptions. The 'Software Decomposition' is usually defined as breaking up a system into its component parts. There are two main software decomposition techniques- 1). Object Oriented Programming and 2). Component-based Development (CBD) or Software Componentry. The software decomposition techniques are used to estimate the size, cost and effort by three main different ways – a). Software Sizing; b). Problem based estimation and c). Process- based estimation. SLOC is a software metric used to measure the amount of code in a program. SLOC is used to estimate the amount of effort that will be required to develop a program as well as to quantify productivity or effort once the software is produced. 'Token Count' measure is the solution of the drawback of SLOC size measure. Function points measure delivered functionality in a way that is independent of the technology used to develop the system. Function points compute size by counting functional components (inputs, outputs, external interfaces, files, and inquiries). The Constructive Cost Model (COCOMO) was developed by 'Boehm' in 1981. It estimates the total effort in terms of 'person-months' of the technical project staff. The effort estimate includes 'development', 'management', and 'support tasks'.

6.5 Self Assessment Questions

1. Discuss the usefulness of 'Software Decomposition'.
2. What are the decomposition techniques? Discuss problem based estimation and process based estimation in brief.
3. Explain the 'SLOC' method of software size measurement.
4. Give your critical views, whether SLOC based estimation is an effective method or not.
5. What is 'COCOMO'? Describe the various modes of COCOMO estimation model given examples of application falling in each area.

6.6 References

- Pressman R.S., 'Software Engineering: A Practitioner's Approach', (Fifth Edition), McGraw-Hill, 2001.
- Keswani Bright, 'Software Engineering', (First Edition), Genius Publications, 2009.
- Jalote Pankaj, 'An Integrated Approach to Software Engineering', (Second Edition), Narosa Publishing House, 1997.
- Puri S. and Singh G., 'Software Engineering Vol. 1', (Fourth Edition), Genius Publications, 2008.
- Sommerville I., 'Software Engineering', (Sixth Edition), Pearson Education, 2002.
- Schach S.R., 'Software Engineering', (Seventh Edition), Tata McGraw-Hill, New Delhi, 2007.
- Mall R., 'Fundamentals of Software Engineering', (Sixth Edition), Prentice-Hall of India, 2002.
- Gill N.S., 'Software Engineering: Software Reliability, Testing and Quality Assurance', Khanna Book Publishing Co (P) Ltd, New Delhi, 2002
- Sabharwal S., 'Software Engineering: Principles, Tools and Techniques', (Second Edition), Umesh Publications, Delhi, 2005.

Unit - 7 : Software Requirement Specification

Structure of the Unit

7.0 Objective

It is a reality that a missed or bad requirement can result into a software error which requires a lot of rework and wastage of resources. It can have also financial and legal implications. So, understanding requirements of a project is most important and crucial to that project and forms foundation of that project. In this chapter we will discussed about the variety of software requirements, their considerations and the entire process of requirement engineering.

7.1 Introduction

The systematic process of documenting requirements through an interactive cooperative process of analyzing a problem is called 'Requirement Engineering'. There are various methodologies like Natural Language descriptions, structured analysis techniques, object oriented requirements analysis techniques etc. have been proposed by researchers for expressing requirements specification.

7.2 The 'Software Requirement'

According to the IEEE, '*Requirement* is a condition of capability, needed by a user to solve a problem or achieve an objective'.

Broadly requirements can be classified as following –

1. Functional Requirements;
2. Non-Functional Requirements; and
3. Domain Requirements.

7.2.1 Functional Requirements

Functional requirements are the services which the end users expect the final product to provide. These requirements focus on the functionality of the software components that build the system. As these components would be doing some kind of transformation on input, functional requirements are expected in terms of inputs, outputs and processing.

Natural language, structured or formatted languages with no rigorous syntax etc. are some ways of expressing such type of requirements.

7.2.2 Non-Functional Requirements

Non-Functional requirements are the constraints imposed on the system and deal with issues like Security, Maintainability, Performance, Reliability etc. and play a very important role in deciding the destiny of the software project. Non-Functional Requirements (NFRs) sometimes known as 'Quality Attributes' or 'Constraints' or 'Goals' or 'Non-behavioral Requirements' etc.

According to (Roman85), Non-Functional Requirements can be classified as-

a) Economic constraints
b) Interface constraints
c) Operating constraints
d) Performance constraints like response time, reliability, storage space, security etc.
e) Life cycle constraints like reusability, portability, maintainability etc.

Sommerville, divided NFRs into three different groups-

a) Process Requirements
b) Product Requirements
c) External Requirements

7.2.3 Domain Requirements

Some requirements are specific to an application domain e.g. hospital, banking, marketing etc. these requirements are known as 'Domain Requirements'. These requirements are not user specific and can be identified from that domain model. Also, these are mandatory requirements to be implemented for a smooth and satisfactory working of the system. For example, in a hospital there will always be requirements concerning Doctors, Staff, Departments and Patients etc.

7.2.4 User Requirements

Some requirements are in the form of sstatements in natural language plus diagrams of the services the system provides and its operational constraints. These requirements are basically written for customers.

In other words, the user requirements are written from the user's point-of-view. User requirements define the information or material that is input into the business process, and the expected information or material as outcome from interacting with the business process (system), specific to accomplish the user's business goal.

7.2.5 System Requirements

To meet out System Requirements, a structured document setting out detailed descriptions of the system services. It is written as a contract between client and contractor.

7.3 Feasibility Study

Preliminary investigations examine the feasibility of the project, that the proposed project will be useful to the organization or not. Following are the main feasibility tests:

(a) Operational Feasibility
(b) Technical Feasibility
(c) Economic Feasibility

(a) Operational Feasibility of the Project

Proposed projects are beneficial only if they can be turned into information systems that will meet the operating requirements of the organisation. This test of feasibility asks if the system will work when it is developed and installed. Some of the important questions that are useful to test the operational feasibility of a project are:

- Is there sufficient support for the project from the management and from users?
- Are current methods acceptable to the users? If they are not, users may welcome a change that will bring about a more useful and convenient system.
- Have the users been involved in the planning and development of the project? If they involved at the earliest stage of project development, the chances of resistance can be possibly reduced.
- Will the proposed system produce poorer result in any case or area?
- Will the performance of staff member fall down after implementation of the proposed project?

Issues that appear to be quite minor at the early stage can grow into major problem after implementation. Therefore, it is always advisable to consider operational aspects carefully.

(b) Technical Feasibility of the Project

Following are some of technical issues which are generally raised during the feasibility stage of the investigation:

- Does the necessary technology available or can be acquired from somewhere else?
- Does the proposed equipment have the technical capacity to hold the amount of data required to use the new system?
- Can the system be upgraded if developed?
- Are there technical guarantees of accuracy, ease of access, reliability and data security?

(c) Financial Feasibility of the Project

A system that can be developed technically and that will be used if installed must be profitable for the organisation. Financial benefits must equal or exceed the costs. The analysts raise various financial questions to estimate the following:

- The cost to conduct a full systems investigation.
- The cost of hardware and software for the class of application being considered.
- The benefits in the form of reduced costs or fewer costly errors.
- The cost if nothing changes (i.e. the proposed system is not developed).

To be judged feasible, a proposal for the specific project must pass all these tests. Otherwise, it is not considered as a feasible project. It is not necessary that all projects that are submitted for evaluation and review are acceptable. In general, requests that do not pass all the feasibility tests are not pursued further, unless they are modified and re-submitted as new proposals. In some cases, it so happens that a part of a newly developed system is unworkable and the selection committee may decide to combine the workable part of the project with another feasible proposal. In still other cases, preliminary investigations produce enough new information to suggest that improvements in management and supervision, not the development of information systems, are the actual solutions to reported problems.

7.4 Requirement Engineering Process

Requirements engineering is the process of eliciting, documenting, analyzing, validating, and managing requirements. Different approaches to requirements engineering exist, some more complete than others. Whatever the approach taken, it is crucial that there is a well-defined methodology and that documentation exists for each stage. A requirement modeling involves the techniques needed to express requirements in a way that can capture user needs. A requirement modeling uses techniques that can range from high-level abstract statements through psuedocode-like specifications, formal logics, and

graphical representations. Whatever representation technique is used, the requirements engineer must always strive to gather complete, precise, and detailed specifications of system requirements. Requirements engineering is a sub-discipline of software engineering that is concerned with determining the goals, functions, and constraints of software systems.

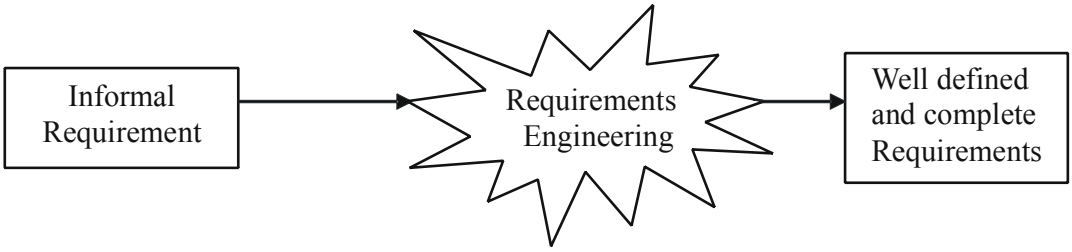


Figure 7.1 Requirement Engineering Process

Requirements engineering also involves the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families. Ideally, the requirements engineering process begins with feasibility study activity, which leads to a feasibility report. It is possible that the feasibility study may lead to a decision not to continue with the development of the software product. If the feasibility study suggests that the product should be developed, then requirements analysis can begin.

The Requirement Engineering Process consists of three main activities,

1. Requirements Elicitation;
2. Requirements Specification; and
3. Requirements Validation.

7.4.1 Requirements Elicitation

The 'Requirements Elicitation' activity is concerned with understanding of problem domain at the starting of the project because at the beginning of the project, requirements are fuzzy and are not clearly understood by the system analyst. So, the best solution is to acquire the domain specific knowledge and become expert in that.

The process of acquiring domain specific knowledge/information through various techniques to build the 'Requirements Model' is called 'Requirement Elicitation'.

The various sources of the domain knowledge can be as follows-

1. Users of the specific problem domain;
2. Existing Software of same type and Business Manuals; and
3. Standards, etc.

7.4.2 Requirement Specification

The 'Requirement Specification' activity is required to produce formal software requirements models. The 'Functional' and 'Non-Functional' properties of the system are specifying by the requirements models along with the constraints imposed on the system. These models are used in successive stages of the System Development Life Cycle (SDLC). Also, it is used as 'Agreement' between the end users and the developers. There are a number of structured and object oriented methodologies that have been proposed to develop these models.

7.4.3 Requirement Validation

This activity is usually defined as a process to ensure the constancy of requirements model with respect to user's needs. The validation is done for the final model as well as for the intermediate models. Following steps should be followed while validating requirements-

a). Ensure that the requirements are not conflicting with other requirements.

b). Ensure that the requirements are complete in all respect.

c). Ensure that requirements are realistic.

If the requirements are not validated properly, resultant errors in the requirements definition needs a lot of rework and modifications. Some of the effective ways to validate requirements are –

- **Reviews** – During reviews end users and developers both study and consider the requirements.
- **Prototyping** – In prototyping an executable model of the product with minimal set of requirements is built and is demonstrated to the customers. Customers in tern use the prototype to ensure that it meets their needs.
- **Test cases generation** – From this, it is made sure that it is possible to design test cases for each and every requirement documented otherwise there is a need to reconsider the requirement.

7.5 Key Challenges In Eliciting Requirements

The extraction and refinement of the product requirements is the most important activity of Software Life Cycle and this process faces many key challenges, some of them are given below-

a) Understanding large and complex system requirements.

b) Identifying the critical requirements.

c) Sometimes users not clear about their needs.

d) Conflicting requirements.

e) Undefined boundaries of the system.

f) Partitioning of the system.

g) Proper documentation of the requirements.

h) Meeting time and budget constraints of the customer.

7.6 Software Requirements Specification (SRS)

Requirements analysis is usually the first phase of a large-scale software development project. The purpose of this phase is to identify and document the exact requirements for the system. It is undertaken after a feasibility study has been performed to define the precise costs and benefits of a software system. In cases where the requirements are not clear, much interaction is required between the user and the developer. There are two major activities in this phase: problem understanding or analysis and requirement specification. In problem analysis, the analyst has to understand the problem and its context.

The Software Project is initiated by the client's need. In the beginning, these needs are in the minds of various people in the client organization. The requirement analyst has to identify the requirements by talking to these people and understand there needs. Analysis also required to add 'new features' when automating an existing manual process. Such analysis typically requires a thorough understanding of the existing system, parts of which have to be automated. A clear understanding is needed of the important data entities in the system, major centers where action is taken, the purpose of the different actions that are performed, and the inputs and outputs. This requires interacting with clients and end users, as well as studying the existing manuals and procedures. With the analysis of the current system, the analyst can understand the reasons for automation and what affects the automated system might have.

Understanding the existing system is usually just the starting activity in problem analysis, and it is relatively simple. The goal of this activity is to understand the requirements of the new system that is to be developed. Understanding the properties of a system that does not exist is more difficult and requires creative thinking. The problem is more complex because an automated system offers possibilities that do not exist otherwise. Consequently, even the client may not really know the needs

of the system. The analyst has to make the client aware of the new possibilities, thus helping both, client and analyst, determine the requirements for the new system.

In Software Requirement Analysis, there are two main concepts –

1. **State –** A state of a system represent some conditions about the system. When using state, a system is first viewed as operating in one of the several possible states, and then objects based analysis is performed for each state.
2. **Projection –** In projection, a system id defined from multiple points of view. While using projection, the system is analyzed from these different viewpoints. The different 'projections' obtained are combined to form the analysis for the complete system.

There is no defined methodology used in the 'Informal approach of Analysis'. The analyst will have conducted a series of meetings with the clients and end-users. They provide explanations to the analyst about their work, their environment and their needs. Any document describing the work or the organization may be given, along with outputs of the existing methods of performing the tasks. Once the analyst understands the system he conducts the next few meetings to seek clarifications of the parts he does not understand. In the final meeting, the analyst essentially explains to the client what he understands the system should do. Once the problem is analyzed and the essentials understood, the requirements must be specified in the requirement specification document. For requirement specification in the form of a document, some specification language has to be. The requirements document must specify all functional and performance requirements; the formats of inputs and outputs; and all design constraints that exist due to political, economic, environmental, and security reasons. Software Requirement Analysis (SRA) ends with a document called 'Software Requirement Specification (SRS) which describes the complete external behavior of the proposed software.

The SRS is a document that completely describes what the proposed software should do without describing how the software will do it.

The basic goal of the requirement phase is to produce the SRS, which describe the complete external behavior of the proposed software.

Need of SRS

There are three major parties, interested in a new system –

1. The Client
2. The User and
3. The Developer.

Somehow, the requirements for the system that will satisfy the needs of the clients and the concerns of the users have t be communicated to the developer. The problem is that the client usually does not understand the software and the development process, and the developer often does not understand the client's problem. This causes a communication gap between the parties involved in the development project. SRS is the medium through which the client and user needs are accurately specified; a good SRS should satisfy all the parties involved in the development process.

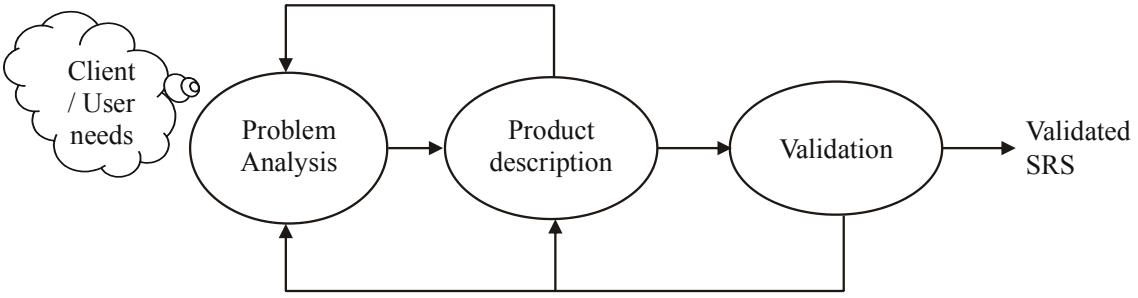


Figure 7.2: The Requirement Process

Advantages of SRS

1. SRS establishes the basis for agreement between the client and the supplier on 'what the software product will do'. Through SRS the client clearly describes that what he expects from the supplier, and the developer clearly understand that what capabilities to build in the software. Without such agreement it is almost guaranteed that once the development is over, the project will have an unhappy client and unhappy developer.
2. SRS provides a reference for validation of the final product. The SRS helps to the client to determine whether the software meets the requirements or not. Without SRS, there is no way for developer to convince the client that all requirements have been fulfilled.

Table 7.1: Cost of fixing requirement errors

Name of the phase	Cost (person-hrs)
Requirement Analysis	2
Design	5
Coding	15
Acceptance test	50
Operation and Maintenance	150

3. If we want a high quality final software that has few errors, we must begin with a high quality SRS.
4. A high quality SRS reduces the development cost. High quality means, customer's and developer's satisfaction, system validation, high quality of the final software and the reduced development cost.

7.7 Summary

'Requirement' is a condition of capability, needed by a user to solve a problem or achieve an objective. Some requirements are specific to an application domain, these requirements are known as 'Domain Requirements'. The process of acquiring domain specific knowledge/information through various techniques to build the 'Requirements Model' is called 'Requirement Elicitation'. The 'Requirements Elicitation' activity is concerned with understanding of problem domain at the starting of the project. The 'Requirement Specification' activity is required to produce formal software requirements models. The 'Requirement Validation' activity is usually defined as a process to ensure the constancy of requirements model with respect to user's needs. 'Software Requirement Specification (SRS) is a document which describes the complete external behavior of the proposed software. SRS establishes the basis for agreement between the client and the supplier on 'what the software product will do'.

7.8 Self Assessment Questions

1. What is meant by 'Requirement Engineering'? Indicate its importance.
2. Discuss different types of requirements.
3. What are non-functional requirements?
4. What do you understand by requirement elicitation?
5. Differentiate between functional and non-functional requirements.
6. What is Software Requirements Specifications (SRS) document? Outline its goals.
7. Explain and illustrate the 'Requirement Engineering Process'.
8. What are the effective ways to validate requirements?

9. What are the key challenges in 'Eliciting Requirements'?
10. Discuss about need and advantages of SRS.

7.9 References

- Pressman R.S., 'Software Engineering: A Practitioner's Approach', (Fifth Edition), McGraw-Hill, 2001.
- Keswani Bright, 'Software Engineering', (First Edition), Genius Publications, 2009.
- Jalote, Pankaj, 'An Integrated Approach to Software Engineering', (Second Edition), Narosa Publishing House, 1997.
- Puri S. and Singh G., 'Software Engineering Vol. 1', (Fourth Edition), Genius Publications, 2008.
- Sommerville I., 'Software Engineering', (Sixth Edition), Pearson Education, 2002.
- Schach S.R., 'Software Engineering', (Seventh Edition), Tata McGraw-Hill, New Delhi, 2007.
- Mall R., 'Fundamentals of Software Engineering', (Sixth Edition), Prentice-Hall of India, 2002.
- Gill N.S., 'Software Engineering: Software Reliability, Testing and Quality Assurance', Khanna Book Publishing Co (P) Ltd, New Delhi, 2002
- Sabharwal S., 'Software Engineering: Principles, Tools and Techniques', (Second Edition), Umesh Publications, Delhi, 2005.

Unit - 8 : Structured Analysis

Structure of the Unit

8.0 Objective

Structural analysis is a model building activity, being as the major in the process of analysis modeling. In this we create data, partition the data and design the functional and behavioral models which reflect the proper image of what must be built. It uses a combination of text and diagrammatic forms to depict requirements for data, function and it's behaviour, in a way that is easier to understand and much simpler to implement.

In this chapter we will discuss about Data Modeling, Functional Modeling and tools of structured analysis.

8.1 Introduction

It is the job of the software engineer to build a structural model by using the requirements as specified by the customer. Furthermore, if a model could help to examine the requirements then it is highly effective. In this sense, analysis models which represent a three dimensional view, is very much effective by increasing the probability of the error to be detected, verifying or in other sense detecting the inconsistencies and omissions, which in other models can remain uncovered. It is not a single model step, but this process has evolved over a period of 30 years, and uses the experience of various techniques that have proved to be successful over the past.

The main goal is same and that is to elaborate or clarify the requirements of the software before the process of coding begins. A system analyst is responsible to build a logical system model that could meet the needs of the requirement such that no requirement is overlooked.

8.2 Various Elements Of The Analysis Model

The analysis model must achieve three primary objectives:

- it must describe the requirements of a customer,
- it must establish a basis for the creation of a software design, and,
- it must define a set of requirements, which can be validated after the complete software is built.

To meet these objective, the analysis model derived from structural analysis takes the form as shown in fig. 8.1.

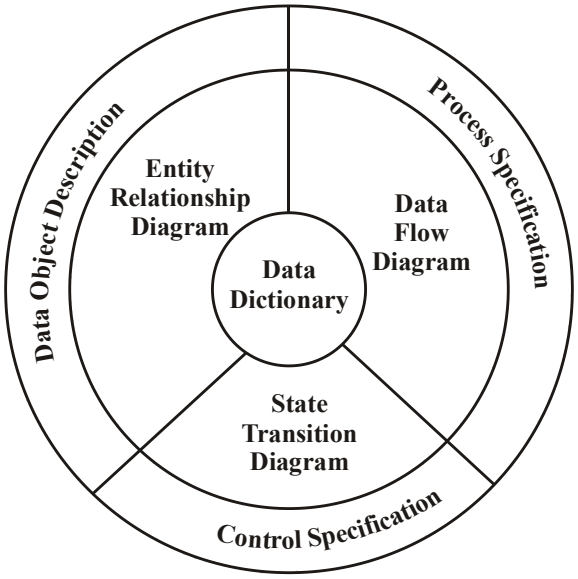


Fig. 8.1: The Structure of the Analysis Model

This analysis model consists of:

1. **Data Dictionary** - *Data Dictionary* forms the core of the model and contains the description of all the data objects that have been defined or concerned with the software. Data dictionary is surrounded by three different kinds of diagrams.
2. **Entity Relationship Diagram (ERD)** - An *Entity Relationship Diagrams* (*ERD*) describes how the various data objects are related to each other. ERD is used to conduct the data modeling activity and is concerned with defining the various attributes or properties of data objects.
3. **Data Flow Diagram (DFD)** - *Data Flow Diagrams* (*DFD*) describes both the functions, that are used to transform a data (input) into some other form (may be output), and the way the data is transformed from input to output. It serves as a basis for functional modeling.
4. **State Transition Diagram (STD) -** *State Transition Diagram* describes how the system behaves to external events. It can be a decision making step. It represents the various modes of the behaviour (called states) and the way in which system transits from one state to another.
5. **Process and Control Specification -** *Process Specification* contains the description of every functional module, defined in the DFD. *Control Specification* holds the additional information regarding the control aspects of the software, responsible for various state transition.
6. **Data Object Description -** *Data Object* description provides the complete knowledge about a data object.

8.3 Data Modeling

Data Modeling defines the primary data object, their attributes and their relationships with other primary data objects. It also defines relationships between objects and the processes that transform them. In order to define the objects and their relationship, it makes use of Entity–Relationship Diagrams (ERD).

ERD helps in defining as well as identifying them and their relationships using graphical notations. ERD defines all the data that are stored, entered, transformed and produced within an application.

Data modeling considers data independent of the processes that transforms data. Now, we will proceed to discuss the data objects, their attributes and relationships, Cardinality and Modality, Entity–Relationship Diagrams in the following sections.

8.3.1 Data Objects, Attributes and Relationships

A data model consists of three different types of interrelated information; data objects, attributes that describe an object and differentiate one object from other, and the relationships that connect data objects to one another.

- **Data objects:** By composite information, we mean something that has a number of different properties or attributes which can differentiate it from other information. Data objects represent such composite informations which must be familiar to or understood by the software. Thus, a single value is not sufficient for a valid data object. But a group of such individual values (e.g. colour, weight) can collectively form a valid data object.

  A valid data object can be of any type: a thing, an external entity, an occurrence, a role, an organizational unit, a place or a structure. These data objects are related to one another. A data object encapsulates only data. There is no references to processes or operations that acts on data within a system.

- **Attributes:** The attributes defines the properties of a data object. It also forms the basis on which two data objects can be distinguished from one another. So, attributes can be used to:
  - name an instance of a data object,
  - describe an instance, and,
  - make reference to another instance of some other data object (referencing).

  One or more attributes can be collectively used as an identifier of a particular entity of a data object. Such collective attributes are known as Keys. Referring to a data object Student, a reasonable identifier must be his/her school or college ID Number.

- **Relationships:** Every data object defined in a system are interrelated to one another (may be not to all) in some ways. Relationships are generally bidirectional, which can be either shown as both sided arrows or no arrows on both sides. For example, consider two data objects medicine and medicine store. Their relationship is shown in fig. 8.2.

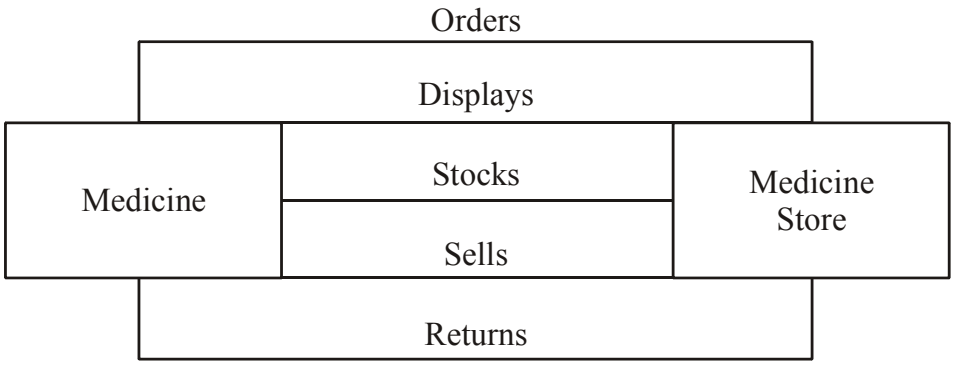


Fig. 8.2: Relationships between Objects

- A set of object/relationship pairs can be defined between these two objects:
- A medicine store orders medicine.
- A medicine store displays medicine.
- A medicine store stocks medicine.
- A medicine store sells medicine.
- A medicine store returns medicine.

8.3.2 Cardinality and Modality

Elements of data modeling such as data objects, attributes and relationships provide the basis for understanding the information domain such as what are the composite informations, its properties and how are they interrelated, but such information is not sufficient. For example, we can say that there are two data objects A and B with attributes a1, a2, a3, ...a_n and b1, b2, b3, ... b_n respectively. They are related to each other by relation R as shown in fig. 8.3.

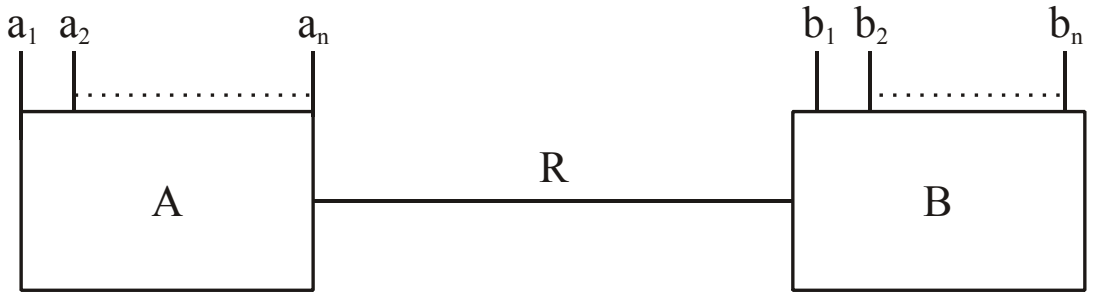


Fig. 8.3: Simple Object Relationship

This information is not sufficient. We must also know how many elements of A is related to how many elements of B by relation R. This concept is called Cardinality. There may be some cases when a defined relationship is not mandatory. This concept is called as Modality.

- **Cardinality :** Cardinality is the specification of number of occurrences of one object that can be related to number of occurrences of another object. It is usually expressed as either 'one' or many. It generally defines the maximum number of objects that can participate in a relationship. On the basis of cardinality a relationship can be one-to-one (1 : 1), one-to-many (1 : M), many-to-many (M-M).
- **Modality :** Modality of a relationship is represented by 1 if it is mandatory, 0 if it is not mandatory.

8.3.3 Entity - Relationship Diagram

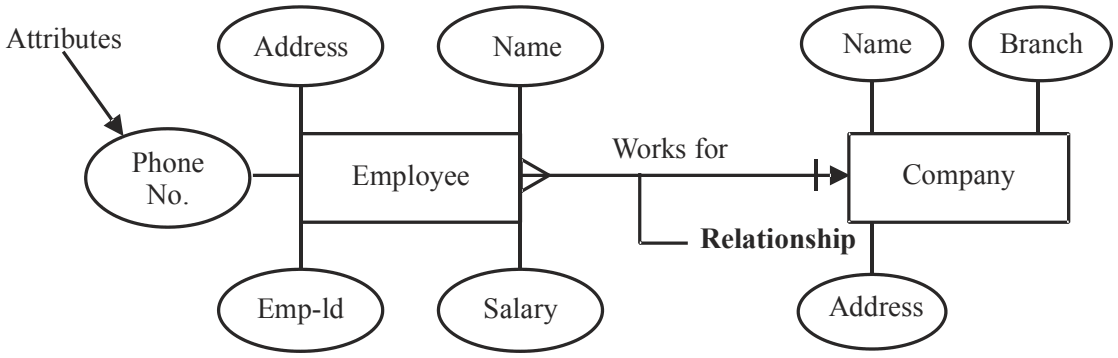
Entity-Relationship Diagram represents the object-relationship pairs in graphical form. Thus, we can say that the primary goal of E-R diagrams is to represent data objects along with their relationships. The various components of data modeling described above forms the primary components of entity-relationship diagram.

ER model for data uses three features to describe data :

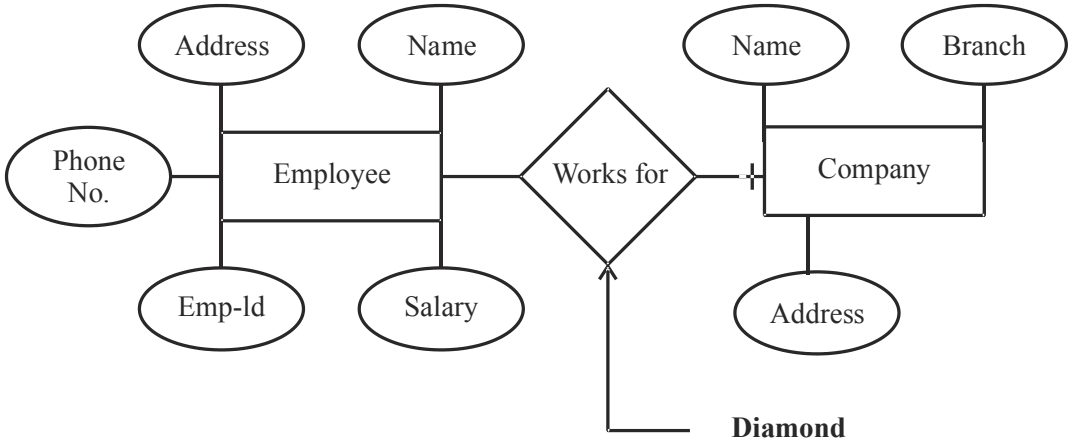
- **Entities** which specify distinct real world items in an application.
- **Relationships** connecting different entities and representing meaningful dependencies between them.
- **Attributes** which specify various properties of entities and relations involved in a system.

In ER diagram, data objects are represented by labeled rectangle. All it's attributes are represented by labeled ellipse connected to data objects by straight line. There are two ways in which relationships are represented in an ER diagram. One way is to use a labeled straight line. Another, also known as conventional way is to use a labeled diamond connected to dependent data objects by straight line.

These two ways are shown in fig 8.4, where an employee works in a company. The attributes of two entities are also shown.



(a)



(b)

Fig. 8.4 : Example of ER diagram,

(a) ER Diagram with Labeled Straight Line **(b)** ER Diagram with Labeled Diamond Connection

In addition to basic ER notations, an analyst can represent data object type hierarchy. For example, data object mobile phone can be categorised as GSM, Color display and Camera with color display, which can further be categorised. It is shown in fig. 8.5.

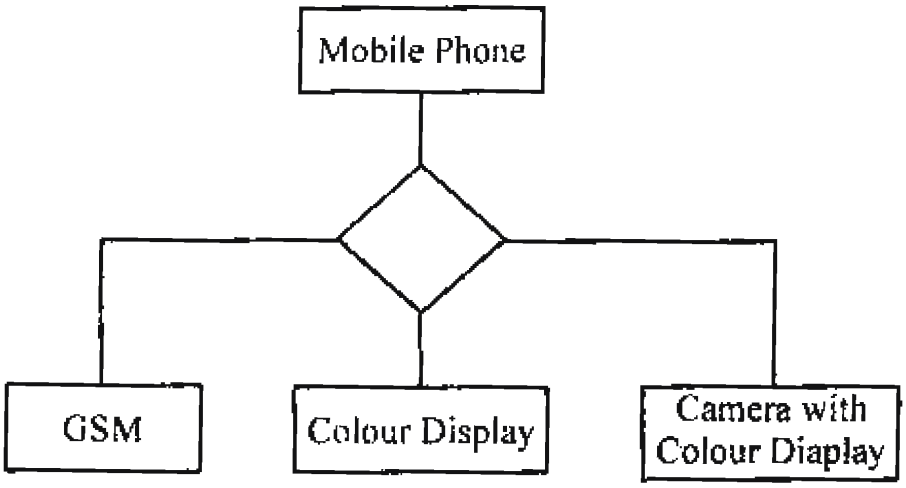


Fig. 8.5 : Data Objective type Hierarchy

The associative data is represented as shown in fig. 8.6.

Fig. 8.6: Associative Data Objects

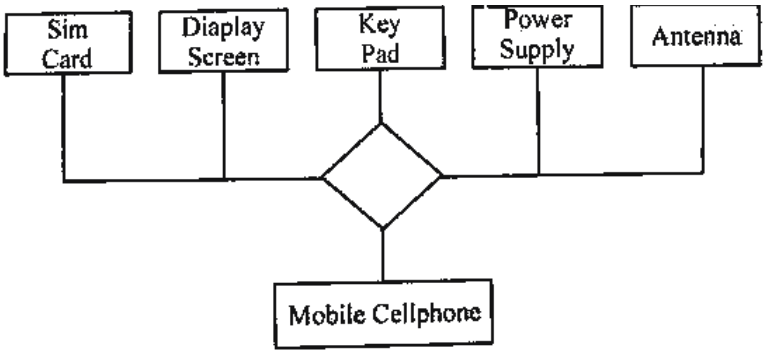


Fig. 8.6 : Associative Data Objects

ER diagram provides the most convenient way of examining data within the context of a software. To create an ER diagram, the following steps should be followed:

- Customers are asked to enlist all the primary data objects that the application or business process addresses. These things or data objects result in the generation of lists of all inputs and their corresponding output data object,
- Every object is considered at a time and its relationship (if any) with other objects is determined,
- Relationship pair is constructed wherever a connection exist,
- For every object-relationship pair, determine it's cardinality and modality,
- Once the entities along with their relationships are completely determined, define their attributes,
- Then Formalize and normalize the entity relationship diagram, and
- Repeat all the above steps till data modeling is complete.

S.No.	Symbol	Names
1	E	Entity Set
2	A	Attribute
3	E	Weak Entity Set
4	A	Multivalued Attribute
5	R	Relationship Set
6	A	Derived Attribute
7	R	Relationship Set for Weak Entity Set
8	R E	Total Participation of Entity Set
9	A	Primary Key
10	A	Discriminating Attribute of Weak Entity Set
11	R	Many–to–Many Relationship
12	R	Many–to–One Relationship
13	R	One–to–One relationship
14	R l.....h E	Cardinality Limits
15	R Role Name E	Role Indicator
16	ISA	ISA (Specialization or Generalization
17	ISA	Total Generalization

Table 8.1: Symbols to create ER Diagrams

There are various symbols and notations that are used to create an ER diagram. These symbols and their names are listed below in the table 8.1

Consider a bank system as shown in fig. 8.7, of a particular Branch, which has its attributes Bank Id, Branch Id and Manager. There is many-to-many relationship between Account Holder and Branch. The Account Holder has its attributes - Name, Account Number and Address. Whereas a Bank has many branches, leading to one-to-many relationship. It has its attributes - Name, Manager and BankId (Bank Identification Number). Now the Branch entity shows Generalization as shown in the downwards direction. It is connected to Saving Account, Current Account and Loan Account entities which have their respective attributes. Loan Account entity set shows the relationship. Paid with the weak entity set payment.

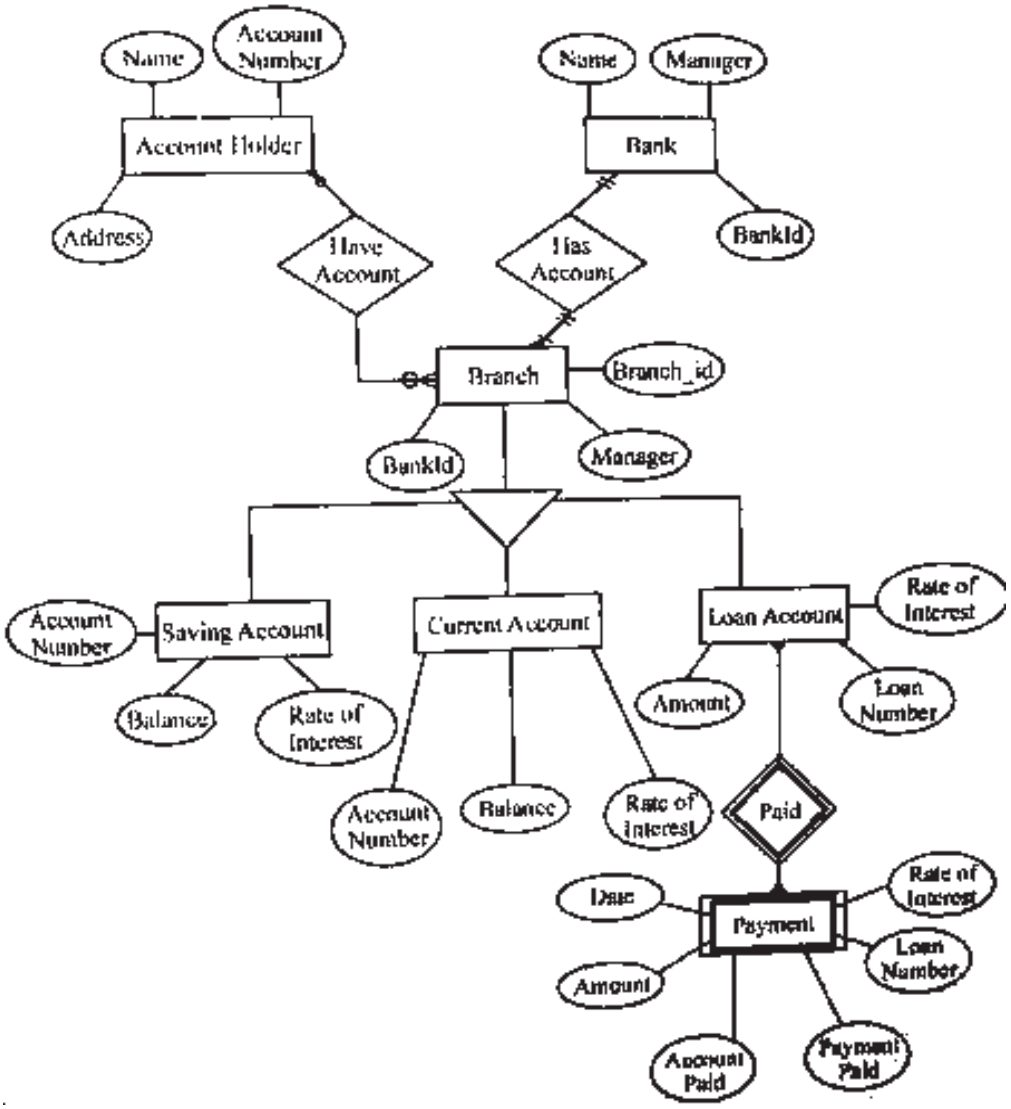


Fig. 8.7 : A Bank System

8.4 Functional Modeling

As we know, in a system, data is accepted in some form, processed in between and then produces the output in some form. In this way, the information flows through the system.

So, structured analysis began as an information flow modeling technique. In a computer-based system, the information flow is shown using some notations. A rectangle is used to represent an external entity, a circle (or bubble) represents a process or transform that is applied to data (or control) and changes it in some way. An arrow links one or more data items (or data objects). The double line represents a data store (to store the information). These notations are widely used to create the data flow diagrams.

8.4.1 Data Flow Diagram and Context Diagram

"*A perfect image can help to understand a topic more clearly*".

Diagrams must help to understand the object behind the workings of a software. A properly defined flow model diagram, not only helps to understand the working of the model, but also helps the software designer to built it and test it for validation with ease.

A Data Flow Diagram (DFD) is a graphical representation that depicts the information flow and the transforms that are applied as data moves from input to output. It can be used to represent a software at any level of abstraction. In fact, DFD's may be partitioned into levels that represent increasing information flow and functional detail. Therefore, it provides mechanism for functional modeling as well as information flow modeling.

DFD models a system using external entities (or events) which defines the type and amount of data that flows to a process, which transforms it and then creates an output, which then flows to some other process or external entities as output flow. Data may even flow from or to some secondary storage (generally described as data dictionary). The graphical representation makes it a good tool for communication between designer and analyst.

DFD's are defined in levels, with every level decreasing the level of abstraction, as well as defining a greater details of the functional organs of the system. A 0'level DFD, also known as Context or Fundamental System Model, represents the entire software elements as a single bubble, with input and output data entities which are indicated as incoming and outgoing arrows respectively.

Additional processes and information flow path are represented as separate bubble as 0'level DFD is partitioned into level 1, which is then partitioned into level 2, and so on. The levels are partitioned till a crystal clear view of the system is achieved.

Following symbols used in a DFD -

1) **Bubble:** A circle (termed bubble) is used to depict a process. Both inputs and outputs to a process are dataflows. It is shown in fig. 8.8.

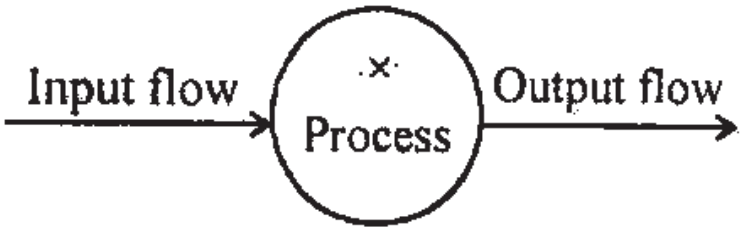


Fig. 8.8: Bubble

2) **Arrow:** Data flows are represented by a line with an arrow, illustrated in fig. 8.9.

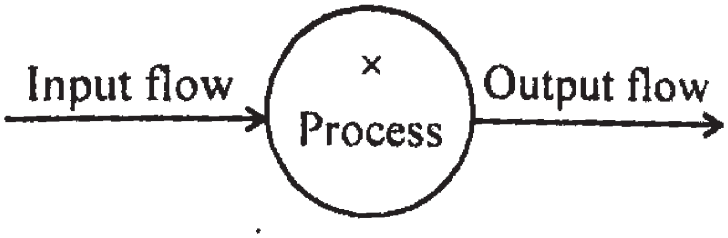


Fig. 8.9: Arrows on Line Shows Input and Output Flows

Ward and Mellor used single head arrow to represent discrete data and double headed arrow to use time continuous data (fig. 8.10).

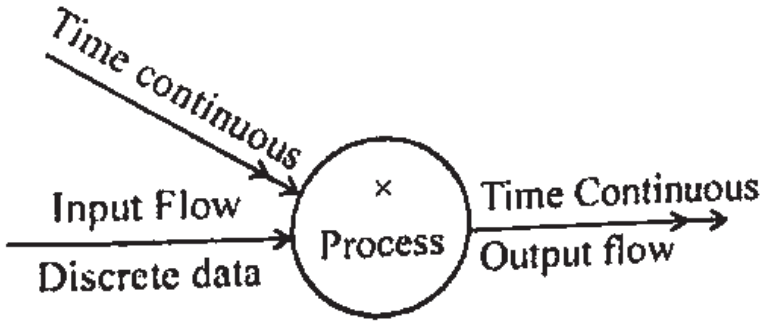


Fig. 8.10: Ward and Mellor Extension

(3) Rectangle: Rectangles are used to represent the entities and are outside the system. Designers do not have any control over them. They either supply or receives data. In the fig. 8.11, two entities supplier and receiver are shown as rectangles. So, supplier entity A supplies some stock delivered to the receiver entity B.

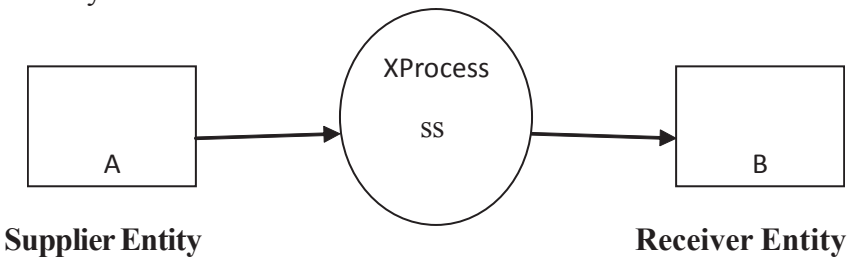


Fig. 8.11: Rectangle as an Entity

Even a single entity can both supply and receive data.

(4) Parallel Lines : Parallel lines are used to depict data stores. Process may store or receive data from data stores. Data cannot flow between two data stores. An arrow towards data store indicates writing data to data store and an arrow from data stores depicts retrieval of data from it, which is shown diagrammatically in fig. 8.12

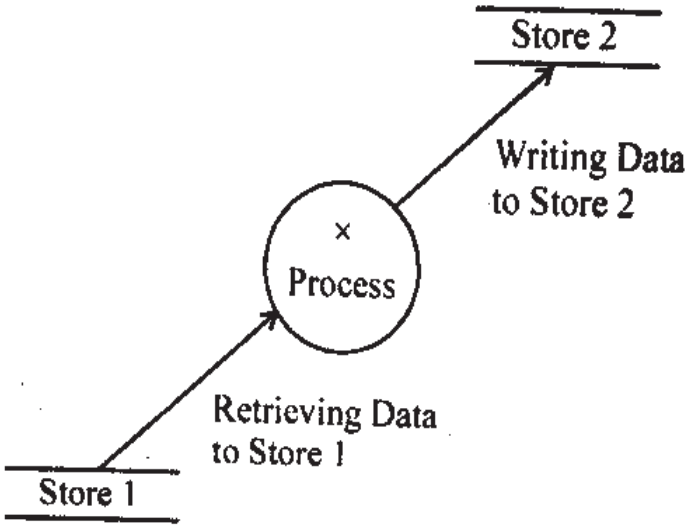


Fig. 8.12: Data Store Representation

There are some guidelines to create a data flow diagram. They are given below :

(i) Level 0 DFD should depict the entire system as single bubble along with all the related entities processing with the system.

(ii) Primary input should be carefully noted.

(iii) Refinement should begin by isolating the candidate processes, data objects and stores to be represented at the next level.

(iv) All arrows and bubbles should be named with meaningful names.

(v) Information flow continuity must be maintained at the next level.

(vi) Only one bubble should be refined at a time.

(vii) A good dataflow diagram should not possess:

a). loops,

b). a process which is taking a pure decision,

c). flow lines crossing each other, and,

d). splitting of a data flow into different flows with different meaning.

(A) Logical and Physical DFD's

All the DFD's discussed earlier represented logical process/operations that are to be performed by the system. Such DFD's are known as Logical DFD's. They specify various logical processes. Instead of logical processes, if DFD represents objects which perform the physical operations, that it is known as Physical DFD.

In fig. 8.13, the logical DFD is shown to process the order of vendor. If order is accepted then it the inventory file process is updated otherwise it is rejected.

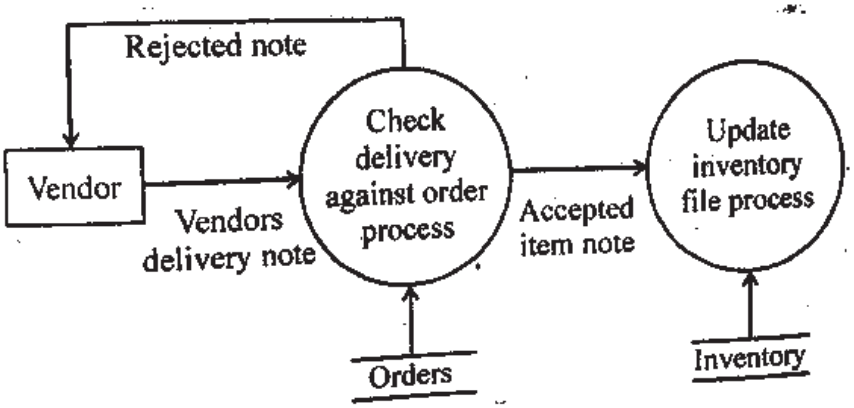


Fig. 8.13: Logical DFDFig. 6.15 : Logical DFD

In fig. 8.14, the physical DFD is shown when vendor requests for his delivery note then it is checked by the clerk, if accepted then stored otherwise if rejected then returned back.

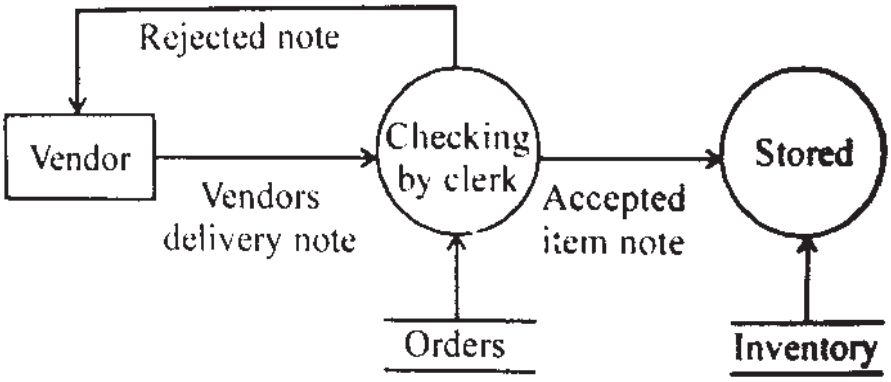


Fig. 8.14: Physical DFD

(B) Leveling of DFDs

As we know that it is easier to solve a problem in 'top-down' manner, that is starting with an overview and then working out the details. Similarly, it is easier to understand leveled DFDs as compared to single large detailed DFD.

Leveled DFDs are drawn in top-down approach, i.e. at top level 0-level DFD or context diagram is made then this 0-Level DFD is expanded to level-1 DFD from (expanding and dividing) the main process/bubble of 0-level DFD.

At level-1, the bubbles are also called as the sub-processes. To draw level-2 DFD, these sub-processes are further expanded & divided individually in 3 to 5 bubbles and so on. A detailed illustration of leveling is shown in fig. 8.15.

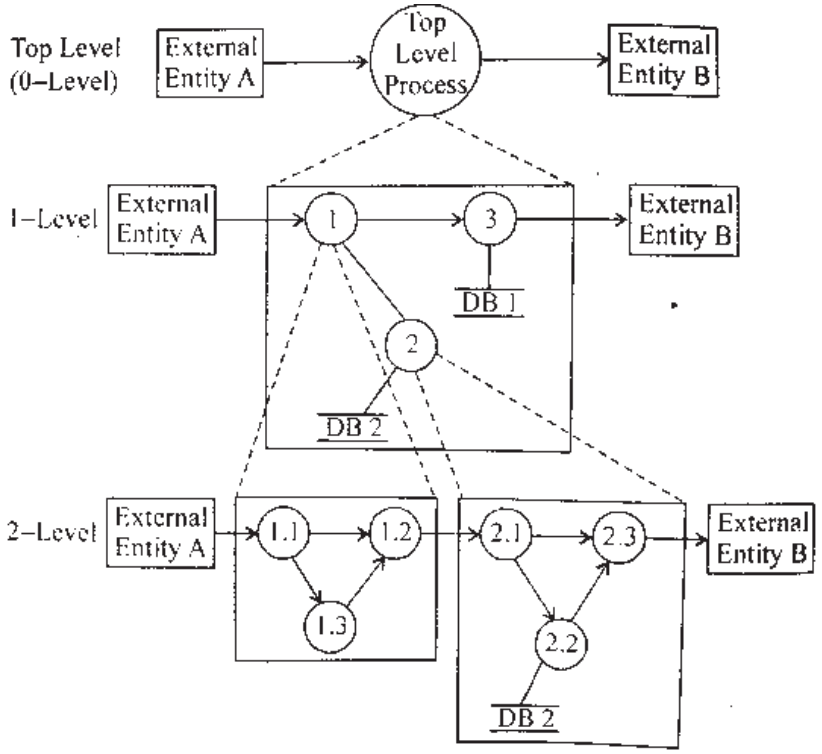


Fig. 8.15: Leveling of a DFD

8.5 Various Tools Of Structured Analysis

There are various tools of structured analysis which are discussed one-by-one in the following sections.

8.5.1 Data Flow Diagram (DFD)

Data flow diagram has been discussed in section 8.4.1.

8.5.2 Data Dictionary (DD)

Data dictionary is very important for representing the characteristics of every data object and control items that acts as the building block of analysis model. Data dictionary is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, output components of stores and intermediate calculations (temporary objects). It is difficult to maintain data dictionary manually. Hence CASE tools are used for this purpose.

8.5.3 Structured English

Structured English expresses the verbal statements of the processes in a more logical form. It uses

natural language along with the structure and block ideas like PASCAL. This technique is successful when there are not too many conditions to check in a process, otherwise it is a complex process to examine. The basic difference between structured English and PASCAL's block structure representation is the use of English. The main aim is to allow easy readability of the code of the process and thus makes it easier to check, debug and maintain. Structured English consists of following features.

1. Imperative sentences consist of imperative verb followed by action to be performed. For an example, multiply gross by discount rate, here multiply is imperative verb.
2. Arithmetic and relational operations uses common mathematical symbols instead of imperative verb like '+' for addition, '-' for subtraction, etc. Symbols used for logical operations are And, Or and Not.
3. Decision structure are of two types, one is 'if then else' which is used to make a choice, second one is for, while, case which is used for loop and selective cases respectively.

8.5.4 Decision Table

Decision table is a very helpful tool when large number of conditions are to be checked in arriving at a set of actions. It is a nonprocedural specification of decision rule. Like state table, it helps in representing which event is related to which condition. It is drawn by dividing the table into four blocks. Leftmost (top) block contains all the separate conditions. Right top block contains all the rules along with the Boolean value for conditions. Left (down) block contains the list of all the actions that is to be performed by the system. Right (down) block keeps the status of every action for every rule. An event which is to be generated when a rule is satisfied is marked '×' in that rule column. An example of decision table is shown in table 8.2. To fill the entries of conditions, true or false is used. Whereas for actions entries, '×' is made.

Table 8.2 : Decision Table

	Rule 1	Rule 2	Rule 3
Condition 1	True	False	False
Condition 2	False	True	False
Action 1	×	-	-
Action 2	-	-	×
Action 3	-	-	×
Action 4	-	×	-

When only condition 1 is satisfied, rule 1 is selected and action 1 takes place. When condition 2 satisfies action 4 takes place. Similarly when no condition is satisfied, rule 3 is selected and actions 2 and 3 occur.

An Example of Decision Table

Problem 1:

Applications for admission to an extension course are screened using the following rules,

(1) For admission, an application should be sponsored by an employer and should possess prescribed minimum academic qualification. If his fee is also paid, then he is sent a letter of admission.

(2) If his fee is not paid, the letter of provisional admission is sent.

(3) In all other cases, a regret letter is sent.

Solution 1:

The conditions to be tested are:

Cond. 1 : Is the application sponsored by an employer?

Cond. 2 : Does the applicant possess the prescribed minimum academic qualification?

Cond. 3 : Is the fee paid?

The actions to be taken are:

Action 1 : Send a letter of admission.

Action 2 : Send a letter of provisional admission.

Action 3 : Send a regret letter.

Table 8.3 is made for this problem.

Table 8.3: An Example

	Rule 1	Rule 2	Else
Condition	1	Y	Y
Condition	2	Y	Y
Condition	3	Y	N
Action 1	×	-	-
Action 2	-	×	-
Action 3	-	-	×

8.6 Summary

In Structured Analysis, we create data, partition the data and designs the functional and behavioral models which reflect the proper image of what must be built. Data Modeling defines the primary data object, their attributes and their relationships with other primary data objects.

A data model consists of three different types of interrelated information; data objects, attributes that describe an object and differentiate one object from other, and the relationships that connect data objects to one another. Cardinality is the specification of number of occurrences of one object that can be related to number of occurrences of another object.

Modality of a relationship is represented by 1 if it is mandatory, 0 if it is not mandatory. The primary goal of E-R diagrams is to represent data objects along with their relationships.

8.7 Self Assessment Questions

1. What is the Structured Analysis? Discuss various tools available for Structured Analysis.
2. Describe data modeling and functional modeling with the help of an example.
3. What is an entity? Explain with an example.
4. What is a context-level in a DFD and how it can be expanded?
5. What is a physical DFD and how it is different from logical DFD?
6. Distinguish between a data flow diagram (DFD) and a flow chart.
7. What is 'Decision Table'? Discuss its elements with suitable example.
8. Discuss the purpose of Data flow diagram with suitable examples.
9. Explain major elements of DFD.
10. What is 'Data Modeling'? Explain the Data Objects, Attributes and Relationships with suitable examples.
11. Differentiate between 'Cardinality' and 'Modality'
12. What do you understand by the term 'Leveling of DFD'? Why leveling is required?

8.8 Referemce

- Pressman R.S., 'Software Engineering: A Practitioner's Approach', (Fifth Edition), McGraw-Hill, 2001.
- Keswani Bright, 'Software Engineering', (First Edition), Genius Publications, 2009.
- Jalote, Pankaj, 'An Integrated Approach to Software Engineering', (Second Edition), Narosa Publishing House, 1997.
- Puri S. and Singh G., 'Software Engineering Vol. 1', (Fourth Edition), Genius Publications, 2008.
- Sommerville I., 'Software Engineering', (Sixth Edition), Pearson Education, 2002.
- Schach S.R., 'Software Engineering', (Seventh Edition), Tata McGraw-Hill, New Delhi, 2007.
- Mall R., 'Fundamentals of Software Engineering', (Sixth Edition), Prentice-Hall of India, 2002.
- Gill N.S., 'Software Engineering: Software Reliability, Testing and Quality Assurance', Khanna Book Publishing Co (P) Ltd, New Delhi, 2002
- Sabharwal S., 'Software Engineering: Principles, Tools and Techniques', (Second Edition), Umesh Publications, Delhi, 2005.

Unit - 9 : Software Design

Structure of the Unit

9.0 Objectives

After completing this unit you will learn

- Design concepts and principles, design tool
- Modular design,Object oriented design, real time design
- Interface design etc.

9.1 Introduction

A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces and components.

9.2 Design concepts and principles

The design process is very important. From a practical standpoint, as a labourer, one would not attempt to build a house without an approved blueprint thereby risking the structural integrity and customer satisfaction. In the same manner, the approach to building software products is no different. The emphasis in design is on quality; this phase provides us with representation of software that can be assessed for quality. Furthermore, this is the only phase in which the customer's requirements can be accurately translated into a finished software product or system. As such, software design serves as the foundation for all software engineering steps that are followed regardless of which process model is being employed. Without a proper design we risk building an unstable system – one that will fail when small changes are made, one that may be difficult to test; one whose quality cannot be assessed until late in the software process, perhaps when critical deadlines are approaching and much capital has already been invested into the product.

During the design process the software specifications are transformed into design models that describe the details of the data structures, system architecture, interface, and components. Each design product is reviewed for quality before moving to the next phase of software development. At the end of the design process a design specification document is produced. This document is composed of the design models that describe the data, architecture, interfaces and components.

At the data and architectural levels the emphasis is placed on the patterns as they relate to the application to be built. Whereas at the interface level, human ergonomics often dictate the design approach employed. Lastly, at the component level the design is concerned with a "programming approach" which leads to effective data and procedural designs.

9.2.1 Design Specification Models

- **Data design** – created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software. Part of the data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.
- **Architectural design** - defines the relationships among the major structural elements of the software, the "design patterns" than can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which the architectural patterns can be applied. It is derived from the system specification, the analysis model, and the subsystem interactions defined in the analysis model (DFD).
- **Interface design** - describes how the software elements communicate with each other, with other systems, and with human users; the data flow and control flow diagrams provide much of the necessary information required.
- **Component-level design** - created by transforming the structural elements defined by the software architecture into procedural descriptions of software components using information obtained from the process specification (PSPEC), control specification (CSPEC), and state transition diagram (STD).

These models collectively form the design model, which is represented diagrammatically as a pyramid structure with data design at the base and component level design at the pinnacle. Note that each level produces its own documentation, which collectively form the design specifications document, along with the guidelines for testing individual modules and the integration of the entire package. Algorithm description and other relevant information may be included as an appendix.

9.2.2 Design Guidelines

In order to evaluate the quality of a design (representation) the criteria for a good design should be established. Such a design should:

- Exhibit good architectural structure
- Be modular
- Contain distinct representations of data, architecture, interfaces, and components (modules)
- Lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable design patterns
- Lead to components that exhibit independent functional characteristics
- Lead to interfaces that reduce the complexity of connections between modules and with the external environment
- be derived using a reputable method that is driven by information obtained during software requirements analysis

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology and through review.

9.2.3 Design Principles

Software design can be viewed as both a process and a model.

"The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. However, it is not merely a cookbook; for a competent and successful design, the designer must use creative skill, past experience, a sense of what makes "good" software, and have a commitment to quality.

The design model is equivalent to the architect's plans for a house. It begins by representing the totality of the entity to be built (e.g. a 3D rendering of the house), and slowly refines the entity to provide guidance for constructing each detail (e.g. the plumbing layout). Similarly the design model that is created for software provides a variety of views of the computer software." – adapted from book by R Pressman.

The set of principles which has been established to aid the software engineer in navigating the design process are:

1. The design process should not suffer from tunnel vision – A good designer should consider alternative approaches. Judging each based on the requirements of the problem, the resources available to do the job and any other constraints.
2. The design should be traceable to the analysis model – because a single element of the design model often traces to multiple requirements, it is necessary to have a means of tracking how the requirements have been satisfied by the model
3. The design should not reinvent the wheel – Systems are constructed using a set of design patterns, many of which may have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. The design should minimise intellectual distance between the software and the problem as it exists in the real world – That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
5. The design should exhibit uniformity and integration – a design is uniform if it appears that one person developed the whole thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

6. The design should be structured to degrade gently, even with bad data, events, or operating conditions are encountered – Well-designed software should never "bomb". It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
7. The design should be reviewed to minimize conceptual (semantic) errors – there is sometimes the tendency to focus on minute details when the design is reviewed, missing the forest for the trees. The designer team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax of the design model.
8. Design is not coding, coding is not design – Even when detailed designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made of the coding level address the small implementation details that enable the procedural design to be coded.
9. The design should be structured to accommodate change
10. The design should be assessed for quality as it is being created

When these design principles are properly applied, the design exhibits both external and internal quality factors. External quality factors are those factors that can readily be observed by the user, (e.g. speed, reliability, correctness, usability). Internal quality factors relate to the technical quality (which is important to the software engineer) more so the quality of the design itself. To achieve internal quality factors the designer must understand basic design concepts.

9.2.4 Fundamental Software Design Concepts

A set of fundamental software design concepts has evolved over the past four decades, each providing the software designer with a foundation from which more sophisticated design methods can be applied. Each concept helps the software engineer to answer the following questions:

1. What criteria can be used to partition software into individual components ?
2. How is function or data structure detail separated from a conceptual representation of software?
3. Are there uniform criteria that define the technical quality of a software design?

The fundamental design concepts are:

- **Abstraction** - allows designers to focus on solving a problem without being concerned about irrelevant lower level details (procedural abstraction - named sequence of events, data abstraction - named collection of data objects)
- **Refinement** - process of elaboration where the designer provides successively more detail for each design component
- **Modularity** - the degree to which software can be understood by examining its components independently of one another
- **Software architecture** - overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
- **Control hierarchy** or **program structure** - represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e.g. event sequences)
- **Structural partitioning** - horizontal partitioning defines three partitions (input, data transformations, and output); vertical partitioning (factoring) distributes control in a top-down manner (control decisions in top level modules and processing work in the lower level modules).
- **Data structure** - representation of the logical relationship among individual data elements (requires at least as much attention as algorithm design)
- **Software procedure** - precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure)
- **Information hiding** - information (data and procedure) contained within a module is inaccessible to modules that have no need for such information

9.3 Tools used for design and analysis

To deliver systems according to user's requirements, therefore, analysis is the heart of the process and is the key component in the system development life cycle.

To analyze the current system and to design the required system, software engineer uses some blueprints as a starting point for system design. The blueprints are the actual representation of the analysis done by the software engineers and are called tools of structured analysis. Some of them are as follows :

- Data Flow Diagram (DFD)
- Data Dictionary
- Decision Tree
- Structured English
- Decision Tables

9.4 Modularity

The concept of modularity in computer software has been advocated for about five decades. In essence, the software is divided into separately named and addressable components called modules that are integrated to satisfy problem requirements. A reader cannot easily grasp large programs comprised of a single module; the number of variables, control paths and sheer complexity would make understanding virtually impossible. Consequently a modular approach will allow for the software to be intellectually manageable. Note however that one cannot subdivide software indefinitely to make the effort required to understand or develop it negligible. This is because as the number of modules increases (causing the effort to develop them to decrease), the effort (cost) of associated with integrating the modules increases.

9.4.1 Cohesion

Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling. Cohesion is a measure of functional strength of a module. A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Classification of cohesion

The different classes of cohesion that a module may possess are depicted in fig.

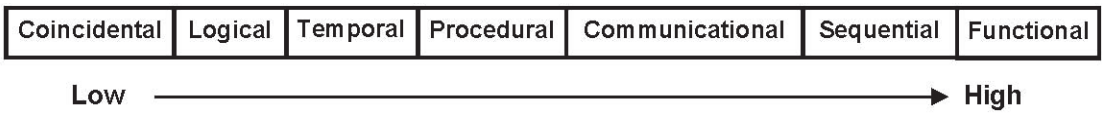


Fig 9.1 : Classification of cohesion

Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design. For example, in a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

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

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

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

Sequential cohesion: A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

Functional cohesion: Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion. Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

9.4.2 Coupling

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

Classification of Coupling

Even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will help to quantitatively estimating the degree of coupling between two modules. Five types of coupling can occur between any two modules. This is shown in fig. 9.2.

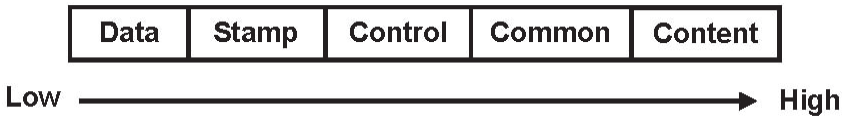


Fig. 9.2 Classification of coupling

Data coupling: Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

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

Common coupling: Two modules are common coupled, if they share data through some global data items.

Content coupling: Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

9.4.3 Functional independence

A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Need fo r functional independence

Functional independence is a key to any good design due to the following reasons:

- **Error isolation:** Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.
- **Scope of reuse:** Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
- **Understandability:** Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

9.5 Function-oriented design

The following are the salient features of a typical function-oriented design approach:

1. A system is viewed as something that performs a set of functions. Starting at this high-level view of the system, each function is successively refined into more detailed functions. For example, consider a function create-new-library-member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge.

Each of these sub-functions may be split into more detailed subfunctions and so on.

2. The system state is centralized and shared among different functions, e.g. data such as member-records is available for reference and updating to several functions.

9.6 Object-oriented design

In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

9.6.1 Function-oriented vs. object-oriented design approach

The following are some of the important differences between function-oriented and object-oriented design.

- Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real-world entities such as employee, picture, machine, radar system, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc. but by designing objects such as employees, departments, etc. Grady Booch sums up this difference as "identify verbs if you are after procedural design and nouns if you are after object-oriented design"
- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc.

are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by interrogating it. Of course, somewhere or other the real-world function must be implemented. In OOD, the functions are usually associated with specific real-world entities (objects); they directly access only part of the system state information.

- Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

It is not necessary an object-oriented design be implemented by using an object-oriented language only. However, an object-oriented language such as C++ supports the definition of all the basic mechanisms of class, inheritance, objects, methods, etc. and also support all key object-oriented concepts that we have just discussed. Thus, an object-oriented language facilitates the implementation of an OOD. However, an OOD can as well be implemented using a conventional procedural language – though it may require more effort to implement an OOD using a procedural language as compared to the effort required for implementing the same design using an object-oriented language.

Even though object-oriented and function-oriented approaches are remarkably different approaches to software design, yet they do not replace each other but complement each other in some sense. For example, usually one applies the top-down function-oriented techniques to design the internal methods of a class, once the classes are identified. In this case, though outwardly the system appears to have been developed in an object-oriented fashion, but inside each class there may be a small hierarchy of functions designed in a top-down manner.

9.7 Real Time Design

Real time embedded system are different from other types of software systems. They have correct functioning on the result produced by the system and the time at which these result are produced.

A soft real time system is a system where operation is degraded if results are not produced according to the specified timing requirements.

A hard real time system is a system whose operation is incorrect if the result are not produced according to time schedule.

Because of the need to respond to timing demands made by different stimuli/responses, the system architecture must allow for fast switching between stimulus handlers.

- Timing demands of different stimuli are different so a simple sequential loop is not usually adequate.
- Real-time systems are therefore usually designed as cooperating processes with a real-time executive controlling these processes.
- Designing both the hardware and the software associated with system. Partition functions to either hardware or software.
- Design decisions should be made on the basis on non-functional system requirements.
- Hardware delivers better performance but potentially longer development and less scope for change.

9.7.1 Design process of real time system

- Identify the stimuli to be processed and the required responses to these stimuli.
- For each stimulus and response, identify the timing constraints.
- Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response. Design algorithms to process each class of stimulus and response. These must meet the given timing requirements.
- Design a scheduling system which will ensure that processes are started in time to meet their deadlines.
- Integrate using a real-time operating system.

9.8 User Interface Design

User interface design is an essential part of the overall software design process. If a software system is to achieve its full potential, it is essential that its user interface should be designed to match the skills, experience and expectations of its anticipated users. Good user interface design is critical for system dependability. Many so-called 'user errors' are 'caused by the fact that user interfaces do not consider the capabilities of real users and their working environment. A poorly designed user interface means that users will probably be unable to access some of the system features, will make mistakes and will feel that the system hinders rather than helps them in achieving whatever they are using the system for.

When making user interface design decisions, you should take into account the physical and mental capabilities of the people who use software. I don't have space to discuss human issues in detail here but important factors that you should consider are :

- People have a limited short-term memory-we can instantaneously remember about seven items of information (Miller, 1957). Therefore, if you present users with too much information at the same time, they may not be able to take it all in.
- We all make mistakes, especially when we have to handle too much information or are under stress. When systems go wrong and issue warning messages and alarms, this often puts more stress on users, thus increasing the chances that they will make operational errors.
- We have a diverse range of physical capabilities. Some people see and hear better than others, some people are color-blind, and some are better than others at physical manipulation. You should not design for your own capabilities and assume that all other users will be able to cope.

9.8.1 User Interface Design Issues

Before going on to discuss the process of user interface design, I discuss some general design issues that have to be considered by UI designers. Essentially, the designer of a user interface to a computer is faced with two key questions:

1. How should the user interact with the computer system ?
2. How should information from the computer system be presented to the user?

A coherent user interface must integrate user interaction and information presentation. This can be difficult because the designer has to find a compromise between the most appropriate styles of interaction and presentation for the application, the background and experience of the system users, and the equipment that is available.

User interaction

User interaction means issuing commands and associated data to the computer system. On early computers, the only way to do this was through a command-line interface, and a special-purpose language was used to communicate with the machine. However, this was geared to expert users and a number of approaches have now evolved that are easier to use.

- **Direct manipulation -**The user interacts directly with objects on the screen. Direct manipulation usually involves a pointing device that indicates the object to be manipulated and the action, which specifies what should be done with that object. For example, to delete a file, you may click on an icon representing that file and drag it to a trash can icon.
- **Menu selection-**The user selects a command from a list of possibilities (a menu). The user may also select another screen object by direct manipulation, and the command operates on that object. In this approach, to delete a file, you would select the file icon then select the delete command.
- **Command language-**The user issues a special command and associated parameters to instruct the system what to do. To delete a file, you would type a delete command with the filename as a parameter.

9.8.2 User Interface design principles

User familiarity : The interface should use terms and concepts drawn from the experience of the people who make most use of the system.

Consistency : The interface should be consistent .in a way that, wherever possible, comparable operations should be activated in the same way.

Minimal surprise : Users should never be surprised by the behaviour of a system.

Recoverability : The interface should include mechanisms to allow users to recover from errors.

User guidance : The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.

User diversity : The interface should provide appropriate interaction facilities for different types of system users.

- We have different interaction preferences. Some people like to work with pictures, others with text. Direct manipulation is natural for some people, but others prefer a style of interaction that is based on issuing commands to the system.

The general principles are applicable to all user interface designs and should normally be instantiated as more detailed design guidelines for specific organizations or types of system.

The principle of user familiarity suggests that users should not be forced to adapt to an interface because it is convenient to implement. The interface should use terms that are familiar to the user, and the objects manipulated by the system should be directly related to the user's working environment. For example, if a system is designed for use by air traffic controllers, the objects manipulated should be aircraft, flight paths, beacons, and so on. Associated operations might be to increase or reduce air. craft speed, adjust heading and change height. The underlying implementation of the interface in terms of files and data structures should be hidden from the end user.

The principle of user interface consistency means that system commands and menus should have the same format, parameters should be passed to all commands in the same way, and command punctuation should be similar. Consistent interfaces reduce user learning time. Knowledge learned in one command or application i; therefore applicable in other parts of the system or in related applications.

Interface consistency across applications is also important. As far as possibly commands with similar meanings in different applications should be expressed in the same way. Errors are often caused when the same keyboard command, such as 'Control-b' means different things in different systems. For example, in the word processor that I normally use, 'Control-b' means embolden text, but in the graphics program that I use to draw diagrams, 'Control-b' means move the selected object behind another object. I make mistakes when using them together and sometimes try to embolden text in a diagram using the key combination.

The principle of minimal surprise is appropriate because people get very irritated when a system behaves in an unexpected way. As a system is used, users build a mental model of how the system works. If an action in one context causes a particular type of change, it is reasonable to expect that the same action in a different context will cause a comparable change. If something completely different happens, the user is both surprised and confused. Interface designers should therefore try to ensure that comparable actions have comparable effects.

Surprises in user interfaces are often the result of the fact that many interfaces are molded. This means that there are several modes of working (e.g., viewing mode and editing mode), and the effect of a command is different depending on the mode. It is very important that, when designing an interface, you include a visual indicator showing the user the current mode.

The principle of recoverability is important because users inevitably make mistakes when using a system. The interface design can minimize these mistakes (e.g. using menus means avoids typing mistakes), but mistakes can never be completely eliminated. Consequently, you should include interface facilities that allow users to recover from their mistakes. These can be of three kinds:

1. **Combination of destructive actions -** If a user specifies an action that is potentially destructive, the system should ask the user to confirm that this is really what is wanted before destroying any information.
2. **The provision of an undo facility -** Undo restores the system to a state before the action occurred. Multiple levels of undo are useful because users don't always recognise immediately that a mistake has been made.
3. **Checkpointing-**Checkpointing involves saving the state of a system at periodic intervals and allowing the system to restart from the last checkpoint. Then, when mistakes occur, users can go back to a previous state and start again. Many systems now include checkpointing to cope up with system failures but, paradoxically, they don't allow system users to use them to recover from their own mistakes.

   A related principle is the principle of user assistance. Interfaces should have built-in user assistance or help facilities. These should be integrated with the system and should provide different levels of help and advice. Levels should range from basic information on getting started to a full description of system facilities. Help systems should be structured so that users are not overwhelmed with information when they ask for help.

   The principle of user diversity recognizes that, for many interactive systems, there may be different types of users. Some will be casual users who interact occasionally with the system while others may be power users who use the system for several hours each day. Casual users need interfaces that provide guidance, but power users require shortcuts so that they can interact as quickly as possible. Furthermore, users may suffer from disabilities of various types and, if possible, the interface should be adaptable to cope with these. Therefore, you might include facilities to display enlarged text, to replace sound with text, to produce very large buttons and so on.

   The principle of recognizing user diversity can conflict with the other interface design principles, since some users may prefer very rapid interaction over, for example, user interface consistency. Similarly, the level of user guidance required can be radically different for different users, and it may be impossible to develop support that is suitable for all types of users. You therefore have to make compromises to reconcile the needs of these users.

9.8.3 User interface design process

User interface design is an iterative process involving close liaisons between users and designers.

The 3 core activities in this process are:

- **User analysis** - Understand what the users will do with the system;
- **System prototyping** - Develop a series of prototypes for experiment;
- **Interface evaluation-** Experiment with these prototypes with users.

9.8.4 User Analysis

If you don't understand what the users want to do with a system, you have no realistic prospect of designing an effective interface.

- User analysis have to be described in terms that users and other designers can understand.
- Scenarios where you describe typical episodes of use, are one way of describing these analyses.

9.8.5 User Interface Prototyping

- The aim of prototyping is to allow users to gain direct experience with the interface.
- Without such direct experience, it is impossible to judge the usability of an interface.

Prototyping may be a two-stage process:

- Early in the process, paper prototypes may be used;
- The design is then refined and increasingly sophisticated automated prototypes are then developed.

9.8.5 Paper prototyping

- Work through scenarios using sketches of the interface.
- Use a storyboard to present a series of interactions with the system.
- Paper prototyping is an effective way of getting user reactions to a design proposal.

Prototyping techniques

- **Script-driven prototyping -** Develop a set of scripts and screens using a tool such as Macromedia Director. When the user interacts with these, the screen changes to the next display.
- **Visual programming-** Use a language designed for rapid development such as Visual Basic.
- **Internet-based prototyping-** Use of web browser and associated scripts.

9.8.6 Interface Evaluation

- Some evaluation of a user interface design should be carried out to assess its suitability.
- Full scale evaluation is very expensive and impractical for most systems.
- Ideally, an interface should be evaluated against a usability specification. However, it is rare for such specifications to be produced.

Simple evaluation techniques

- Questionnaires for user feedback.
- Video recording of system use and subsequent tape evaluation.
- Instrumentation of code to collect information about facility use and user errors.
- The provision of code in the software to collect on-line user feedback.

9.9 Summary

- A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria.
- To analyze the current system and to design the required system software engineer uses some blue-prints as a starting point for system design. The blueprints are the actual representation of the analysis done by the software engineers and are called tools of structured analysis.
- The concept of modularity in computer software has been advocated for about five decades. In essence, the software is divided into separately names and addressable components called modules that are integrated to satisfy problem requirements.
- In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information.
- Real time embedded system are different from other types of software system. Their correct functioning on the result produced by the system and the time at which these result are produced.
- A soft real time system is a system where operation is degraded if results are not produced according to the specified timing requirements.
- A hard real time system is a system whose operation is incorrect if the result are not produced according to time schedule.
- User interface design is an essential part of the overall software design process. If a software system is to achieve its full potential, it is essential that its user interface should be designed to match the skills, experience and expectations of its anticipated users.

9.10 Self Assessment Questions

1. What are the steps followed in software design phase?
2. Explain Design concepts in software engineering.
3. Write a short note on software design tools.
4. What do mean by modular designing ?
5. What is function oriented design ?
6. What is the difference between function oriented and object oriented design ?
7. What is real time design ?
8. What is the role of user interface in software design and why it is important ?

9.11 Reference

- Pressman R.S., 'Software Engineering: A Practitioner's Approach', (Fifth Edition), McGraw-Hill, 2001.
- Keswani Bright, 'Software Engineering', (First Edition), Genius Publications, 2009.
- Jalote, Pankaj, 'An Integrated Approach to Software Engineering', (Second Edition), Narosa Publishing House, 1997.
- Puri S. and Singh G., 'Software Engineering Vol. 1', (Fourth Edition), Genius Publications, 2008.
- Sommerville I., 'Software Engineering', (Sixth Edition), Pearson Education, 2002.
- Schach S.R., 'Software Engineering', (Seventh Edition), Tata McGraw-Hill, New Delhi, 2007.
- Mall R., 'Fundamentals of Software Engineering', (Sixth Edition), Prentice-Hall of India, 2002.
- Gill N.S., 'Software Engineering: Software Reliability, Testing and Quality Assurance', Khanna Book Publishing Co (P) Ltd, New Delhi, 2002
- Sabharwal S., 'Software Engineering: Principles, Tools and Techniques', (Second Edition), Umesh Publications, Delhi, 2005.

Unit - 10 : Software Quality Assurance

Structure of the Unit

10.0 Objectives

After completing this unit you will be able to define

- What is software quality concepts ?
- Why Software quality assurance is required ?
- What is Software reliability ?
- Software quality plans
- ISO 9000 quality standards etc.

10.1 Introduction

The concept of software quality is more complex than what common people tend to believe.

However, it is very popular both for common people and IT professionals. If we look at the definition of quality in a dictionary, it is usual to find something like the following: set of characteristics that allows us to rank things as better or worse than other similar ones.

Different philosophers attempts to define software quality as a complex concept that can be decomposed in more detailed characteristics to enable evaluation of quality through the evaluation of more detailed characteristics that are supposed to be easy to measure or assess.

Software Quality Engineering is a process that evaluates, assesses, and improves the quality of software.

Software quality is often defined as "the degree to which software meets requirements for reliability, maintainability, transportability, etc., as contrasted with functional, performance, and interface requirements that are satisfied as a result of software engineering."

Quality must be built into a software product during its development to satisfy quality requirements established for it.

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.

10.2 Quality Concepts

Quality can be defined as as "a characteristic or attribute of something". As an attribute of an item, quality refers to measurable characteristics-thing we are able to compare to known standards such

as length, color, electrical properties, malleability, and so on. However, software, largely an intellectual entity, is more challenging to characterize than physical objects.

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-graded materials are used and 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 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 Quality.

Quality control

Variation control may be equated to quality control. But how do we achieve quality control? Quality control is the series of inspections reviews, and tests used throughout the development cycle to ensure that 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 clearly defined and measurable specifications. To which we may compare the outputs of each process. The feedback loop is essential to minimize the defects produced.

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.

Cost of Quality

Cost of quality includes all costs incurred in the pursuit of quality or in performing quality related activities. Cost of quality, to identify opportunities for reducing the cost of quality and to provide a normalized basis of comparison.

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 "first time through" each process.

appraisal costs include:

- In-process and inter process inspection
- Equipment calibration and maintenance
- Testing

Failure costs are costs 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 the costs incurred when we detect an error in our product prior to shipment . Internal failure costs include:

- Rework
- Repair
- Failure mode analysis

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

- Complaint resolution
- Product return and replacement
- Help line support
- Warranty work

10.3 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, directed, 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" .

Measures of Reliability and Availability

Early work in software reliability attempted to extrapolate the mathematics of hardware reliability theory. 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. There is still debate over the relationship between key concepts in hardware reliability and their applicability to software , although certain link has yet to 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 mean time between failure (MTBF), where

$$\mathbf{MTBF = MTTF + MTTR}$$

(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 them defects/KLOC . Stated simply an end user is concerned with failures, not with the total defect count. Because each defect contained within a program does not have the same failure rate, the total defect count provides little indication of the reliability of a system. For example, consider a program that has been in operation for 14 months. Many defects in this program may remain undetected for decades before they are discovered. The MTBF of such obscure defect might be 50 or even 100 years . Other defects in

as yet undiscovered, might have a failure rate if 18 or 24 months . Even if every one of the first category of defects (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 requirements at a given point in time and is defined as:

Availability = MTTF/ (MTTF + MTTR) ? 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.

Software safety

When software is used as part of the control system, complexity can increase by an order of magnitude or more. Design faults induced by human error-something that can be uncovered and eliminated in hardware-based conventional control become much more difficult to uncover when software is used.

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential risk that may affect software negatively and cause an entire system to fail. If risk can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential risk.

A modeling and analysis process is conducted as part of software safety. Initially, risks are identified and categorized by criticality and risk intensity. For example, some of the risk associated with a computer-based cruise control for an automobile might be :

- Causes uncontrolled acceleration that cannot be stopped
- Does not respond to depression of brake pedal (by turning off) does not engage when switch is activated
- Slowly loses or gains speed

Once these system-level risks are identified, analysis techniques are used to assign severity and probability of occurrence. To be effective, software must be analyzed in the context of the entire system. For example, a subtle user input error may be magnified by a software fault to produce control data that improperly positions a mechanical device. If a set of external environmental conditions are met, the improper position of the mechanical device will cause an unfortunate failure. Analysis techniques such as fault tree analysis , real-time logic, can be used to predict the chain of events that can cause risk and the probability that each of the events will occur to create the chain.

Once risk are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is, failures are not considered in a vacuum, but are evaluated in the context of an entire computer-based system.

10.4 SQA Plan

The SQA plan provides a road map for instituting software quality assurance. Developed by the SQA group and the project team, the plan serves as a template for SQA activities that are instituted for each software project.

The purpose and scope of the document and indicate those software process activities that are overed by quality assurance. All documents noted in the SQA plan are listed and all applicable standards are noted.

The Management section of the plan describes SQA's place in the organizational structure; SQA tasks and activities and their placement throughout the software process; and the organizational roles and responsibilities relative to product quality.

The Documentation section describes each of the work products produced as part of the software process. These include:

- Project documents (e.g., project plan)
- Models (e.g., ERDs, class hierarchies)
- Technical documents (e.g., specifications, test plans)
- User documents (e.g., help files)

In addition this section defines the minimum set of work products that are acceptable to achieve high quality.

Standards, Practices, and Conventions lists all applicable standards/practices that are applied during the software process (e.g., document standards, coding standards, and review guidelines). In addition all project, process, and (in some instances) product metrics that are to be collected as part of software engineering work are listed.

The Reviews and Audits section of the plan identifies the reviews and audits to be conducted by the software engineering team, the SQA group and the customer. It provides an overview of the approach for each review and audit.

I. Purpose of plan
II. References
III. Management
    1. Organization
    2. Tasks
    3. Responsibilities
IV. Documentation
    1. Purpose
    2. Required software engineering documents
    3. Other documents
V. Standards, Practices, and Conventions
    1. Purpose
    2. Conventions
VI. Reviews and Audits
    1. Purpose
    2. Review Requirements
        a. software requirements review
        b. design reviews
        c. software verification and validation reviews
        d. functional audit
        e. physical audit
        f. in-process audits
        g. management reviews

VII. Test

VIII Problem Reporting and Corrective Action

IX. Tools, Techniques, and Methodologies

X. Code Control

XI. Media Control

XII. Supplier Control

XIII. Records Collection, Maintenance, and Retention

XIV. Training

XV. Risk Management

It also defines test record-keeping requirements, Problem Reporting and Corrective Action defines procedures for reporting, tracking, and resolving errors and defects, and identifies the organizational responsibilities for these activities.

The remainder of the SQA plan identifies the tools and methods that support SQA activities and tasks references software configuration management procedures for controlling change; defines a contract management approach, establishes methods for assembling, safeguarding, and maintaining all records, identifies training required to meet the needs of the plan, and defines methods for identifying, assessing, monitoring, and controlling risks.

10.5 The ISO 9000 Quality Standards

A Quality assurance system may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management. ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semiannual surveillance audits ensure continued compliance to the standard.

The ISO Approach to Quality Assurance Systems

The ISO 9000 quality assurance models treat an enterprise as a network of interconnected processes. For a quality system to be ISO-compliant, these processes must address the areas identified in the standard and must be documented and practiced as described. Documenting a process helps an organization understand, control, and improve it. It is the opportunity to understand control and improve the process network that offers, perhaps, the greatest benefit to organizations that design and implement ISO compliant quality systems.

ISO 9000 describes the elements of a quality assurance system in general terms. These elements include organizational structure, procedures, processes, and resources needed to implement quality planning, quality control, quality assurance, and quality improvement. However, ISO 9000 does not describe how an organization should implement these quality system elements, Consequently, the challenge lies in designing and implementing a quality assurance system that meets the standard and fits the company's products, services, and culture.

The ISO 9001 Standard

ISO 9001 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. Because the ISO 9001 standard is applicable to all engineering disciplines, a special set of ISO guidelines (ISO 9000-3) have been developed to help interest the standard for use in the software process.

The 20 requirements delineated by ISO 9001 address the following topics:

1. Management responsibility
2. Quality system
3. Contract review
4. Design control
5. Document and data control
6. Purchasing
7. Control of customer supplied product
8. Product identification and tractability
9. Process control
10. Inspection and testing
11. Control of inspection, measuring, and test equipment
12. Inspection and test status
13. Control of nonconforming product
14. Corrective and preventive action
15. Handling, storage, packaging, preservation, and delivery
16. Control of quality records
17. Internal quality audits
18. Training
19. Servicing
20. Statistical techniques

In order for a software organization to become registered to ISO 9001, it must establish policies and procedures to address each of the requirements noted above and then be able to demonstrate that these policies and procedures are being followed.

10.6 Summary

- Software quality assurance is an umbrella activity that is applied at each step in the software process. SQA encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, poka-yoke devices, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms.
- SQA is complicated by the complex nature of software quality-an attribute of computer programs that is defined as "conformance to explicitly and implicitly specified requirements." But when considered more generally, software quality encompasses many different product and process factors and related metrics.
- Software reviews are one of the most important SQA activities. Reviews serve as filters throughout all software engineering activities, removing errors while they are relatively inexpensive to find and correct. The formal technical review is a stylized meeting that has been shown to be extremely effective in uncovering errors.
- To properly conduct software quality assurance, data about the software engineering process should be collected, evaluated, and disseminated. Statistical SQA helps to improve the quality of the product and the software process itself.
- Software reliability models extend measurements, enabling collected defect data to be extrapolated into projected failure rates and reliability predictions.

- Software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable managerial and technological space of software engineering. The ability to ensure quality is the measure of a mature engineering discipline. When the mapping is successfully accomplished, mature software engineering is the result.

10.7 Self Assessment Questions

1. What is Software Quality?
2. What is software reliability?
3. Explain ISO 9000 Quality Standards.

10.8 References

- Roger S. Pressman, Software Engineering : Practioners approach, 6th Edition, The McGrawhill Companies, New York.
- Sommerville, I (2000), Software Engineering, 6th Edition, Pearson
- Pratap K.J. Mohpatra, Software Engineering, A lifecycle Approach, Ist Edition, New age Informational Publishers.
- Ali Behforanz and Prederick J. Hudson, Software Engineering Fundamentals, Oxford University Press, first Edition, 1966

Unit -11 : Testing Fundamentals

Structure of the Unit

11.0 Objectives

Afther Completing this unit you will understang :

Different testing principles

Test cases for testing

White Box and Black Box Testing

Control strucutre and basis path testing

11.1 Introduction

Testing presents an interesting role for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. Now comes testing. The engineer creates a series of test cases that are intended to "demolish" the software that has been built.

Testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.

Software engineers are by their nature constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered.

11.2 Testing Objectives

In software testing, there are number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of coding an error.
2. A good test case is one that has a high probability of coding an as-yet- undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

These objectives imply a impressive change in viewpoint. They move counter to the commonly held view that a successful test is one in which no errors are found. The objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

If testing is conducted successfully according to the objectives, it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.

In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present. It is important to keep this statement in mind as testing is being conducted.

11.3 Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the following basic principles that guide software testing.

All tests should be traceable to customer requirements. As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

The Pareto principle applies to software testing. The Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Testing should begin in the small and progress toward testing in the large The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component level design have been exercised.

To be most effective, testing should be conducted by an independent third party. The software engineer who created the system is not the best person to conduct all tests for the software. It should be conducted by a third party.

11.4 Test Case Design

The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. Software engineers often treat testing as an afterthought, developing test cases that may "feel right" but have little assurance of being complete.

The objectives of testing, states that we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort.

A rich variety of test case design methods have evolved for software. These methods provide the developer with a systematic approach to testing. More important, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any engineered product can be tested in one of two ways:

1. Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.
2. Knowing the internal workings of a product, tests can be conducted to ensure that, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach is called black-box testing and the second, white-box testing.

When computer software is considered, **Black-Box** testing is conducted at the software interface. Although they are designed to uncover errors, black-box tests are used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information (e.g., a database) is maintained. A black-box test examines some fundamental aspects of a system with little regard for the internal logical structure of the software.

White-Box Testing of software is predicated on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The "status of the program" may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

At first glance it would seem that very thorough white-box testing would lead to 100 percent correct programs. All we need to do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, use test cases to exercise program logic in detail. Unfortunately, complete testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. Complete testing is impossible for large software systems.

So a limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity. The attributes of both black-box and white-box testing can be combined to provide an approach that validates the software interface and selectively ensures that the internal workings of the software are correct.

11.5 White-Box Testing

White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that:

1. Guarantee that all independent paths within a module have been exercised at least once,
2. Exercise all logical decisions on their true and false sides,
3. Execute all loops at their boundaries and within their operational bounds, and
4. Exercise internal data structures to ensure their validity.

11.5.1 Basis Path Testing

Basis path testing is a white-box testing technique. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

11.5.2 Control Structure Testing

The basis path testing technique is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. Here other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.

11.5.2.1 Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a

program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¨) operator. A relational expression takes the form

E1 < relational-operator > E2

Where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following:

Relational Operators (< ,> <=, >=) , compound condition, Boolean condition AND ,OR, NOT etc.

A condition without relational expression is referred to as a Boolean expression. Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

1. Boolean operator error (incorrect/missing/extra Boolean operators).
2. Boolean variable error.
3. Boolean parenthesis error.
4. Relational operator error.
5. Arithmetic expression error.

The condition testing method focuses on testing each condition in the program. Condition testing strategies generally have two advantages. First, measurement of test coverage of a condition is simple. Second, the test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program ....

11.5.2.2 Data Flow Testing

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. A number of data flow testing strategies have been studied and compared.

11.5.2.3 Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined, simple loops, concatenated loops, nested loops, and unstructured loops.

Simple loops. The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where m \< n.
5. n-1,n,n+1 Passes through the loop.

Nested loops. If we extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests.

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

Concatenated loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

11.6 Black Box Testing

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories:

1) Incorrect or missing functions
2) Interface errors
3) Errors in data structures or external database access,
4) Behavior or performance errors.
5) Initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black- box testing tends to be applied during later stages of testing. Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria:

1) Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing.
2) Test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

11.7 Summary

- The primary objective for test case design is to derive a set of tests that have the highest likelihood for uncovering errors in the software. To accomplish this objective, two different categories of test case design techniques are used: white-box testing and black-box testing.
- White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.
- Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough tests coverage. Equivalence partitioning divides the input domain into classes of data that are likely to exercise specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters.
- Specialized testing methods encompass a broad array of software capabilities and application areas. Testing for graphical user interfaces, client/server architectures, documentation and help facilities, and real-time systems, each require specialized guidelines and techniques.
- Experienced software developers often say, "Testing never ends, it just gets transferred from you the software engineer to your customer. Every time your customer uses the program, a test is being conducted." By applying test case design, the software engineer can achieve more complete testing and thereby uncover and correct the highest number of errors before the "customer's tests" begin.

11.8 Self Assessment Questions

1. What is White Box Testing?
2. What is Black Box Testing
3. What do you mean Test Case Design?
4. What is Control Structure Testing?

11.9 References

- Roger S. Pressman, Software Engineering : Practioners approach, 6th Edition, The McGrawhill Companies, New York.
- Sommerville, I (2000), Software Engineering, 6th Edition, Pearson
- Pratap K.J. Mohpatra, Software Engineering, A lifecycle Approach, Ist Edition, New age Informational Publishers.
- Ali Behforanz and Prederick J. Hudson, Software Engineering Fundamentals, Oxford University Press, first Edition, 1966

Unit - 12 : Software Testing Strategies

Structure of the Unit

12.0 Objectives

After completing this unit you will understand :

- The difference between software verification and software validation.
- Program inspection methods of discovering defects in program.
- Automated static analysis and how it is used in verification and validation.
- How static verification is used in the clean room development process.

12.1 Introduction

Software Verification and Validation (V&V) is the process of ensuring that software being developed or changed will satisfy functional and other requirements (validation) and each step in the process of building the software yields the right products (verification). The differences between verification and validation are unimportant except to the theorist, practitioners use the term V&V to refer to all of the activities that are aimed at making sure that the software will function as required.

V&V is intended to be a systematic and technical evaluation of software and associated products of the development and maintenance processes. Reviews and tests are done at the end of each phase of the development process to ensure software requirements are complete and testable and that design, code, documentation, and data satisfy those requirements.

12.2 Verification and Validation Activities

The two major V&V activities are reviews, including inspections and walkthroughs, and testing.

1. **Reviews, Inspections, and Walkthroughs**
   Reviews are conducted during and at the end of each phase of the life cycle to determine whether established requirements, design concepts, and specifications have been met. Reviews

consist of the presentation of material to a review board or panel. Reviews are most effective when conducted by personnel who have not been directly involved in the development of the software being reviewed.

Informal reviews are conducted on an as-needed basis. The developer chooses a review panel and provides and/or presents the material to be reviewed. The material may be as informal as a computer listing or hand-written documentation.

Formal reviews are conducted at the end of each life cycle phase. The acquirer of the software appoints the formal review panel or board, who may make or affect a go/no-go decision to proceed to the next step of the life cycle.

Formal reviews include the Software Requirements Review, the Software Preliminary Design Review, the Software Critical Design Review, and the Software Test Readiness Review.

An inspection or walkthrough is a detailed examination of a product on a step-by-step or line-of-code by line-of-code basis. The purpose of conducting inspections and walkthroughs is to find errors. The group that does an inspection or walkthrough is composed of peers from development, test, and quality assurance.

2. Testing

Testing is the operation on the software with real or simulated inputs to demonstrate that a product satisfies its requirements and, if it does not, to identify the specific differences between expected and actual results. There are varied levels of software tests, ranging from unit or element testing through integration testing and performance testing, up to software system and acceptance tests.

a. Informal Testing

Informal tests are done by the developer to measure the development progress. "Informal" in this case does not mean that the tests are done in a casual manner, just that the acquirer of the software is not formally involved, that witnessing of the testing is not required, and that the prime purpose of the tests is to find errors. Unit, component, and subsystem integration tests are usually informal tests.

Informal testing may be requirements-driven or design- driven. Requirements-driven or black box testing is done by selecting the input data and other parameters based on the software requirements and observing the outputs and reactions of the software. Black box testing can be done at any level of integration. In addition to testing for satisfaction of requirements, some of the objectives of requirements-driven testing are to ascertain:

- Computational correctness.
- Proper handling of boundary conditions,
- Including extreme inputs and conditions that cause extreme outputs.
- State transitioning as expected.
- Proper behavior under stress or high load.
- Adequate error detection, handling, and recovery.

Design-driven or white box testing is the process where the tester examines the internal workings of code. Design- driven testing is done by selecting the input data and other parameters based on the internal logic paths that are to be checked.

The goals of design-driven testing include ascertaining correctness of:

- All paths through the code. For most software products, this can be feasibly done only at the unit test level.
- Bit-by-bit functioning of interfaces.
- Size and timing of critical elements of code.

b. Formal Tests

Formal testing demonstrates that the software is ready for its intended use. A formal test should include an acquirer- approved test plan and procedures, quality assurance witnesses, a record of all discrepancies, and a test report. Formal testing is always requirements-driven, and its purpose is to demonstrate that the software meets its requirements.

Each software development project should have at least one formal test, the acceptance test that concludes the development activities and demonstrates that the software is ready for operations.

In addition to the final acceptance test, other formal testing may be done on a project. For example, if the software is to be developed and delivered in increments or builds, there may be incremental acceptance tests. As a practical matter, any contractually required test is usually considered a formal test; others are "informal."

After acceptance of a software product, all changes to the product should be accepted as a result of a formal test. Post acceptance testing should include regression testing. Regression testing involves rerunning previously used acceptance tests to ensure that the change did not disturb functions that have previously been accepted.

C. Verification and Validation

During the Software Acquisition Life Cycle, the V&V Plan should cover all V&V activities to be performed during all phases of the life cycle. The V&V Plan Data Item Description (DID) may be rolled out of the Product Assurance Plan DID contained in the SMAP Management Plan Documentation Standard and DID.

1. Software Concept and 3Initiation Phase

The major V&V activity during this phase is to develop a concept of how the system is to be reviewed and tested.

Simple projects may compress the life cycle steps; if so, the reviews may have to be compressed. Test concepts may involve simple generation of test cases by a user representative or may require the development of elaborate simulators and test data generators. Without an adequate V&V concept and plan, the cost, schedule, and complexity of the project may be poorly estimated due to the lack of adequate test capabilities and data.

2. Software Requirements Phase

V&V activities during this phase should include:

- Analyzing software requirements to determine if they are consistent with, and within the scope of system requirements.
- Assuring that the requirements are testable and capable of being satisfied.
- Creating a preliminary version of the Acceptance Test Plan, including a verification matrix, which relates requirements to the tests used to demonstrate that requirements are satisfied.
- Beginning development, if needed, of test beds and test data generators.
- The phase-ending Software Requirements Review (SRR).

3. Software Architectural (Preliminary) Design Phase

V&V activities during this phase should include:

- Updating the preliminary version of the Acceptance Test Plan and the verification matrix.
- Conducting informal reviews and walkthroughs or inspections of the preliminary software and data base designs.

- The phase-ending Preliminary Design Review (PDR) at which the allocation of requirements to the software architecture is reviewed and approved.

4. Software Detailed Design Phase

V&V activities during this phase should include:

- Completing the Acceptance Test Plan and the verification matrix, including test specifications and unit test plans.
- Conducting informal reviews and walkthroughs or inspections of the detailed software and data base designs.
- The Critical Design Review (CDR) which completes the software detailed design phase.

5. Software Implementation Phase

V&V activities during this phase should include:

- Code inspections and/or walkthroughs.
- Unit testing software and data structures.
- Locating, correcting, and retesting errors.
- Development of detailed test procedures for the next two phases.

6. Software Integration and Test Phase

This phase is a major V&V effort, where the tested units from the previous phase are integrated into subsystems and then the final system. Activities during this phase should include:

- Conducting tests per test procedures.
- Documenting test performance, test completion, and conformance of test results versus expected results.
- Providing a test report that includes a summary of nonconformance found during testing.
- Locating, recording, correcting, and retesting nonconformance.
- The Test Readiness Review (TRR), confirming the product's readiness for acceptance testing.

7. Software Acceptance and Delivery Phase V&V activities during this phase should include:

- By test, analysis, and inspection.
- Demonstrating that the developed system meets its functional, performance, and interface requirements.
- Locating, correcting, and retesting nonconformance's.
- The phase-ending Acceptance Review (AR).

8. Software Sustaining Engineering and Operations Phase

Any V&V activities conducted during the prior seven phases are conducted during this phase as they pertain to the revision or update of the software.

D. Independent Verification and Validation Independent Verification and Validation (IV&V)

It is a process whereby the products of the software development life cycle phases are independently reviewed, verified, and validated by an organization that is neither the developer nor the acquirer of the software. The IV&V agent should have no stake in the success or failure of the software. The IV&V agent's only interest should be to make sure that the software is thoroughly tested against its complete set of requirements.

The IV&V activities duplicate the V&V activities step-by- step during the life cycle, with the exception that the IV&V agent does no informal testing. If there is an IV&V agent, the formal acceptance testing may be done only once, by the IV&V agent. In this case, the developer will do a formal demonstration that the software is ready for formal acceptance.

12.3 Software Inspections

Perhaps more tools have been developed to aid the V&V of software (especially testing) than any other software activity. The tools available include code tracers, special purpose memory dumpers and formatters, data generators, simulations, and emulations. Some tools are essential for testing any significant set of software, and, if they have to be developed, they may turn out to be a significant cost and schedule driver.

An especially useful technique for finding errors is the formal inspection. Formal inspections were developed by Michael Fagan of IBM. Like walkthroughs, inspections involve the line-by-line evaluation of the product being reviewed. Inspections, however, are significantly different from walkthroughs and are significantly more effective.

Inspections are done by a team, each member of which has a specific role. The team is led by a moderator, who is formally trained in the inspection process. The team includes a reader, who leads the team through the item; one or more reviewers, who look for faults in the item; a recorder, who notes the faults; and the author, who helps explain the item being inspected.

This formal, highly structured inspection process has been extremely effective in finding and eliminating errors. It can be applied to any product of the software development process, including documents, design, and code. One of its important side benefits has been the direct feedback to the developer/ author, and the significant improvement in quality that results.

12.4 Verification and Formal Methods

Formal method of software development is based on mathematical representation of the software, usually as a formal specification. These formal methods are mainly concerned with a mathematical analysis of the specification with transforming the specification to a more detailed, semantically equivalent representation or with the formally verifying that one representation of the system is semantically equivalent to another representation.

Formal method may be used at different stages in the validation and verification process.

1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors.
2. A formal verification using mathematically arguments that the code of a software is consistent with its specification. This requires a formal specification and is effective in discovering programming and some design errors . A transformational development process where a formal specification is transformed through a series of more detailed representation or a cleanroom process may be used to support the formal verification process.

12.4.1 Clean Room Software Development

The Clean room process is a theory-based, team-oriented process for the development and certification of high-reliability software systems under statistical quality control. Its principle objective is to develop software that exhibits zero failures in use. For this purpose the life cycle is different from conventional software development techniques. The approach combines mathematical-based methods of software specification, design and correctness verification with statistical, usage-based testing to certify software fitness for use. Therefore the goals in this method is to reduce the failures found during testing by enabling good and correct designs that avoid rework. Most designs pass through detailed specifications and modeling which are evaluated and proved for correctness using formal methods. In this paper we take a look at the Cleanroom software process and describe a few of its methods.

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

Requirement Analysis: To define requirements for the software product (including function,usage, environment, and performance) as well as to obtain agreement with the customer on the requirements as the basis for function and usage specification. Requirements analysis may identify opportunities

to simplify the customer's initial product concept and to reveal requirements that the customer has not addressed.

Function Specification: To make sure the requirement behavior of the software in all possible circumstances of use is defined and documented. The function specification is complete, consistent, correct, and traceable to the software requirements. The customer agrees with the function specification as the basis for software development and certification. This process is to express the requirements in a mathematically precise, complete, and consistent form.

Usage Specification: The purpose is to identify and classify software users, usage scenarios, and environments of use, to establish and analyze the highest level structure and probability distribution for software usage models, and to obtain agreement with the customer on the specified usage as the basis for software certification.

Architecture Specification: The purpose is to define the 3 key dimensions of architecture: Conceptual architecture, module architecture and execution architecture. The Cleanroom aspect of architecture specification is in decomposition of the history-based black box Function Specification into state-based state box and procedure-based clear box descriptions. It is the beginning of a referentially transparent decomposition of the function specification into a box structure hierarchy, and will be used during increment development.

Increment Planning: The purpose is to allocate customer requirements defined in the Function specification to a series of software increments that satisfy the Software Architecture, and to define schedule and resource allocations for increment development and certification. In the incremental process, a software system grows from initial to final form through a series of increments that implement user function, execute in the system environment, and accumulate into the final system.

Cleanroom Development Processes

There are three terms involved when the cleanroom process is used for large system development:

Software Reengineering: The purpose is to prepare reused software (whether from Cleanroom environments or not) for incorporation into the software product. The functional semantics and interface syntax of the reused software must be understood and documented, and if incomplete, can be recovered through function abstraction and correctness verification. Also, the certification goals for the project must be achieved by determining the fitness for use of the reused software through usage models and statistical testing.

Increment Design: The purpose is to design and code a software increment that conforms to Cleanroom design principles. Increments are designed and implemented as usage hierarchies through box structure decomposition, and are expressed in procedure-based clear box forms that can introduce new black boxes for further decomposition. The design is performed in such a way that it is provably correct using mathematical models. Treating a program as a mathematical function can do this. Note that specification and design are developed in parallel, resulting in a box structure hierarchy affording complete traceability.

Correctness Verification: The purpose is to verify the correctness of a software increment using mathematically based techniques. Black box specifications are verified to be complete, consistent, and correct. State box specifications are verified with respect to black box specifications, and clear box procedures are verified with respect to state box specifications. A set of correctness questions is asked during functional verification. Correctness is established by group consensus and/or by formal proof techniques. Any part of the work changed after verification, must be reverified.

12.5 Unit Testing

In Unit testing, the module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.

Boundary conditions are tested to ensure that the module operates properly at boundaries established

to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing. Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow. Basis path and loop testing are effective techniques for uncovering a broad array of path errors.

Among the more common errors in computation are :

1) misunderstood or incorrect arithmetic precedence,
2) mixed mode operations,
3) incorrect initialization,
4) precision inaccuracy,
5) incorrect symbolic representation of an expression.

Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison). Test cases should uncover errors such as

1) comparison of different data types,
2) incorrect logical operators or precedence,
3) expectation of equality when precision error makes equality unlikely,
4) incorrect comparison of variables,
5) improper or nonexistent loop termination,
6) failure to exit when divergent iteration is encountered, and
7) improperly modified loop variables.

12.6 Integration testing

Integration testing is a systematic technique for constructing the program structure at the same time when conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design. There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

In Integration Testing the program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test Approach may be applied.

Big Bang

The way this integration testing type works is, most or all of the modules are integrated together to form nearly a complete system. This is very similar to system testing as this basically has a whole system before starting the testing.

There are not many advantages of the big bang integration; the only advantage is that for a smaller system this will be the ideal integration testing technique.

The disadvantages is that you would have to wait for all the modules to be integrated in order to do big-bang testing so there will be quite a lot of delay. Any errors are identified at a very late stage and it is very hard to identify the fault. It is very difficult to be sure that all testing has been done before product release.

Top-Down testing

This is where the highest level components are tested first and then step by step we start working downwards (lower components). Top-down testing mainly requires for the testing team to separate what is important and what is least important, then the most important modules are worked on first. The top-down approach is similar to a binary tree you would start off by integrating the top level before slowly working your way down the tree and integrating all the components at that level.

The advantage to this way of testing is that if a prototype is released or shown then most of the main functionality will already be working. It is also easy to maintain the code and there will be better control in terms of errors so most of the errors would be taken out before going to the next stage of testing.

The disadvantage is that it is hard to test the lower level components using test data. The other thing is that lower level modules may not be tested as much as the upper level modules.

Bottom-up testing

This is the opposite to top-down where you test the lower level components and start testing your way upwards to the higher level components. The components will be separated into the level of importunacy and the least important modules will be worked on first, then slowly you would work your way up by integrating components at each level before moving upwards.

The advantage is that with this method you would be able to maintain code more easily and there is a more clear structure of how to do things.

The disadvantage is that when releasing a prototype you can not see a working prototype until nearly all the program has been completed so that may take a long time before this happens. There may be a lot of errors later on regarding the GUI and programming.

Integration - Overall

Integration testing is best to be used in an iterative process as this way it will save time in the long run and will help when trying to keep to a budget.

The reason why an iterative process is the best for integration testing is simply because this allows for more feedback from the client, so if the client is involved with the project a lot it would be less likely in terms of having to make a lot of changes in the integration process, for example test plan.

The best integration testing type would be top-down as this is the fastest way for the integration process to be completed. There is however one problem that may cause time delay in the integration process that is the when using testing data to test the process if faults or errors are reported. This will have to be documented, fixed and re-done and will cause a delay in the time taken.

Integration testing again uses black box testing as all the integration stage is doing is checking for the correct outputs.

12.7 Validation Testing

At the culmination of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests-validation testing-may begin. Validation can be defined in many ways, but a simple (Albeit Harsh) definition is that "validation succeeds when software functions in a manner that can be reasonably expected by the customer". At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?"

Reasonable expectations are defined in the Software Requirements Specification a document that describes all user-visible attributes of the software. The specification contains a section called Validation Criteria. Information contained in that section forms the basis for a validation testing approach.

Validation Test Criteria

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and humanengineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exist:

1) The function or performance characteristics conform to specification and are accepted or
2) A deviation from specification is uncovered and a deficiency list is created.
3) Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

Configuration Review

An important element of the validation process is a configuration review. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle. The configuration review, sometimes called an audit.

Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end-user seems able to find.

The alpha test is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The beta test is conducted at one or more customer sites by the end-user of the

software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

12.8 System Testing

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, we discuss the types of system tests that are worthwhile for software-based systems.

Recovery Testing

Many computer based systems must recover from faults and resume processing within a prespecified time. In some cases, a system must 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 of time 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 recovery is automatic (performed by the system itself), initialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer "The system's security must, of course, be tested for invulnerability from frontal attack-but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?" Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,

1) Special tests may be designed that generate ten interrupts per second, when one or two is the average rate.
2) Input data rates may be increased by an order of magnitude to determine how input functions will respond.
3) Test cases that require maximum memory or other resources are executed.
4) Test cases that may cause thrashing in a virtual operating system are designed.
5) Test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called sensitivity testing. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By incrementing a system, the tester can uncover situations that lead to degradation and possible system failure.

12.9 Summary

- Verification And validation are not the same thing. Verification show that a program meets its specification. And validation shows that the program does what the requires.
- Test plan should include a details of the items to be tested, the testing schedule, the procedure for managing the testing process, the hardware and software requirements , and any testing problems that may be arise.
- Program inspection are effective in finding program errors. The aim of an inspection is to search faults. A fault check list should derive the inspection process.
- Verification technique involved examination and analysis of the program to find the errors .
- In a program inspection a team checks the code.
- Clean room development relies on static technique for program verification and statistical testing for system reliability certification, it has successful in producing systems that have a high level of reliability.

12.10 Self Assessment Questions

1. Give definition to validation and verification (V&V)
2. Two main objectives of the V&V.
3. Name two types of verification.
4. What is the goal of program testing?
5. What are two types of testing?
6. Explain the difference between testing and debugging.
7. What is the software test plan?
8. What is "inspection"?
9. What is a static analyser?
10. Three stages of static analysis?

10.11 References

- Roger S. Pressman, Software Engineering : Practioners approach, 6th Edition, The McGrawhill Companies, New York.
- Sommerville, I (2000), Software Engineering, 6th Edition, Pearson
- Pratap K.J. Mohpatra, Software Engineering, A lifecycle Approach, Ist Edition, New age Informational Publishers.

Unit - 13 : Software Re-Engineering

Structure of the Unit

13.0 Objectives

The objective of this chapter is to explain the process of software reengineering to improve the maintainability of a software system. When you have read this chapter, you will:

- Understand why re-engineering is sometimes a cost-effective option for software system evolution.
- Understand the activities such as reverse engineering and program restructuring which may be involved in the software re-engineering process.

13.1 Introduction

The objective of re-engineering is to improve the system structure to make it easier to understand and maintain. In re-engineering, the functionality and system architecture remains the same but it includes re-documenting, organizing and restricting, modifying and updating the system. It is a solution to the problems of system evolution. In other words,

Re-engineering essentially means having a re-look at an entity, such as a process, task, designs, approach, or strategy using engineering principles to bring in radical and dramatic improvements.

Re-engineering basically refers to re-implement systems to make them more maintainable. This approach attacks five parameters, namely: management philosophy, pride, policy, procedures, and practices to bring in radical improvements impacting cost, quality, service, and speed. When re-engineering principles are applied to business process then it is called Business Process Re-engineering (BRP).

The re-engineering process involves source code translation, reverse engineering, program structure improvement, program modularisation and data re-engineering.

- Source code translation is the automatic conversion of program in one language to another.
- Restructuring or rewriting part or all of a system without changing its functionality
- Applicable when some (but not all) subsystems of a larger system require frequent maintenance

- Reengineering involves putting in the effort to make it easier to maintain
- The reengineered system may also be restructured and should be redocumented

13.2 Why Software Reengineering?

Every software has limited age. At a specific period of time, it is needed to update or re-change or re-structure according to need of the organization or user requirements. Thus, is needed for the following reason-

- Legacy software are increasing
- New technology appearing
- Successful ratio of projects increasing
- Companies are more competing
- Quality attributes are demanding
- People are changing (developers joining and leaving, customers are changing)
- Software maintenance are pressing

13.3 Advantages of Software re-engineering

The Software re-engineering has following advantages-

1. **Reduced risk**
   There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
2. **Reduced cost**
   The cost of re-engineering is often significantly less than the costs of developing new software.

13.4 Disadvantages of Software re-engineering

The main disadvantage of software re-engineering is that there are practical limits to the extent that a system can be improved by re-engineering. It is not possible, for example, to convert a system written using a functional approach to an object-oriented system. Major architectural changes or radical re-organising of the system data management cannot be carried out automatically so involve high additional costs. Although re-engineering can improve maintainability, the reengineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

13.5 Software Reengineering Process Model

13.5.1 Reengineering Process

Following figure illustrates a possible re-engineering process. The input to the process is a legacy program and the output is a structured, modularised version of the same program. At the same time as program re-engineering, the data for the system may also be re-engineered.

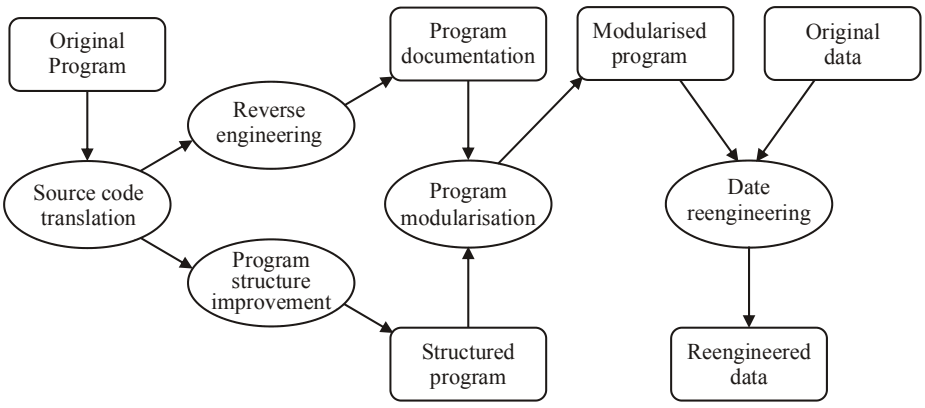


Fig. 13.1

The activities in this re-engineering process are:

Source code translation: The program is converted from an old programming language to a more modern version of the same language or to a different language.

Reverse engineering: The program is analysed and information, extracted from it which helps to document its organisation and functionality.

Program structure improvement: The control structure of the program is analysed and modified to make it easier to read and understand.

Program modularisation: Related parts of the program are grouped together and, where appropriate, redundancy is removed.

Data re-engineering: The data processed by the program is changed to reflect program changes.

13.5.2 Software Reengineering Process Model

Re-engineering of information systems is an activity that will absorb information technology resources for many years. That's why every organization needs a pragmatic strategy for software re-engineering. Re-engineering is a rebuilding activity. To implement re-engineering principles, we apply a software re-engineering process model. The re-engineering paradigm shown in the following figure is a cyclical model.

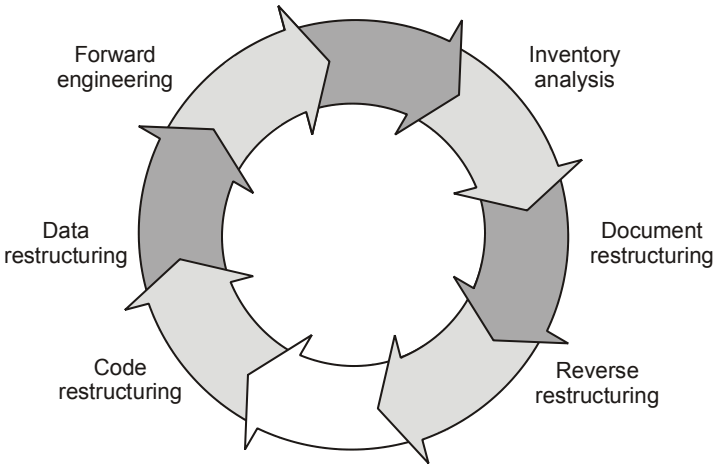


Fig. 13.2 Software Re-engineering Process Model

There are six activities in the model:

1. Inventory analysis
2. Document restructuring
3. Reverse engineering
4. Code restructuring
5. Data restructuring
6. Forward engineering

The activities of the software re-engineering process model are described as follows:

1. Inventory Analysis

Every software organization should have an inventory of all applications. The inventory can be nothing more than spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability, and other locally important criteria, candidates for re-engineering appear. Resources can then be allocated to candidate applications for re-engineering work. It is important to note that the inventory should be revisited on a regular cycle. The status of applications (e.g., business criticality) can change as a function of time and as a result, priorities for re-engineering will shift.

2. Document Restructuring

Weak documentation is the trademark of many legacy systems. But what do we do about it? What are our options?

(i) **Creating documentation is far too time consuming:** If the system works, we'll live with what we have. In some cases, this is the correct approach. It is not possible to recreate documentation for hundreds of computer programs. If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change.

(ii) **Documentation must be updated, but we have limited resources:** We will use a "document when touched" approach. It may not be necessary to fully re-document an application. Rather, those portions of the system that are currently undergoing change are fully documented. Over time, a collection of useful and relevant documentation will evolve.

(iii) **The system is business critical and must be fully re-documented:** Even in this case, an intelligent approach is to pare documentation down to an essential minimum.

(iv) **Each of these options is viable :** A software organization must choose the one that is most appropriate for each case

3. Reverse Engineering

Reverse engineering for software is quite similar. In most cases, however, the program to be reverse engineered is not a competitor's. Rather, it is the company's own work (often done many years earlier). The "secrets" to be understood are obscure because no specification was ever developed. Therefore, reverse engineering for software is the process of analyzing a program in an effort to create is presentation of the program at a higher level of abstraction than source code. Reverse-engineering is a process of design recovery. Reverse-engineering tools extract data and architectural and procedural design information from an existing program.

4. Code Restructuring

The most common type of re-engineering is code restructuring. Some legacy systems have relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases the code within the suspect modules can be restructured. To accomplish this activity, the source code is analyzed using are structuring tool. Violations of structured programming constructs are noted, and code is then restructured (this can be done automatically).The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

5. Data Restructuring

Programs with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, data architecture has more to do with the long-term viability of a program than the source code itself unlike code restructuring, which occurs at a relatively low level of abstraction, data structuring is a full-scale re-engineering activity. In most cases, data restructuring begins with a reverse-engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

6. Forward Engineering

Applications would be rebuilt using an automated "re-engineering engine." The old program would be fed into the engine analyzed, restructured, and then regenerated in a form that exhibited the best aspects of a software quality. CASE vendors have introduced tools that provide a limited subset of these capabilities that address specific application domains. Forward engineering, also called renovation or reclamation, not only recovers design information from existing software, but also uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.

13.6 Software Reverse Engineering

Reverse engineering is the process of deriving the system design and specification from its source code. Program structure improvement replaces unstructured control constructs with while loops and simple conditionals. Program modularisation involves reorganisation to group related items. Data re-engineering may be necessary because of inconsistent data management.

- Analyzing software with a view to understanding its design and specification
- May be part of the reengineering process
- May be used to specify a system for prior to reimplementation
- Program understanding tools may be useful (browsers, cross-reference generators, etc.)

13.6.1 Characteristics of Reverse Engineering

The various characteristics of reverse software engineering are as follows:

1. Abstraction Level

- The abstraction level of a reverse-engineering process and the tools used to affect it refers to the sophistication of the design information that can be extracted from the source code. Ideally, the abstraction level should be as high as possible. As the abstraction level increases the software engineer is provided with information that will allow easier understanding of the program

2. Completeness

- The completeness of a reverse-engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given source code listing, it is relatively easy to develop a complete procedural design representation. Simple design representations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models.

3. Interactiv ity

- Interactivity refers to the degree to which the human is "integrated" with automated tools to create an effective reverse-engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.

4. Directionalit.y

- If the directionality of the reverse-engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a re-engineering tool that attempts to restructure or regenerate the old program.

5. Extract Abstractions

- The core of reverse engineering is an activity called extract abstractions. The engineer must evaluate the old program and from the (often undocumented) source code, develop a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

13.7 Restructuring

See the section 13.5.2 (5) about Restructuring.

13.7.1 Advantages of Restructuring

- Improved program and documentation quality.
- Makes programs easier to learn.
- Improves productivity.
- Reduces developer frustration.
- Reduces effort required to maintain software.
- Software is easier to test and debug.

13.7.2 Types of Restructuring

The major classification of Software Restructuring are classified into-

Code restructuring

- program logic modeled using Boolean algebra and series of transformation rules are applied to yield restructured logic
- create resource exchange diagram showing
  - data types
  - procedures
  - 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

13.7.3 Forward Engineering

Chikofsky and Cross (1990) calls conventional development forward engineering to distinguish it from software re-engineering. This distinction is illustrated in the figure. Forward engineering starts with a system specification and involves the design and implementation of a new system. Re-engineering starts with an existing system and the development process for the replacement is based on the understanding and transformation of the original system.

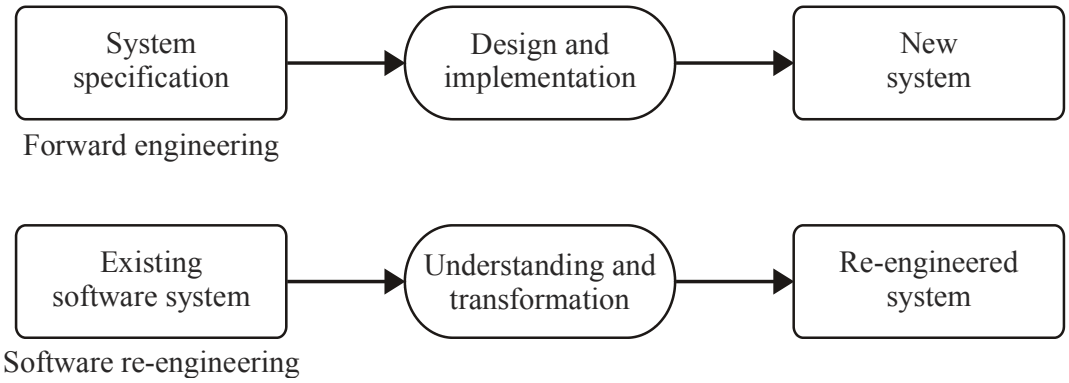


Fig. 13.1

13.8 Summary

Software industry has a lot of legacy programs that have been used and maintained for several years. These programs are important and valuable form the business point of view, because despite their age they are still in use. a-low-ever mere maintenance of these legacy programs is not always adequate; instead they require larger modernization and re-engineering. After this modernization future maintaining actions become easier. The other goals of modernization are to make programs more compact and structural, and easier to understand and reuse. Modernization is a way to improve software quality.

The process of software re-engineering must incorporate techniques for manipulating software which is imperative, declarative, or functional in nature. This generality needs a mechanism for deriving the original requirements of the underlying data structures contained within the source code itself. A reference model is proposed from which it is possible to derive the necessary techniques required to implement such a mechanism. The proposed model, the source code re-engineering reference model (SCORE/RM), consists of eight layers (encapsulation, transformation,

normalization, interpretation, abstraction, causation, regeneration, and certification) which provide a step-by-step approach to rationalizing the software. comprehending its functions and requirements, and rebuilding it according to well-established practices of software engineering. Such a framework can then be used to provide improved maintenance capability, language translation, retrospective standardization of the code and documentation, code migration onto parallel arch itechlres, and formal specification.

13.9 Self Assessment Questions

1. Under what circumstances do you think that software should be scrapped and re-written rather than re-engineered?
2. Write a set of guidelines that may be used to help find modules in an unstructured program.
3. Explain why it is impossible to recover a system specification by automatically analysing system source code.

13.10 References

- Roger S. Pressman, Software Engineering : Practioners approach, 6th Edition, The McGrawhill Companies, New York.
- Sommerville, I (2000), Software Engineering, 6th Edition, Pearson
- Pratap K.J. Mohpatra, Software Engineering, A lifecycle Approach, Ist Edition, New age Informational Publishers.

Unit - 14 : CASE (Computer Aided Software Engineering)

Structure of the Unit

14.0 Objectives

The objective of this chapter is to know about the Computer Aided Software engineering and explain the architecture. When you have read this chapter, you will:

- Understand why CASE is needed and know the objectives of its.
- Understand the various CASE tools and how these tools are used in software development process.

14.1 Introduction

CASE (Computer Aided Software Engineering) technologies are tools that provide the assistance to the developer in the development of software. The main purpose of the CASE tools is to decrease the development time and increase the quality of software.

CASE tools assist software engineering managers and practitioners in every activity associated with the software process. They automate project management activities; manage all work products produced throughout the process, and assist engineers in their analysis, design, coding and testing work. CASE tools can be integrated within a sophisticated environment.

It is the use of a computer-assisted method to organize and control the development of software, especially on large, complex projects involving many software components and people. Using CASE allows designers, code writers, testers, planners, and managers to share a common view of where a project stands at each stage of development. CASE helps ensure a disciplined, check-pointed process.

It is the integration of software-based modelling tools into the software development process. Analysis and design methodologies and modelling notations were developed to formalize the software engineering process; CASE tools automate that process by assisting in each step. Some types of CASE tools are analysis and design tools, automated code generation tools, and software testing tools. Analysis and design tools aid in the creation of diagrams and project documentation. Automated code generation assists in the implementation phases. Testing tools lead to a more thorough evaluation of an application.

14.2 Need of CASE

Software developers always looking for such CASE tools that help them in many different ways during the different development stages of software, so that they can understand the software and prepare a good end product that efficiently fulfil the user requirements. These tools provide the ways that can fulfil this requirement of software developers and provide the computerized setting to software developers to analyze a problem and then design its system model.

14.3 Objectives of CASE

The major objectives of CASE are as follows-

14.3.1 Improve Productivity

Most organizations use CASE to increase the speeds with which systems are designed and developed. Imagine the difficulties carpenters would face without hammers and saws. Tools increase the analysts 'productivity by reducing the time needed to document, analyze, and construct an information system.

14.3.2 Improve Information System Quality

When tools improve the processes, they usually improve the results as well. They:

- Ease and improve the testing process through the use of automated checking.
- Improve the integration of development activities via common methodologies.
- Improve the quality and completeness of documentation.
- Help standardize the development process.
- Improve the management of the project.
- Simplify program maintenance.
- Promote reversibility of modules and documentation.
- Shorten the overall construction process.
- Improve software portability across environments. Through reverse engineering and re-engineering, CASE products extend the files of existing systems.

Despite the various driving forces (objectives) for the adoption of CASE, there are many resisting forces that also preclude many organizations from investing in CASE

14.3.3 Improve Effectiveness

Effectiveness means doing the right task (i.e., deciding the best task to perform to achieve the desired result). Tools can suggest procedures (the right way) to approach a task. Identifying user requirements, stating them in an understandable form, and communicating them to all interested parties can be an effective development process.

14.3.4 Organizations Reject CASE Because:

- The start-up cost of purchasing and using CASE.
- The high cost of training personnel.
- The big benefits of using CASE come in the late stages of the SDLC.
- CASE often lengthens the duration of the early stage of the project.
- CASE tools cannot easily share information between tools.
- Lack of methodology standards within organizations. CASE products force analysts to follow a specific methodology for system development.
- Lack of confidence in CASE products.
- IS personnel view CASE as a threat to their job security.

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

14.4 What are CASE Tools ?

Computer Aided Software Engineering (CASE) tools are gradually becoming popular for the development of software as they are improving in the capabilities and functionalities and are proving to be beneficial for the development of quality software. But, what are the CASE tools? And how do they support the process of development of software?

CASE tools are the software engineering tools that permit collaborative software development and maintenance. CASE tools support almost all the phases of the software development life cycle such as analysis, design, etc., including umbrella activities such as project management, configuration management etc.

In general, it support standard software development methods such as Jackson Structure programming or structured system analysis and design method. The CASE tools follow a typical process for the development of the system, for example, for developing data base application, CASE tools may support the following development steps:

- Creation of data flow and entity models
- Establishing a relationship between requirements and models
- Development of top-level design
- Development of functional and process description
- Development of test cases.

14.4.1 Benefits of CASE Tools

Every program you create using the Program Generator automatically includes and uses the functionality, 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
- Processing options

This functionality is consistent across all applications you generate because it is built into the Program Generator and Master Source.

The major benefits of CASE Tools are-

1. They provides better perceptive of system.
2. Facilitates communication among team members.
3. Tools are more effective for large scale systems and immense projects.
4. These tools provide visibility of processes and logic.
5. These tools improve quality and productivity of software.
6. These tools reduce the time for error correction and maintenance.
7. These tools provide clear readability and maintainability of the system.

14.5 Levels of CASE

There are three different levels of CASE technology:

1. **Production Process Support Technology**

   This includes support for process activities, such as specification, design, implementation, testing, and so on.

2. **Process Management and Technology**

   This includes tools to support process modelling and process management. These tools are used for specific support activities.

3. **Meta-CASE Technology**

   Meta-CASE tools are generators, which are used to create production process-management support tools

14.6 Lists of CASE Tools

Most of the CASE tools include one or more of the following types of tools:

- Analysis tools
- Repository to store all diagrams, forms, models and report definitions etc.
- Diagramming tools
- Screen and report generators
- Code generators
- Documentation generators
- Reverse Engineering tools (that take source code as input and produce graphical and textual representations of program design-level information)
- Re-engineering tools (that take source code as the input analyse it and interactively alters an existing system to improve quality and/or performance).

14.6.1 Taxonomy of CASE Tools

1. **Business process engineering tools**

   This tool is used to model the business information flow. It represents business data objects, their relationships and how data objects flow between different business areas within a company.

2. **Process modelling and management tools**

   It models software processes. First the processes need to be understood then only it could be modelled. This tool represents the key elements of the processes. Hence it is possible to carry out work tasks efficiently.

3. **Project planning tools**

   These tools help to plan and schedule projects. Examples are PERT and CPM. The objective of this tool is finding parallelism and eliminating bottlenecks in the projects.

4. **Risk analysis tools**

   It helps in identifying potential risks. These tools are useful for building the risk table and thereby providing details guidance in identification and analysis of risks. Using this tool one can categorize as catastrophic, critical, marginal, or negligible. A cost is associated with each risk which can be calculated at each stage of development.

5. **Metrics and management tools**

   These tools assist to capture specific metrics that provide an overall measure of quality. These tools are intended to focus on process and product characteristics.

6. **Project Management tools**

   These track the progress of the project. These tools are extension to the project planning tools and the use of these tools is to update plan s and schedules.

7. **Requirements tracing tools**

   The objective of these tools is to provide a systematic approach to isolate customer requirements and then to trace these requirements in each stage of development.

8. **Documentation tools**

   Most of the software development organizations spend lot of time in developing the documents. For this reason documentation tools provide a way to develop documents efficiently. For example- word processor that gives templates for the organization process documents.

9. **System software tools**

   These tools provide services to network system software object management and distributed component support. For example- email, bulletin boards etc.

10. **Quality assurance tools**

    These tools are actually metrics tools that audit source code to insure compliance with language standards. The other CASE tools are-

- Database Management tool
- Analysis and design tool
- Prototype tools
- Programming tools
- Interface design and development tool.

14.7 Features of CASE tools

- It should have Security of information. The information may be visible/ changeable by authorised users only.
- Version Control for various products.
- A utility to Import/Export information from various external resources in a compatible fashion
- The process of Backup and Recovery as it contains very precious data.

14.8 Categories of CASE Tools

CASE tools are divided into the following two categories:

1. Vertical CASE tools
2. Horizontal CASE tools

14.8.1 Vertical CASE Tools

Vertical CASE tools provide support for certain activities within a single phase of the software life-cycle. There are two subcategories of vertical CASE tools:

(i) First Category

It is the set of tools that are within one phase of the life-cycle. These tools are important so that the development in each phase can be done as quickly as possible.

(ii) Second Category

It is a tool that is used in more than one phase, but does not support moving from one phase to the next. These tools ensure that the developer does move onto the next phase as appropriate.

14.8.2 Horizontal CASE Tools

These tools support the automated transfer of information between the phases of a life-cycle. These tools include project management, configuration-management tools, and integration services.The above two categories of CASE tools can further be broken down into the following:-

1. Upper CASE tools
2. Lower CASE tools
3. Integrated CASE tools
4. Reverse-engineering tools

Upper CASE Tools: Upper CASE tools mainly focus on the analysis and design phases of software development. They include tools for analysis modelling, reports and forms generation.

Lower CASE: Lower CASE tools support implementation of system development. They include tools for coding, configuration management, etc.

Integrated CASE Tools: Integrated CASE tools help in providing linkages between the lower and upper CASE tools. Thus creating a cohesive environment for software development where programming by lower CASE tools may automatically be generated for the design that has been developed in an upper CASE tool.

Reverse-engineering Tools: These tools build bridges from lower CASE tools to upper CASE tools. They help in the process of analyzing existing applications and perform and database code to create higher level representations of the code.

14.9 Need of CASE Tools

The software development process is expensive and as the projects become more complex in nature, the project implementations become more demanding and expensive. The CASE tools provide the integrated homogenous environment for the development of complex projects. They allow creating a shared repository of information that can be utilised to minimise the software development time. The CASE tools also provide the environment for monitoring and controlling projects such that team leaders are able to manage the complex projects. Specifically, the CASE tools are normally deployed to –

- Reduce the cost as they automate many repetitive manual tasks.
- Reduce development time of the project as they support standardisation and avoid repetition and reuse.
- Develop better quality complex projects as they provide greater consistency and coordination.
- Create good quality documentation
- Create systems that are maintainable because of proper control of configuration item that support traceability requirements. But please note that CASE tools cannot do the following:
- Automatic development of functionally relevant system
- Force system analysts to follow a prescribed methodology
- Change the system analysis and design process. There are certain disadvantages of CASE tools. These are:

- Complex functionality
- Many project management problems are not amenable to automation. Hence, CASE tools cannot be used in such cases.

14.10 Architecture of CASE Environments

The architecture of the CASE environment is illustrated in the following figure. The important components of a modern CASE environment are the user interface, the Tools Management System (tools-set), the Object Management System (OMS), and the repository. These various components are discussed as follows:

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

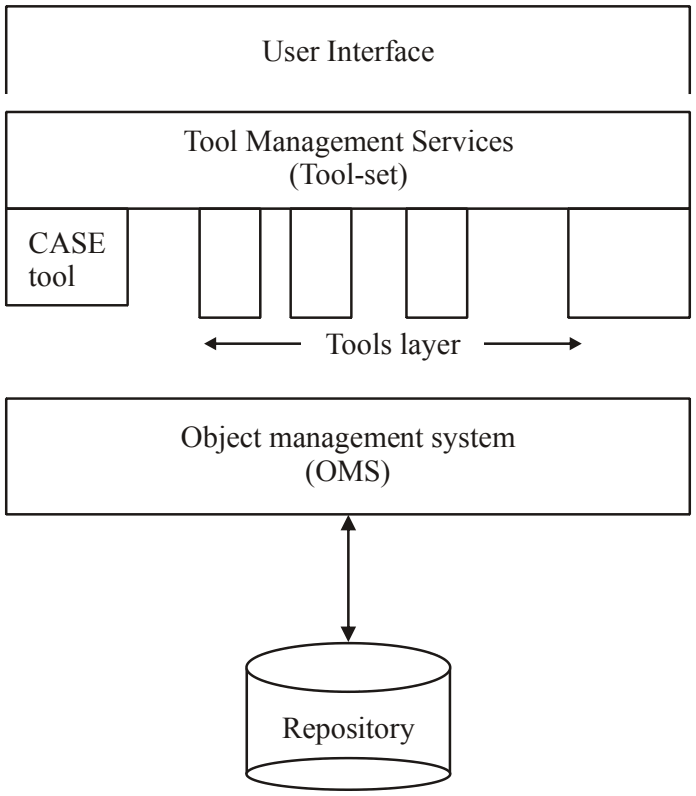


Fig. 14.1 Tools Management Servies (Tools set)

14.10.2 Tools-Management Services (Tools-set)

The tools-set section holds the different types of improved quality tools. The tools layer incorporates set of tools-management services with the CASE tool themselves. The Tools Management Service (TMS) controls the behaviour of tools within the environment. If multitasking is used during the execution of one or more tools, TMS performs multitask synchronization and communication, coordinates the flow of information from the repository and object-management system into the tools, accomplishes security and auditing functions, and collects metrics on tool usage.

14.10.2 Object-Management System (OMS)

The object-management system maps the (specification design, text data, project plan, etc.) logical entities into the underlying storage-management system, i.e., the repository. Working in conjunction with the CASE repository, the OML provides integration services a set of standard modules that couple tools with the repository. In addition, the OML provides configuration management services by enabling the identification of all configuration objects performing version control, and providing support for change control, audits, and status accounting.

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

14.11 Summary

CASE (computer-aided software engineering) is the use of a computer-assisted method to organize and control the development of software, especially on large, complex projects involving many software components and people. Using CASE allows designers, code writers, testers, planners, and managers to share a common view of where a project stands at each stage of development. CASE helps ensure a disciplined, check-pointed process. A CASE tool may portray progress (or lack of it) graphically. It may also serve as a repository for or be linked to document and program libraries containing the project's business plans, design requirements, design specifications, detailed code specifications, the code units, test cases and results, and marketing and service plans. To speed up the software system building process, a new concept of designing software is introduced in the '70s, called Computer Aided Software Engineering (CASE). This term is used for a new generation of tools that applies rigorous engineering principles to the development and analysis of software specifications. Simply, computers develop software for other computers in a fast way by using specific tools.

14.12 Self Assessment Questions

1. What is computer- Aided software Engineering ?
2. What are the criteria should one consider in a CASE tool ?
3. Describe two ways in which system building tools can optimise the process of building a version of a system from its components.
4. Explain the advantages and disadvantages of using CASE tools
5. How are CASE tools classified ?
6. Explain CASE repository in detail ?

14.13 References

- Roger S. Pressman, Software Engineering : Practioners approach, 6th Edition, The McGrawhill Companies, New York.
- Sommerville, I (2000), Software Engineering, 6th Edition, Pearson
- Pratap K.J. Mohpatra, Software Engineering, A lifecycle Approach, Ist Edition, New age Informational Publishers.