GOVERNMENT OF TAMILNADU
DIRECTORATE OF TECHNICAL EDUCATION
CHENNAI – 600 025

STATE PROJECT COORDINATION UNIT

Diploma in Computer Engineering

Course Code: 1052

M – Scheme

e-TEXTBOOK
on
SOFTWARE ENGINEERING

V Semester

Convener for Computer Enginnering Discipline:

Mr. D.Arulselvan,
HOD /Post Diploma in Computer Applications
Thiagarajar Polytechnic College,
Salem - 636 005.

Team Members for Software Engineering:

Mr. D.Arulselvan,
HOD /Post Diploma in Computer Applications
Thiagarajar Polytechnic College,
Salem - 636 005.

Mrs. V. Saranya
HoD / Computer Engineering,
Thiagarajar Polytechnic College,
Salem – 636 005.

Ms. M.Gomathi,
Lecturer/ Computer Engineering
Thiagarajar Polytechnic College,
Salem – 636 005.

Validated by

Mr. M. Suresh Babu
HOD / Computer Engineering
N.P.A. Centenary Polytechnic College
Kotagiri

STATE BOARD OF TECHNICAL EDUCATION & TRAINING, TAMILNADU.

DIPLOMA IN COMPUTER ENGINEERING

M- SCHEME

Course Name : Diploma in Computer Engineering.

Subject Code : 35272

Semester : V

Subject title : SOFTWARE ENGINEERING

TEACHING & SCHEME OF EXAMINATION

No. of weeks per Semester: 15 Weeks

Subject	Instructions		Examination			Duration
	Hours / Week	Hours / Semester	Internal Assessment	Board Examination	Total	
SOFTWARE ENGINEERING	4	60	25	75	100	3 Hrs

TOPICS AND ALLOCATION OF HOURS

Unit No	Topic	No of Hours
I	INTRODUCTION TO SOFTWARE ENGINEERING	**10**
II	SOFTWARE DESIGN AND PLANNING	**10**
III	SOFTWARE MAINTENANCE AND RISK MANAGEMENT	**10**
IV	SOFTWARE TESTING	**10**
V	SOFTWARE RELIABILTY AND QUALITY ASSURANCE	**10**
TEST AND REVISION		**10**
		60

RATIONALE

Software Engineering deals with reliability and quality assurance of the software under development. It provides framework for development of quality software product. The course enables the students to write specifications for software system understand the importance of good software, design and develop test plans from design specifications. The course also covers other important aspects of software Engineering such as software lifecycle, requirement analysis and documentation, characteristics of good design, design techniques, testing, software implementation and maintenance etc.

OBJECTIVES

On completion subject, the students must be able to

- Define Software Engineering.
- Understand the characteristics of Software Engineering.
- Explain different software development models.
- Learn about the phases of software development cycle.
- Understand the significance of requirement analysis.
- Know various tools and techniques used for requirement analysis.
- Understand architectural and modular design.
- Understand the different types of project metrics.
- Understand different software estimation techniques.
- Describe CASE.
- Explain about software maintenance.
- Need for software maintenance.
- Identify and mange risks.
- Know the different scheduling methods.
- Define the basic terms used in testing terminology.
- Describe black box and white box testing.
- Describe testing tools.
- Understand the concepts of Software quality and quality assurance.
- Know the concepts of software reliability and software quality standards.
- Define software re-engineering.
- Differentiate forward engineering from re-engineering.

DETAILED SYLLABUS

UNIT I INTRODUCTION TO SOFTWARE ENGINEERING .. 10 HOURS		
1.1	**Basics of Software Engineering:** Need for Software Engineering – Definition – Software Characteristics – Software Myths – Program versus Software Products.	2 Hrs
1.2.	**Software Development Life Cycle Models:** Introduction –- Waterfall Model – Prototyping model – Spiral Model – Iterative Enhancement model - RAD model – Object Oriented Model - Advantages and Disadvantages of above models – Comparison of various models.	4 Hrs
1.3	**Software Requirement Analysis (SRS):** Value of good SRS – Requirement Process – Requirement Specification – Desirable characteristics of an SRS – Components of an SRS – Structures of a requirements documents - Problems in SRS – Requirements gathering	4 Hrs
UNIT – II SOFTWARE DESIGN AND PLANNING .....10 HOURS		
2.1.	**Software Design:** Definition of software design – Objectives of software design – Process of software design – Architectural design – Modular design – Structure chart – Coupling and Cohesion – Different types – Interface design – Design of Human Computer Interface	3 Hrs

2.2.	**CODING:** Information Hiding – Programming style – Internal documentation – Monitoring and Control for coding – Structured programming	2 Hrs
2.3.	**Software Planning:** Software metrics- Definition – Types of metrics – Product and Project metrics – Function point and feature point metrics – Software project estimation – Steps for estimation – Reason for poor and inaccurate estimation – Project estimation guidelines – Models for estimation – COCOMO Model – Automated tools for estimation.	3 Hrs
2.4.	**CASE:** CASE and its scope – Architecture of CASE environment – Building blocks for CASE – CASE support in software Life cycle – Objectives of CASE – Characteristics of CASE tools – List of CASE tools – Categories, advantages and advantages of CASE tools.	2 Hrs
UNIT – III	**SOFTWARE MAINTENANCE AND RISK MANAGEMENT**	**..... 10 HOURS**
3.1.	**Software Maintenance:** Software as an evolution entity – Software configuration management activities – Change control process – Software version control – Software configuration management – Need for maintenance – Categories of maintenance – Maintenance cost – Factors affecting the effort – Modelling maintenance effort	5 Hrs
3.2.	**Risk management:** Definition of risk – Basics for different types of software risks – Monitoring of risks – Risk management – Risk avoidance – Risk detection – Risk control – Risk recovery – Sources of risks – Types of risks	3 Hrs
3.3.	**Project scheduling**: Introduction – Factors affecting the task set for the project – scheduling methods – Work breakdown structure – Flow graph – Gant chart - PERT	2 Hrs
UNIT – IV	**SOFTWARE TESTING**	**..... 10 HOURS**
4.1.	**Software Testing:** Introduction to testing – Testing principles – Testing objectives – Test Oracles - Basic terms used in testing – Fault – Error – Failure - Test cases – Black box and white box testing – Advantages and disadvantages of above testing – Methods for Block box testing strategies – Methods for white box testing strategies – Testing activities – Test plan.	2 Hrs
4.2.	**Levels of testing:** Unit testing - Integration tests – System testing – Types.	2 Hrs
4.3.	**Software Testing strategies:** Static testing strategies – Formal technical reviews – Code walkthrough – Code inspection - Debugging – Definition – Characteristics of bugs – Life cycle of a Debugging task – Debugging approaches.	2 Hrs
4.4	**Software Testing Tools:** Need for tools – Classification of tools – Functional/Regression Testing tools – Performance/Load Testing Tools – Testing process management Tools – Benefits of tools – Risk Associated with tools – Selecting tools – Introducing the tool in the testing process - Different categories of tools – Examples for commercial software testing tool.	2 Hrs
4.5	**Code of Ethics for Software Professionals:** Human Ethics – Professional Ethics – Ethical issues in Software Engineering – Code of Ethics and professional Practice: Software Engineering code of ethics and professional Practice – Ethical issues: Right versus Wrong.	2 Hrs

UNIT – V SOFTWARE RELIABILITY AND QUALITY ASSURANCE		10
5.1.	**Software Quality Assurance :** Verification and validation – SQA - Objectives and Goals – SQA plan - Definition of software quality – Classification of software qualities - Software quality attributes – Important qualities of software products - Importance of software quality – SEI – CMM - Five levels - ISO 9000 – Need for ISO Certification – Benefits of ISO 9000 certification – Limitation of ISO 9000 certification – Uses of ISO - Salient features of ISO 9000 Requirements – Introduction to ISO 9126	5 Hrs
5.2	**Software Reliability:** Definition – Reliability terminologies – Classification of failures – Reliability metrics – Reliability growth modeling - Reliability measurement process	2 Hrs
5.3	**Reverse Software Engineering:** Definition – Purpose - Reverse engineering Process – Reverse engineering tasks – Characteristics and application areas of reverse engineering – Software re-engineering – Principle – Re- engineering process – Difference between forward engineering and re-engineering.	3 Hrs

REFERENCES

S. No	TITLE	AUTHOR	PUBLISHER	Year of Publishing / Edition
1.	Software Engineering	Ian Sommerville	Pearson Education	Sixth Edition
2.	Fundamentals of Software Engineering	Rajib Mall	PHI Learning Pvt Limited, New Delhi	28th Printing
3.	Software Engineering	Bharat Bhusan Agarwal, Sumit Prakash Tayal	Firewall Media, New Delhi	Second Edition 2008
4.	Software Testing	K.Mustafa and R.A.Khan	Narosa Publishing House, New Delhi	Reprint 2009
5.	Software Quality	R.A. Khan, K.Mustafa and SI Ahson	Narosa Publishing House, New Delhi	Reprint 2008
6.	Software Engineering	Stephen Schach	TMGH Education Pvt Ltd, New Delhi	Eight Reprint 2011
7.	Software Testing Principles and Practices	Srnivasan desikan, Gopalswamy Ramesh	Pearson	First Edition
8.	Software Testing Concepts and Tools	Nageshwara Rao Pusulri	DreamTeach	First Edition
9.	Software Engineering Concepts and application	Subhasjit Dattun	OXFORD University Press	2010
10.	Software Engineering	Rohit Khurana	Vikas Publishing	Second Edition

INDEX

UNIT – I	

INTRODUCTION TO SOFTWARE ENGINEERING

OBJECTIVES

At the end of the unit, the students will be able to

- Define Software Engineering.
- Understand the characteristics of Software Engineering.
- Explain different software development models.
- Learn about the phases of software development cycle.
- Understand the significance of requirement analysis.
- Learn Components of SRS
- Understand the problems in SRS
- Know various tools and techniques used for requirement analysis.

1.0 INTRODUCTION

The term Software Engineering is made of two words, software and engineering. **Software** is more than just a program code. A program is an executable code, which serves some computational purpose. Software is collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product. Engineering** on the other hand, is all about developing products, using well-defined, scientific principles and methods.

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

1.1 BASICS OF SOFTWARE ENGINEERING

1.1.1 NEED FOR SOFTWARE ENGINEERING

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software** - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.
- **Scalability -** If the software processes were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost -** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

- **Dynamic Nature** - Growing and adapting nature of software hugely depends upon the environment in which user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management-** Better process of software development provides better and quality software product.

1.1.2 SOFTWARE ENGINEERING- DEFINITION

IEEE Definition: Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, i.e., the application of engineering to software.

Other Definition: Software Engineering deals with cost effective solutions to practical problems by applying scientific knowledge in building software artifacts in the service of mankind.

1.1.3 SOFTWARE CHARACTERISTICS

The software has a very special characteristic e.g., "it does not wear out". Its behavior and nature is quite different than other products of human life. Some of the important characteristics are :

(i) Software does not wear out: There is a well-known "bath tub curve" for hardware products. The curve is given in Fig. 1.1. The shape of the curve is like "bath tub"; and is known as bath tub curve.

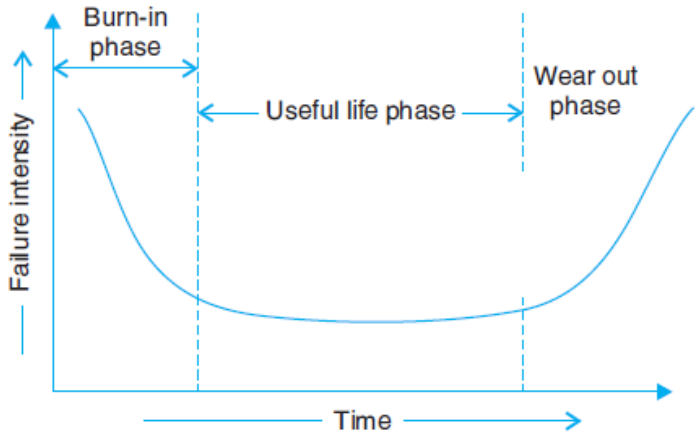


Fig No 1-1. Bath Tub Curve

There are three phases for the life of a hardware product. Initial phase is **burn-in phase**, where failure intensity is high. It is expected to test the product in the industry before delivery. Due to testing and fixing faults, failure intensity will come down initially and may stabilize after certain time. The second phase is the **useful life phase** where failure intensity is approximately constant and is called useful life of a product.

After few years, again failure intensity will increase due to wearing out of components. This phase is called **wear out phase.** The software does not have this phase, since it does not wear out. The curve for software is given in figure 1-2.

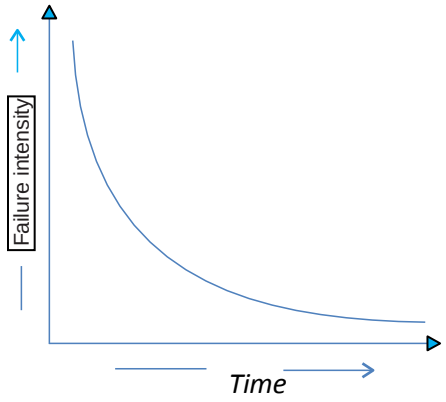


Fig No 1-2 Software curve

Software becomes reliable overtime instead of wearing out. It becomes obsolete, if the environment for which it was developed, changes. Hence software may be retired due to environmental changes, new requirements, new expectations, etc.

(ii) Software is not manufactured: The life of software is from concept exploration to the retirement of the software product. It is one time development effort and continuous maintenance effort to keep it operational. However, making 1000 copies is not an issue and it does not involve any cost. In case of hardware product, every product costs due to raw material and other processing expenses. There is no assembly line in software development. Hence it is not manufactured in the classical sense.

(iii) Reusability of components:

In software, every project is a new project and it will start from the scratch and design every unit of the software product. Huge effort is required to develop software which further increases the cost of the software product. However, effort has been made to design standard components that may be used in new projects. Software reusability has introduced another area and is known as component based software engineering.

In the hardware world, component reuse is a natural part of the engineering process. In software, there is only a humble beginning like graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms.

(iv) Software is flexible: A program can be developed to do almost anything. Sometimes, this characteristic may be the best and may help us to accommodate any kind of change. However, most of the times, this “almost anything” characteristic has made software development difficult to plan monitor and control. This unpredictability is the basis of what has been referred to for the past 30 years as the “Software Crisis”.

1.1.4 SOFTWARE MYTHS

There are number of myths associated with software development community. Some of them really affect the way, in which software development should take place.

SOFTWARE MYTHS - MANAGEMENT

"We already have a book that is full of standards and procedures for building software. Won't that provide my people with everything they need to know?"
Not used, not up to date, not complete, not focused on quality, time, and money

"If we get behind, we can add more programmers and catch up"
Adding people to a late software project makes it later Training time, increased communication lines

"If I decide to outsource the software project to a third party, I can just relax and let that firm build it"
Software projects need to be controlled and managed

SOFTWARE MYTHS - CUSTOMER

"A general statement of objectives is sufficient to begin writing programs – we can fill in the details later"

Ambiguous statement of objectives spells disaster

"Project requirements continually change, but change can be easily accommodated because software is flexible"
Impact of change depends on where and when it occurs in the software life cycle (requirements analysis, design, code, test)

SOFTWARE MYTHS - PRACTITIONER

"Once we write the program and get it to work, our job is done"
60% to 80% of all effort expended on software occurs after it is delivered

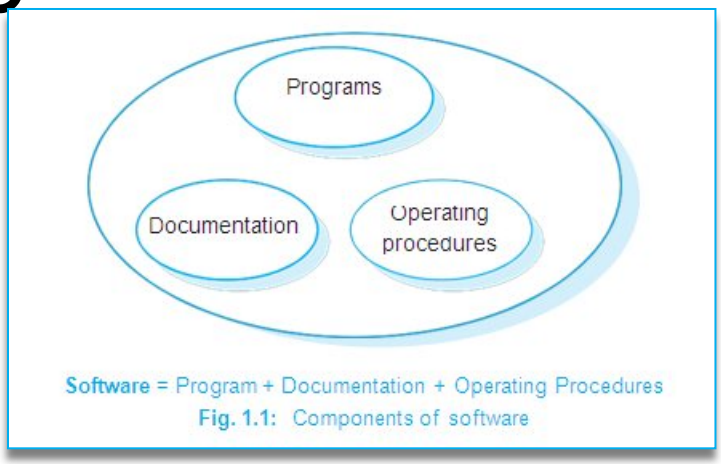
"Until I get the program running, I have no way of assessing its quality
Formal technical reviews of requirements analysis documents, design documents, and source code (more effective than actual testing)

"The only deliverable work product for a successful project is the working program"
Software, documentation, test drivers, test results

"Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down"
Creates quality, not documents; quality reduces rework and provides software on time and within the budget

1.1.5 PROGRAM VERSUS SOFTWARE PRODUCTS

Software is more than programs. It consists of programs; documentation of any facet of the program and the procedures used to setup and operate the software system. The components of the software systems are shown in Fig 1.3.



Software = Program + Documentation + Operating Procedures

Fig. 1.1: Components of software

Any program is a subset of software and it becomes software only if documentation and operating procedure manuals are prepared. Program is a combination of source code and object code. Documentation consists of different types of manuals as shown in following fig No 1-4

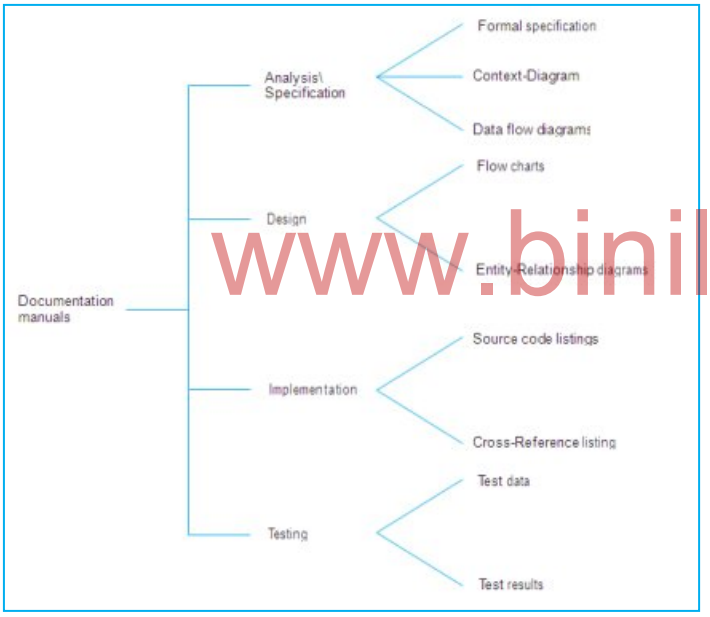


Fig No 1-4 List of documentation manuals

Operating procedures consist of instructions to setup and use the software system and instructions on how to react to system failure. List of operating procedure manuals / documents is given in figure 1.5.

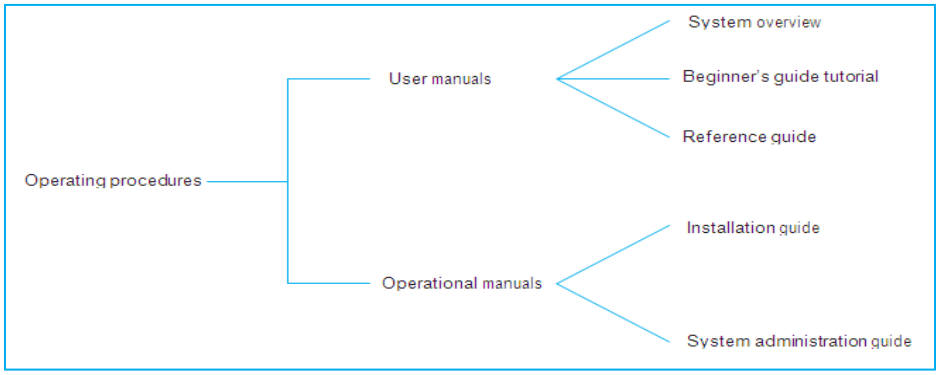


Fig No 1-5. Operating Procedures

Program versus Software Products

S.No	Programs	Software Products
1	Programs are developed by individuals for their personal use.	A software products is usually developed by a group of engineers working in a team.
2	Usually small in size.	Usually large in size.
3	Single user	Large number of users.
4	Lacks proper documentation.	Good documentation support.
5	Lack of user interface.	Good user interface.
6	Have limited functionality.	Exhibit more functionality.

GENERIC AND CUSTOMISED SOFTWARE PRODUCTS

The software products are divided in two categories:

(i) Generic products (ii) Customized products.

Generic products are developed for anonymous customers. The target is generally the entire world and many copies are expected to be sold. Infrastructure software like operating systems, compilers, analyzers, word processors, CASE tools etc. are examples for this category.

The customized products are developed for customers. The specific product is designed and developed as per customer requirements. Most of the development projects (say about 80%) comes under this category.

1.2 SOFTWARE DEVELOPMENT LIFE CYCLE MODELS

1.2.1 INTRODUCTION

Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software engineering to develop the intended software product.

1.2.2 WATERFALL MODEL

Waterfall model is the simplest model of software development paradigm and was popularized in the 1970s. The waterfall model, Illustrated in fig 1.6. Because of the cascade from one phase to another, this model is known as the *waterfall model or software life cycle.*

The process is structured as a cascade of phases, where the output of one phase constitutes the input to the next one. All the phases of SDLC will function one after another in linear manner. That is, when the first phase is finished then only the second phase will start and so on.

The phases of waterfall model are:

- Requirements gathering
- System Analysis and Design
- Coding
- Testing
- Integration and Implementation
- Operation and Maintenance

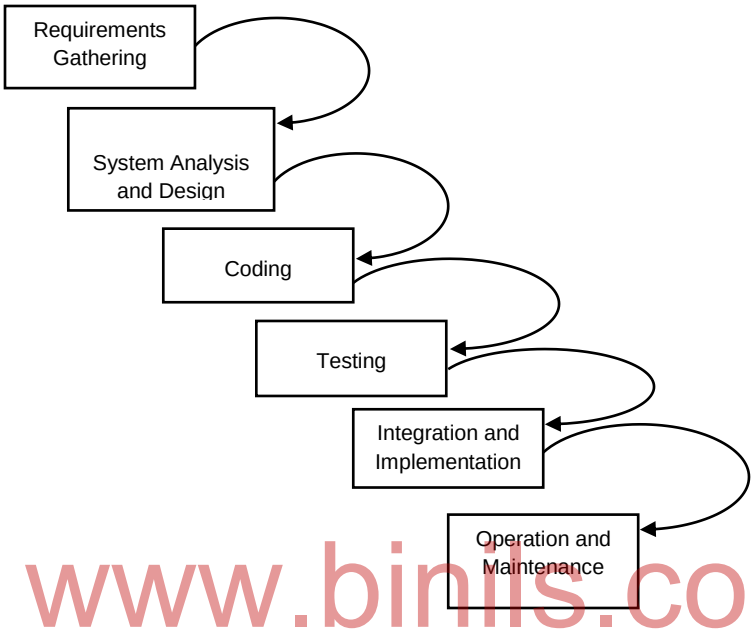


Fig No 1-6 Stages of Waterfall Model

Requirement Gathering

The team holds discussions with various stakeholders from problem domain and tries to bring out as much information as possible on their requirements. The requirements are contemplated and segregated into user requirements, system requirements and functional requirements. The requirements are collected using a number of practices as given

- studying the existing or obsolete system and software,
- conducting interviews of users and developers,
- referring to the database or
- collecting answers from the questionnaires.

System Analysis and Design

At this step the developers decide a roadmap of their plan and try to bring up the best software model suitable for the project. System analysis includes understanding of software product limitations, learning system related problems or changes to be done in existing systems. The project team analyzes the scope of the project and plans the schedule and resources accordingly.

Next step is to bring down whole knowledge of requirements and analysis on the desk and design the software product. The inputs from users and information gathered in requirement gathering phase are

the inputs of this step. The output of this step comes in the form of two designs; logical design, and physical design. Engineers produce meta-data and data dictionaries, logical diagrams, data-flow diagrams, and in some cases pseudo codes.

Coding

This step is also known as programming phase. The implementation of software design starts in terms of writing program code in the suitable programming language and developing error-free executable programs efficiently.

Testing

50% of whole software development process should be tested. Errors may ruin the software from critical level to its own removal. Software testing is done while coding by the developers and thorough testing is conducted by testing experts at various levels of code such as module testing, program testing, product testing, in-house testing, and testing the product at user's end.

Integration and Implementation

Software may need to be integrated with the libraries, databases, and other program(s). This stage of SDLC is involved in the integration of software with outer world entities. Implementation means installing the software on user machines. At times, software needs post-installation configurations at user end. Integration related issues are solved during implementation.

Operation and Maintenance

This phase confirms the software operation in terms of more efficiency and less errors. If required, the users are trained on, or aided with the documentation on how to operate the software and how to keep the software operational. The software is maintained timely by updating the code according to the changes taking place in user end environment or technology. This phase may face challenges from hidden bugs and real-world unidentified problems.

This model assumes that everything is carried out and taken place perfectly as planned in the previous stage and there is no need to think about the past issues that may arise in the next phase. This model does not work smoothly if there are some issues left at the previous step. The sequential nature of model does not allow us to go back and undo or redo our actions. This model is best suited when developers already have designed and developed similar software in the past and are aware of all its domains.

Advantages of Waterfall Model

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model.
- Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Easy to arrange tasks.
- Process and results are well documented

Disadvantages of Waterfall Model

- It is difficult to define all requirements at the beginning of a project.
- This model is not suitable for accommodating any change.
- A working version of the system is not seen until late in the project's life.
- It involves heavy documentation.
- There is no risk analysis.
- If there is any mistake or error in any phase then it is not possible to make good software.
- It is a document driven process that requires formal documents at the end of each phase.

Reasons for the failure of water fall Model

- Real Project rarely follow Sequential Flow. Iterations are made in indirect manner
- Difficult for customer to state all requirements explicitly
- Customer needs more patients as working product reach only at Deployment phase

1.2.3 PROTOTYPING MODEL

The prototyping model is applied when detailed information related to input and output requirements of the system is not available. In this model, it is assumed that all the requirements may not be known at the start of the development of the system. It is usually used when a system does not exist or in the case of a large and complex system where there is no manual process to determine the requirements. This model allows the users to interact and experiment with a working model of the system known as prototype. The prototype gives the user an actual feel of the system.At any stage, if the user is not satisfied with the prototype, it can be discarded and an entirely new system can be developed. Generally, a prototype can be prepared by the following approaches:

- By creating main user interfaces without any substantial coding so that the users can get a feel of how the actual system will appear
- By abbreviating a version of the system that will perform limited subsets of functions
- By using system components to illustrate the functions that will be included in the system to be developed.

The fig 1.7. Illustrates the steps carried out in the prototyping model.

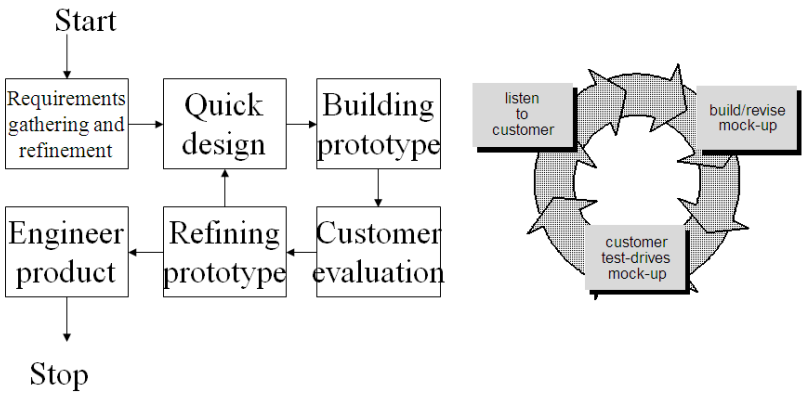


Fig No 1-7. Prototype Model

1. **Requirements gathering and analysis:** A prototyping model begins with requirements analysis and the requirements of the system are defined in detail. The user is interviewed in order to gather the requirements of the system.
2. **Quick design:** When the requirements are known, a preliminary design or quick design for the system is created. It is not a detailed design and includes only the important aspects of the system which gives an idea of the system to the user. A quick design helps in developing the prototype.
3. **Prototype building**: Information gathered from the quick design is modified to form the first prototype, which represents the working model of the proposed system.
4. **User evaluation:** Next, the proposed system is presented to the user for a thorough evaluation of the prototyping to recognize its strengths and weaknesses. Comments and suggestions are collected from the users and provided to the developer.
5. **Prototype refining**: Once the user evaluates the prototype and if he is not satisfied, the current prototype is refined according to the requirements. That
    a. Is, a new prototype is developed based on the feedback provided by the user. The new prototype is evaluated just like the previous protype. This process is continuing until all the requirements specified by the user are met.
    b. Once the user is satisfied with the developed prototype, a final system is developed on the basis of the final prototype.
6. **Engineer product:** Once the requirements are completely met, the user accepts the final prototype. The final system is evaluated thoroughly, followed by routine maintenance for preventing large-scale failures and for minimizing downtime.

Reasons for Using Prototyping Model

- The prototype model is very useful in developing the Graphical User Interface (GUI) part of a system.
- The prototyping model can be used when the technical solutions are unclear to the development team.
- The third reason for developing a prototype is that it is impossible to “get in right” the first time and one must plant to throw away the first product in order to develop a good product later, as advocated.

Advantages of Prototyping Models

- Suitable for large system for which there is no manual process to define the requirements.
- User training to use the system.
- User service determination
- System training
- Quality of software is good
- Requirements are not free zed.

Limitations of Prototyping Model

- It was difficult to find all the requirements of the software initially.

1.2.4 SPIRAL MODEL

The spiral model starts with an initial pass through a standard waterfall lifecycle, using a subset of the total requirements to develop a robust prototype. After an evaluation period, the cycle is initiated again, adding new functionality and releasing the next prototype. This process continues, with the prototype becoming larger and larger with each iteration. Hence, the "spiral."

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

The objective of the spiral model is to help the management to evaluate and resolve risk in the software project. Different areas of the risks in the software project are project overruns, changing , requirement, loss of key project personnel, delay necessary hardware, competition with other software development and technological breakthroughs which make the project obsolete.

The steps involved in the Spiral model are:

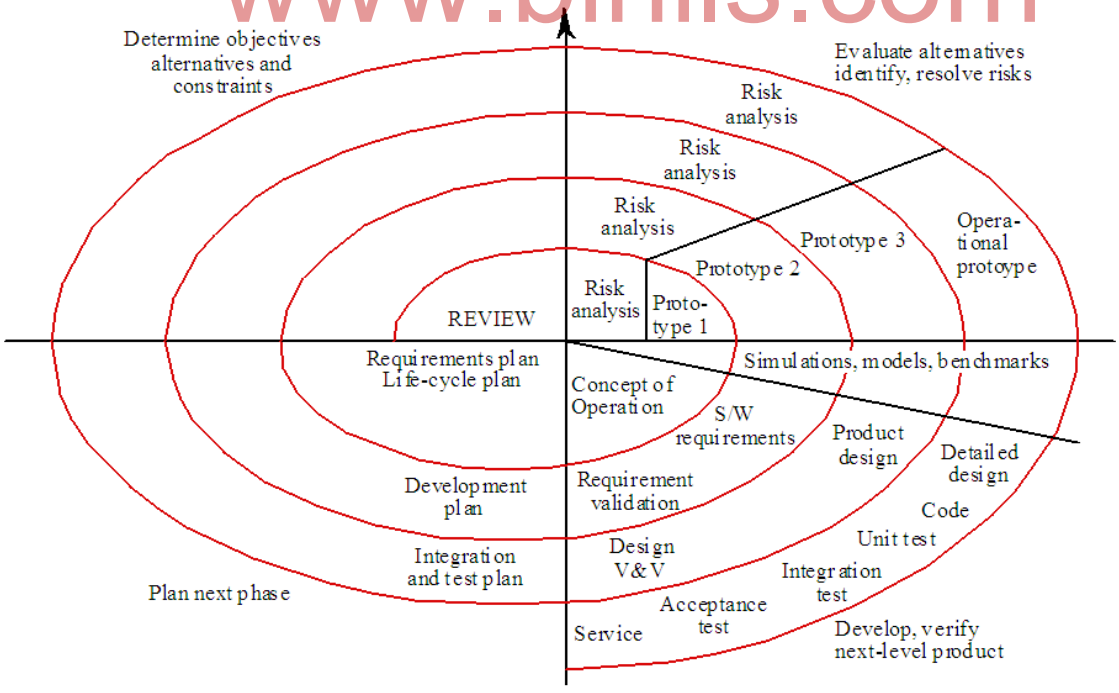


Fig No 1-8. Spiral Model

1. Each cycle of the quadrant commences with identifying the goals for that cycle. In addition, it determines other alternatives, which are possible in accomplishing those goals.
2. The next step in the cycle evaluates alternatives based on objectives and constraints. This process identifies the area of uncertainty and focuses on significant sources of project risks. Risk signifies that there is a possibility that the objectives of the project would not be accomplished. If so, the formulation of a cost-effective strategy for resolving the risks is followed.
3. The development of the software depends on managing the remaining risks. The third quadrant develops the final software while considers the time and effort to be devoted to each project activity, such as planning, configuration management, and quality assurance verification and testing.
4. The last quadrant plans the next step, and includes planning for the next prototype and thus, comprises the requirement plan, development plan, integration plan and test plan.

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

The spiral model is like the waterfall model as software requirements are understood at the early stages in both the models. However, the major risks involved in detailed design of the software are developed. Processes in the waterfall model are followed by different cycles in the spiral model.

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

1.2.5 ITERATIVE ENHANCEMENT MODEL

The iterative enhancement model combines linear sequential model (applied repetitively) with the prototyping. In this model, the software is broken down into several modules, which are incrementally developed and delivered. First, the development team develops the core module of the system and then it is later refined into increasing levels of capability of adding new functionalities in successive version.

Each linear sequence produces a deliverable *increment* of the software. For example, word processing software developed using the iterative paradigm might deliver basis file management, editing, and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. The process flow for any increment could incorporate the prototyping paradigm.

When an iterative enhancement model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features remain undelivered. The core product is used by the customer (or undergoes detailed review). As a result of evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

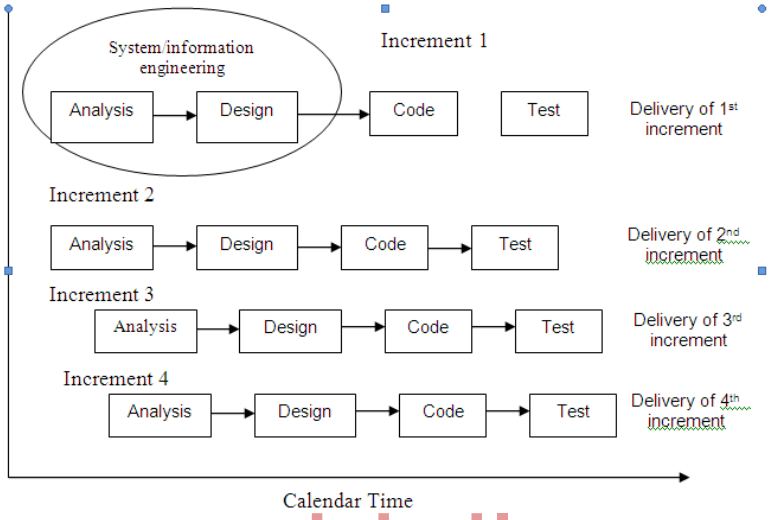


Fig No 1-9. Iterative Enhancement Model

Advantages of iterative enhancement model

- The possibilities of changes in requirement are reduced because of the shorter time span between the design of a component and its delivery.
- Users get benefits earlier than with a conventional approach.
- Early delivery of some useful components improves cash flow, because you get some return on investment early on.
- Smaller subprojects are easier to control and manage.
- The project can be temporally abandoned if more urgent work crops up.

Disadvantages of Iterative Enhancement Model

- Software breakage, that is , later increments may require modifications to earlier increments.
- Programmers may be more productive working on one large system than on a series of smaller ones.

1.2.6 RAD MODEL

Rapid Application Development Model is similar to incremental model and waterfall model. In RAD Model, development should be done in specified time frame. RAD Model is suitable for the small project where

all the requirements are gathered before starting development of the project and no any concrete plan required. Development starts as soon as requirement is gathered and initial working prototype is delivered to the client to get the feedback. Once client gives the feedback, other changes are done. This process goes parallel with co-operation with client and developers. Each prototype is delivered to the client with working functionality and changes made based on the client's feedback. Development moves faster in RAD Model with minimum errors. RAD Model follows the incremental delivery of the modules. The main goal of RAD Model is to make the reusability of the developed components.

Phases in RAD Model:

- Business Modeling
- Data Modeling
- Process Modeling
- Application Modeling
- Testing and Turnover

- **Business Modeling:** In this phase of development business model should be designed based on the information available from different business activities. Before start the development there should be a complete picture of business process functionality.
- **Data Modeling:** Once the business modeling phase over and all the business analysis completed, all the required and necessary data based on business analysis are identified in data modeling phase.
- **Process Modeling:** All the data identified in data modeling phase are planned to process or implement the identified data to achieve the business functionality flow. In this phase all the data modification process is defined.

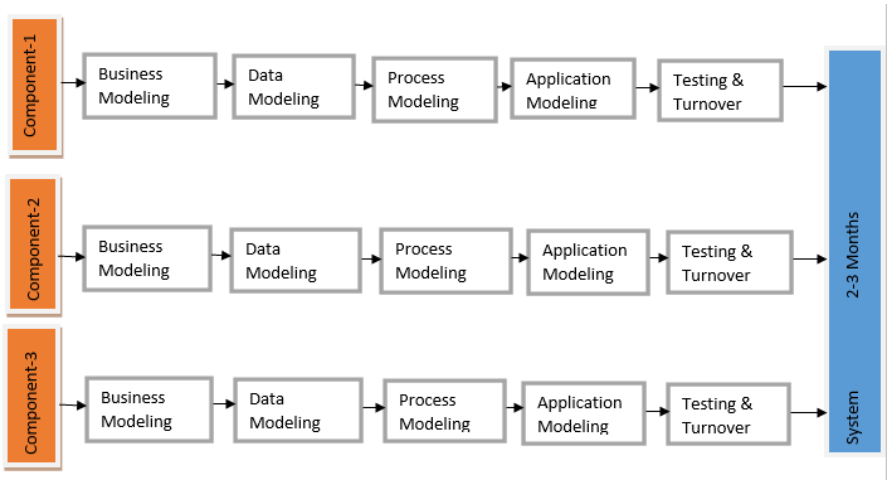


Fig No 1-10. RAD Enhancement Model

Application Modeling: In this phase application id developed and coding completed. With help of automation tools all data implemented and processed to work as real time.

Testing and turnover: All the testing activates are performed to test the developed application.

Advantages of RAD Model:

- Fast application development and delivery.
- Less testing activity required.
- Visualization of progress.
- Less resource required.
- Review by the client from the very beginning of development so very less chance to miss the requirements.
- Very flexible if any changes required.
- Cost effective and good for small projects.

Disadvantages of RAD Model:

- High skilled resources required.
- On each development phase client's feedback required.
- Automated code generation is very costly.
- Difficult to manage.
- Not a good process for long term and big projects.
- Proper modularization of project required.

When RAD Model should be followed:

- For low budges projects for which automated code generation cost is low.
- When high skilled resources are available.
- When it's easy to modularize the project.
- If technical risks are low.
- If development needed to complete in specified time.
- RAD Model is suitable if the functionality have less dependencies on other functionality.

1.2.7 OBJECT ORIENTED MODEL

This data model is another method of representing real world objects. It considers each object in the world as objects and isolates it from each other. It groups its related functionalities together and allows inheriting its functionality to other related sub-groups.

Consider an Employee database to understand this model better. In this database , assume there are different types of employees – Engineer, Accountant, Manager, Clark. But all these employees belong to Person group. Person can have different attributes like name, address, age and phone. Two separate procedure sp_getAddress and sp_getPhone are used to get a person's address and phone number.

All the employees will have all the attributes what a person has. In addition, they have their EMPLOYEE_ID, EMPLOYEE_TYPE and DEPARTMENT_ID attributes to identify them in the organization and their department. The sp_getDeptDetails procedure is used to get a person's address and phone number. Currently, say we need to have only these attributes and functionality.

Since all employees inherit the attributes and functionalities of Person, those features can be re-used in Employee. Hence, group the features of person together into class. Hence a class has all the attributes and functionalities. For example, create a person class with name, address, age and phone as its attribute, and sp_getAddress and sp_getPhone as procedures in it. The values for these attributes at any instance of time are object. i.e. ; {Arul, Salem, 25, 2453545 : sp_getAddress (Arul), sp_getPhone (Arul} forms one person object. {Selvan, Chennai Town, 28, 5645677: sp_getAddress (Selvan), sp_getPhone (Selvan} forms another person object.

Now, create another class called Employee which will inherit all the functionalities of Person class. In addition it will have attributes EMPLOYEE_ID, EMPLOYEE_TYPE and DEPARTMENT_ID, and sp_getDeptDetails procedure. Different objects of Employee class are Engineer, Accountant, Manager and Clerk.

Observe that the features of Person are available only if other class is inherited from it. It would be a black box to any other classes. This feature of this model is called encapsulation. It binds the features in one class and hides it from other classes. It is only visible to its objects and any inherited classes.

Advantages

- It becomes more flexible in the case of any changes.
- Codes are re-used because of inheritance.
- It is more understandable.
- It reduces the overhead and maintenance costs.

Disadvantages

- It is not widely developed and complete to use it in the database systems. Hence it is not accepted by the users.
- It is an approach for solving the requirement. It is not a technology. Hence it fails to put it in the database management systems.

1.2.8 COMPARISON OF VARIOUS MODELS

MODEL	STRENGTHS	WEAKNESSES
Water fall Model	• Simple • Easy to execute • Intuitive and logical	• Disallows changes • Cycle time too long. • User feed back not allowed
Prototyping	• Reduce risk • Leads to a better System • Helps in requirements elicitation	• Higher cost • Disallows later changes
Iterative Enhancement	• Regular and fast deliveries • Reduces risk • Allows user feedback • Accommodates changes	• Each iteration can have planning overhead. • Cost may increase. • System Architecture and structure may suffer as frequent changes are made.
Spiral	• Controls project risks, • Very flexible. • Less documentation.	• No strict standards for software development. • No particular beginning or end of the particular phase.
RAD	• Fast application development and delivery. • Less testing activity required.	• High skilled resources required. • Difficult to manage.
Object Oriented	• More flexible. • Codes are re-used	• it is not accepted by the users. • it fails to put it in the database management systems.

1.3 SOFTWARE REQUIREMENT ANALYSIS (SRS)

1.3.1 VALUE OF GOOD SRS

The origin of most software systems is in the needs of some clients. The software system itself is created by some developers. Finally, the completed system will be used by the end users. Thus, there are three major parties: the client, the developer, and the users. The requirements for the system that will satisfy

the needs of the clients and the concerns of the users have to be communicated to the developer. The problem is that the client usually does not understand software or the software development process, and the developer often does not understand the client's problem and application area. This causes a communication gap between the parties involved in the development project. A basic purpose of the SRS is to bridge this communication gap so they have a shared vision of the software being built. Hence, one of the main advantages of a good SRS is: –

- **An SRS establishes the basis for agreement between the client and the supplier on what the software product will do.**

This basis for agreement is frequently formalized into a legal contract between the client (or the customer) and the developer (the supplier). So, through SRS, the client clearly describes what it expects from the supplier, and the developer clearly understands what capabilities to build in the software.

An SRS provides a reference for validation of the final product.

The SRS helps the client to determine if the software meets the requirements. Without a proper SRS, there is no way a client can determine if the software being delivered is what was ordered, and there is no way the developer can convince the client that all the requirements have been fulfilled.

- **A high-quality SRS is a prerequisite to high-quality software.**

Many errors are made during the requirements phase. And an error in the SRS will create an error in the final system implementing the SRS. Clearly, to create a high-quality end product with few errors, begin with a high-quality SRS.

- **A high-quality SRS reduces the development cost.**

The quality of SRS has an impact on cost of the project. Errors can exist in the SRS. The cost of fixing an error increases almost exponentially as time progresses Hence, by improving the quality of requirements, huge cost is saved savings in the future by having fewer expensive defect removals.

1.3.2 REQUIREMENT PROCESS

The requirement process typically consists of three basic tasks: problem or requirement analysis, requirements specification, and requirements validation.

Problem analysis often starts with a high-level "problem statement." During analysis, the problem domain and the environment are modeled in an effort to understand the system behavior, constraints on the system, its inputs and outputs, etc. The basic purpose of this activity is to obtain a thorough understanding of what the software needs to provide. Frequently, during analysis, the analyst will have a series of meetings with the clients and end users. In the early meetings, the analyst is basically the listener, absorbing the information provided by the end users. Once the analyst understands the system to some extent, the analyst uses the next few meetings to seek clarifications of the parts he does not understand. The analyst may document the information or build some models, and the analyst may do some

brainstorming or thinking about what the system should do. In the final few meetings, the analyst essentially explains to the client what he understands the system should do and uses the meetings as a means of verifying if what he proposes the system should do .

The understanding obtained by problem analysis forms the basis of requirements specification, in which the focus is on clearly specifying the requirements in a document. Issues such as representation, specification languages, and tools are addressed during this activity. As analysis produces large amounts of information and knowledge with possible redundancies, properly organizing and describing the requirements is an important goal of this activity.

Requirements validation focuses on ensuring that what have been specified in the SRS are indeed all the requirements of the software and making sure that the SRS is of good quality. The requirements process terminates with the production of the validated SRS. We will discuss this more later in the chapter.

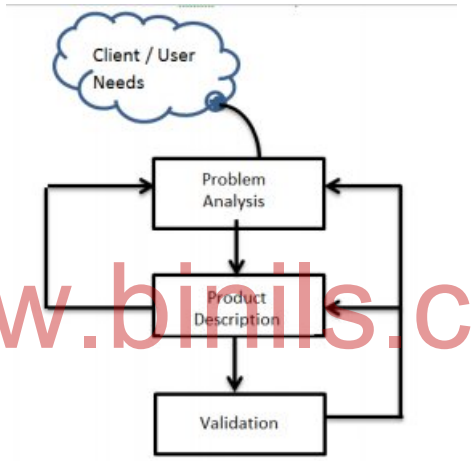


Figure 1.12: The requirement process.

The requirements process is not a linear sequence of these three activities and there is considerable overlap and feedback between these activities. The overall requirement process is shown in Figure 1.12. As shown in the figure, from the specification activity we may go back to the analysis activity. This happens as frequently some parts of the problem are analyzed and then specified before other parts are analyzed and specified. Furthermore, the process of specification frequently shows shortcomings in the knowledge of the problem, thereby necessitating further analysis. Once the specification is done, it goes through the validation activity. This activity may reveal problems in the specification itself, which requires going back to the specification step, or may reveal shortcomings in the understanding of the problem, which requires going back to the analysis activity.

1.3.2 REQUIREMENT SPECIFICATION

The final output is the SRS document. Modeling generally focuses on the problem structure, not its external behavior. Consequently, things like user interfaces are rarely modeled, whereas they frequently form a major component of the SRS. Similarly, for ease of modeling, frequently “minor issues” like

erroneous situations (e.g., error in output) are rarely modeled properly, whereas in an SRS, behavior under such situations also has to be specified. Similarly, performance constraints, design constraints, standards compliance, recovery, etc., are not included in the model, but must be specified clearly in the SRS because the designer must know about these to properly design the system. It should therefore be clear that the outputs of a model cannot form a desirable SRS.

The transition from analysis to specification should also not be expected to be straightforward, even if some formal modeling is used during analysis. A good SRS needs to specify many things, some of which are not satisfactorily handled during analysis. Essentially, what passes from requirements analysis activity to the specification activity is the knowledge acquired about the system. The modeling is essentially a tool to help obtain a thorough and complete knowledge about the proposed system. The SRS is written based on the knowledge acquired during analysis. As converting knowledge into a structured document is not straightforward, specification itself is a major task, which is relatively independent.

1.3.4 DESIARABLE CHARACTERISTICS OF SRS

CORRECTNESS: An SRS is correct if every requirement included in the SRS represents something required in the final system.

COMPLETENESS: An SRS is complete when it is documented after:

- The involvement of all types of concerned personnel.
- Focusing on all problems, goals, and objectives, and not only on functions and features.
- Correct definition of scope and boundaries of the software and system.3.

UNAMIGUOUS: An SRS is unambiguous if and only if every requirement stated has one and only one interpretation. Requirements are often written in a natural language. The SRS writer has to be especially careful to ensure that there are no ambiguities. One way to avoid ambiguities is to use some formal requirements specification language. The major disadvantage of using formal languages is the large effort required to write an SRS.

VERIFIABLE: An SRS is verifiable if and only if there exists some cost-effective process that can check whether the final product meets the requirements.

MODIFIABLE: An SRS is modifiable if its structure and style are such that any necessary change can be made easily while preserving completeness and consistency. The presence of redundancy is a major hindrance to modifiability, as it can easily lead to errors. For example, assume that a requirement is stated in two places and that the requirement later needs to be changed. If only one occurrence of the requirement is modified, the resulting SRS will be inconsistent

TRACEABLE: The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended:

- Backward traceability: This depends upon each requirement explicitly referencing its source in earlier documents.

- Forward traceability: This depends upon each requirement in the SRS having a unique name or reference number.

CONSISTENCY: Consistency in the SRS is essential to achieve correct results across the system. This is achieved by:

(i) The use of standard terms and definitions.
(ii) The consistent application of business rules in all functionality.
(iii) The use of a data dictionary

1.3.5 COMPONENETS OF SRS

Completeness of specifications is difficult to achieve and even more difficult to verify. The basic issues an SRS must address are:

- Functionality
- Performance
- Design constraints imposed on an implementation
- External interfaces

1. Functional Requirements

Functional requirements specify what output should be produced from the given inputs. So they basically describe the connectivity between the input and output of the system. For each functional requirement:

1. A detailed description of all the data inputs and their sources, the units of measure, and the range of valid inputs be specified:
2. All the operations to be performed on the input data and the output should be specified, and
3. Care must be taken not to specify any algorithms that are not parts of the system but that may be needed to implement the system.
4. It must clearly state what the system should do if system behaves abnormally when any invalid input is given or due to some error during computation. Specifically, it should specify the behaviour of the system for invalid inputs and invalid outputs.

2. Performance Requirements (Speed Requirements)

This part of an SRS specifies the performance constraints on the software system. All the requirements related to the performance characteristics of the system must be clearly specified. Performance requirements are typically expressed as processed transactions per second or response time from the system for a user event or screen refresh time or a combination of these.

3. Design Constraints

The client environment may restrict the designer to include some design constraints that must be followed. The various design constraints are standard compliance, resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

Standard Compliance: It specifies the requirements for the standard the system must follow. The standards may include the report format and according procedures.

Hardware Limitations: The software needs some existing or predetermined hardware to operate, thus imposing restrictions on the design. Hardware limitations can include the types of machines to be used, operating system availability ,memory space etc.

Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex and expensive, so they should be minimized.

Security: Currently security requirements have become essential and major for all types of systems. Security requirements place restrictions on the use of certain commands control access to database, provide different kinds of access, requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system.

External Interface Requirements

For each external interface requirements:

1. All the possible interactions of the software with people, hardware and other software should be clearly specified.
2. The characteristics of each user interface of the software product should be specified and
3. The SRS should specify the logical characteristics of each interface between the software product and the hardware components for hardware interfacing.

1.3.6 STRUCTURE OF A SRS

Requirements have to be specified using some specification language. Though formal notations exist for specifying specific properties of the system, natural languages are now most often used for specifying requirements. When formal languages are employed, they are often used to specify properties or for specific parts of the system, as part of the overall SRS.

All the requirements for a system, stated using a formal notation or natural language, have to be included in a document that is clear and concise. For this, it is necessary to properly organize the requirements document.

The IEEE standards recognize the fact that different projects may require their requirements to be organized differently, that is, there is no one method that is suitable for all projects. It provides different ways of structuring the SRS. The first two sections of the SRS are the same in all of them. The general structure of an SRS is given below:

1. **Introduction**
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definition, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
   - 1.6
2. **Overall Description**
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and dependencies
3. **Specific Requirements**
   - 3.1 External Interfaces
   - 3.2 Functional requirements
   - 3.3 Performance requirements
   - 3.4 Logical Database requirements
   - 3.5 Design Constraints
   - 3.6 Software system attributes
   - 3.7 Organizing the specific requirements
   - 3.8 Additional Comments
4. **Supporting Information**
   - 4.1 Table of contents and index
   - 4.2 Appendixes

1.3.7 PROBLEMS IN SRS

- **Requirements are difficult to uncover**: In recent trends in engineering, the processes are automated and it is practically impossible to understand the complete set of requirements during the commencement of the project itself.
- **Requirements change:** Requirements are continuously generated. Defining the complete set of requirements in the starting is difficult. When the system is put under run, the new requirements are obtained and need to be added to the system. But, the project schedules are seldom adjusted to reflect these modifications. Otherwise, the development of software will never commence.
- **Over-reliance on CASE tools**: CASE tools are good helping agents, over reliance on these Requirements Engineering Tools may create false requirements. Thus, the requirements corresponding to real system should be understood and only a realistic dependence on tools should be made.

- **Tight project schedule**: The software projects are generally given tight project schedules. Pressure is created from customer side to hurriedly complete the project. This normally cuts down the time of requirements analysis phase, which frequently lead to.
- **Communication barriers**: Requirements engineering is communication intensive. Users and developers have different vocabularies, professional backgrounds and psychology. User writes specification in natural language and developer usually demands precise and well-specified requirement.
- **Market – driven software development**: In present time, the software development is market driven having high commercial aspect. The software developed should be a general purpose one to satisfy anonymous customer, and then, it is customized to suit a particular application.
- **Lack of resources**: The resources may not be enough to build software that fulfils all the customer's requirements. It is left to the customer to prioritize the requirements and develop software fulfilling important requirements.

1.3.8 REQUIREMENTS GATHERING TOOLS

The following four tools are primarily used for information gathering:

1. **Record review:** A review of recorded documents of the organization is performed. Procedures, manuals, forms and books are reviewed to see format and functions of present system. The search time in this technique is more.
2. **On site observation:** In case of real life systems, the actual site visit is performed to get a close look of system. It helps the analyst to detect the problems of existing system.
3. **Interview:** A personal interaction with staff is performed to identify their requirements. It requires experience of arranging the review, setting the stage, avoiding arguments and evaluating the outcome.
4. **Questionnaire:** It is an effective tool which requires less effort and produces a written document about requirements. It examines a large number of respondents simultaneously and gets customized answers. It gives person sufficient time to answer the queries and give correct answer.

1.3.9 ANALYSIS TOOLS

Software analysis and design includes all activities, which help the transformation of requirement specification into implementation. Requirement specifications specify all functional and non-functional expectations from the software. These requirement specifications come in the shape of human readable and understandable documents, to which a computer has nothing to do.

Software analysis and design is the intermediate stage, which helps human-readable requirements to be transformed into actual code. Analysis tools are used in the system analysis and Design phase.

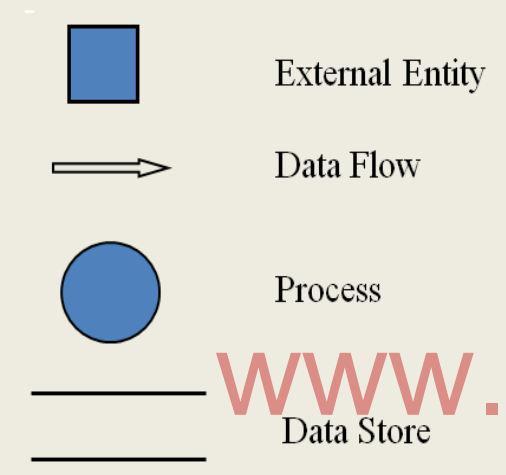
1.3.10 DATAFLOW DIAGRAM

Data Flow Diagram (DFD) is a graphical representation of flow of data in an information system. It is capable of showing incoming data flow, outgoing data flow, and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart shows flow of control in program modules. DFDs show flow of data in the system at various levels. It does not contain any control or branch elements.

DFD can easily illustrate relationships among data flows, external entities and data stores.

SYMBOLS USED IN DFD



External entities : External entities, from which the data flows and where results terminate. An External Entity is a provider (source) or receiver (sink) of data and information of the system- Data must always originate somewhere and must always be sent to something e.g., a person, a device, a sensor, computer-based.

They do not process data. They either supply data or receive data . A Rectangle represents an external entity.

Process: Process is a data transformer (changes input to output)

- A Process is a work or action performed on input data flow to produce an output data flow.
- Use a verb/verb phrase to label the action performed by the process
- A Process must have at least one input data flow and at least one output data flow.
- Circle represents a process
- Straight lines with incoming arrows are input data flows
- Straight lines with outgoing arrows are output data flows
- Processes are given serial numbers for easy reference

Data store: Data stores from which the data are read or into which data are written by the processes. A Data store is an inventory of data.

- A data store means “data at rest.”
- A data flow means “data in motion”
- A Data Store is a repository of data
- Data can be written into the data store. This is shown by n outgoing arrow

- Data can be read from a data store .This is shown by an incoming arrow
- External entity cannot read or write to the data store
- Two data stores cannot be connected by a data flow

Difference between flowcharts & DFD

The program flowchart describes boxes that describe computations, decisions, interactions and loops. Data flow diagrams are not program flowcharts and should not include control elements. A good DFD should

- have no data flows that split up into a number of other data flows
- have no crossing lines
- not include flowchart loops of control elements

RULES OF DATA FLOW:

Data can flow from

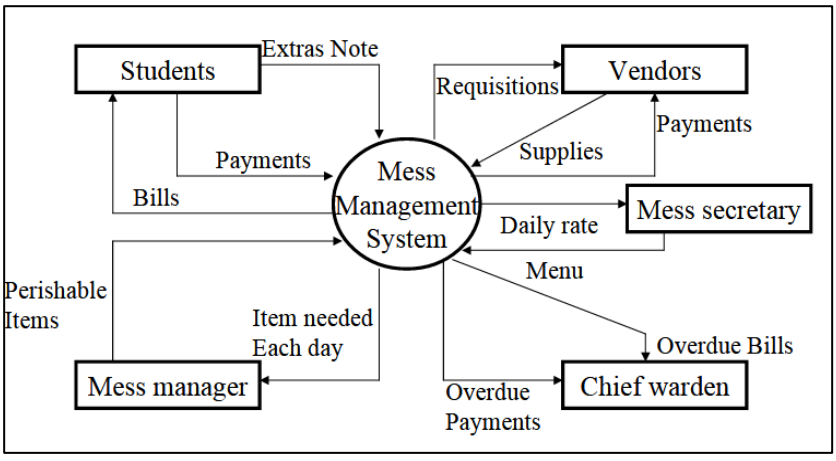
- External entity to Process
- Process to External Entity
- Process to store and back
- Process to Process

Data cannot flow from

- External entity to external entity
- External entity to store
- Store to external entity
- Store to store

Describing a system with a DFD:

Level 0 - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs. Context diagram of mess management system is shown below:



CONTEXT DIAGRAM OF MESS MANAGEMENT SYSTEM

DFD can also be drawn in increasing levels of detail, starting with a summary high level view and proceeding to more detailed lower level views.

The main merit of DFD: The main merit of DFD is that it provides an overview of what data flows in a system, what transformations are done on the data, what files are used and where results flow.

Role of DFD as a documentation aid: It is a good documentation aid which is understood by both programmers and non-programmers (i.e., laypersons). As DFD specifies only what processes are performed and not how they are performed it is easily understood by a non programming user.

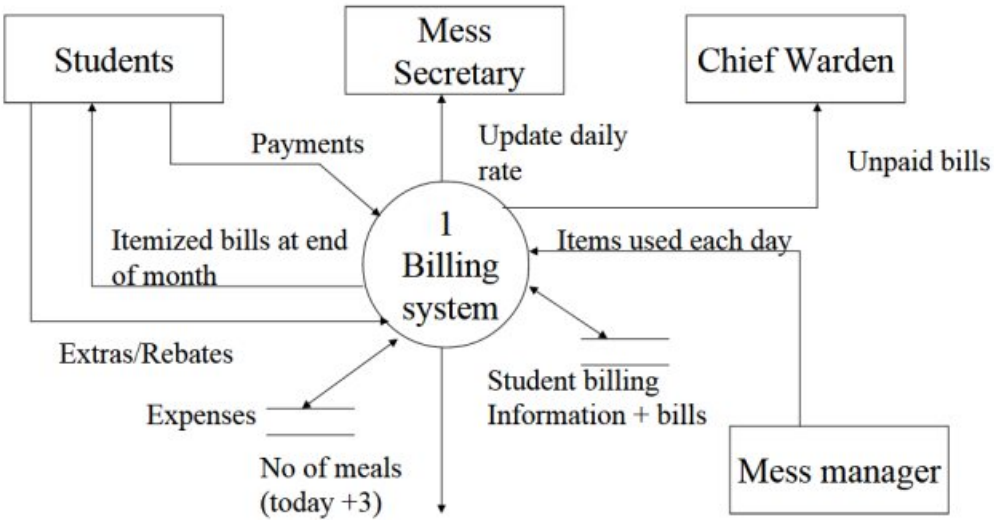
Context diagram: A diagram giving an entire system's data flows and processing with a single Process (circle) is called a context diagram.

Levelling of DFD : A context diagram is expanded into a number of inter-related processes. Each process may be further expanded into a set of inter-connected sub processes. This procedure of expanding a DFD is known as levelling.

Levelling DFD

Level 1 - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.

If a DFD is too detailed it will have too many data flows and will be large and difficult to understand



EXPANDED DFD FOR HOSTEL MESS MANAGEMENT

Level 2 - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

LEVELLING RULES

- If process *p* is expanded, the process at the next level are labeled as *p.1, p.2* etc.
- All data flow entering or leaving *p* must also enter or leave its expanded version.
- Expanded DFD may have data stores
- No external entity can appear in expanded DFD
- Keep the number of processes at each level less than.

DATA DICTIONARY

A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data **grossPay** consists of the components regularPay and overtimePay.

grossPay = regularPay + overtimePay

For the smallest units of data items, the data dictionary lists their name and their type. Composite data items can be defined in terms of primitive data items using the following data definition operators:

- **+**: denotes composition of two data items, e.g. **a+b** represents data a and **b**.
- **[,,]**: represents selection, i.e. any one of the data items listed in the brackets can occur. For example, **[a,b]** represents either **a** occurs or **b** occurs.
- **()**: the contents inside the bracket represent optional data which may or may not appear. e.g. **a+(b)** represents either **a** occurs or **a+b** occurs.
- **{}**: represents iterative data definition, e.g. **{name}5** represents five **name** data. **{name}*** represents zero or more instances of **name** data.
- **=**: represents equivalence, e.g. **a=b+c** means that **a** represents **b** and **c**.
- **/* */**: Anything appearing within **/*** and ***/** is considered as a comment.

DATA DICTIONARY

Provides definitions for all elements in the system which include:

- Meaning of data flows and stores in DFDs
- Composition of the data flows e.g. customer address breaks down to street number, street name, city and postcode
- Composition of the data in stores e.g. in Customer store include name, date of birth, address, credit rating etc.
- Details of the relationships between entities

EXAMPLES:

- name = courtesy-title + first-name + (middle-name) + last-name
- courtesy-title = [Mr. | Miss | Mrs. | Ms. | Dr. | Professor]
- first-name = {legal-character}
- middle-name = {legal-character}
- last-name = {legal-character}
- legal-character = [A-Z|a-z|0-9|'|-| |]
- order = customer-name + shipping-address + 1{item}10
  means that an order always has a customer name and a shipping address and has between 1 and 10 items

The Data Dictionary can be used to:

- Create and ordered listings of all data items
- Create an ordered listing of a subset of data items.
- Find a data item name from a description.
- Design the software and test cases.

1.3.11 ER DIAGRAM

The Entity-relationship (E-R) diagram is detailed logical representation of data for an organization. It is data oriented model of a system whereas DFD is a process oriented model. The ER diagram represents data at rest while DFD tracks the motion of data. ERD does not provide any information regarding functionality of data

Entity relationship diagrams have three different components: ENTITIES, RELATIONSHIPS and ATTRIBUTES

ENTITY

An entity can be a person, place, event, or object that is relevant to a given system. For example, a school system may include students, teachers, major courses, subjects, fees, and other items. Entities are represented in ER diagrams by a rectangle and named using singular nouns.

An **entity set** is a set of entities of the same type that share the same properties.

Example: set of all persons, companies, trees, holidays

RELATIONSHIP

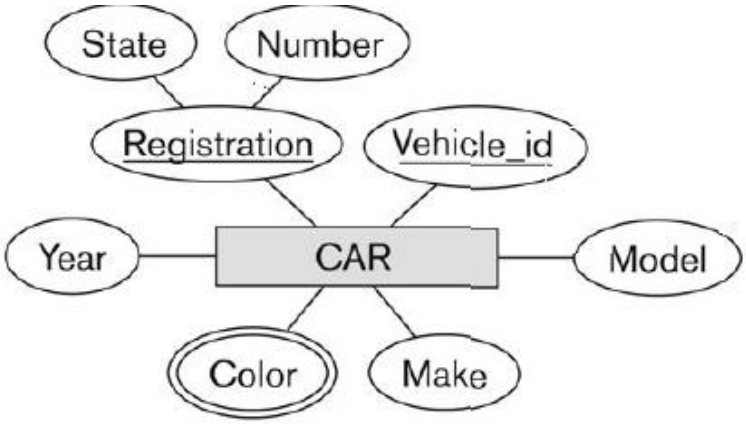
Entities are connected to each other by relationships. It indicates how two entities are associated. A diamond notation with name of relationship represents as written inside.

ATTRIBUTES

Attributes are specific properties of entities. For example, the attributes of the entity car is (Registration no., vehicle-id, year of manufacturing, model, color, make).

- Attributes are displayed in ovals
- Each attribute is connected to its entity type
- Components of a composite attribute are connected to the oval representing the composite attribute
- Each key attribute is underlined
- Multivalued attributes displayed in double ovals

Example:

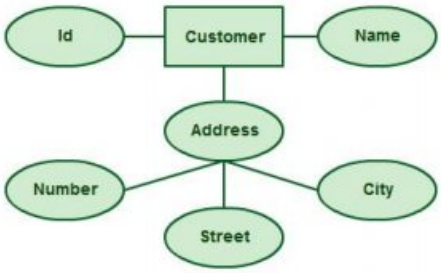


TYPES OF ATTRIBUTES

Simple Attributes: having atomic or indivisible values. Example: *Dept*–a string, *PhoneNumber*–an eight digit number

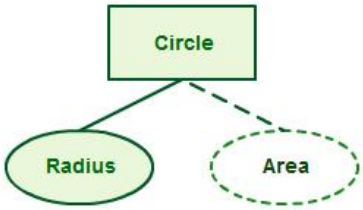
Composite Attributes:

having several components in the value. Example: *Address* with components (Number, street , city)



Derived Attributes:

An attribute based on another attribute. This is found rarely in ER diagrams. For example for a circle the area can be derived from the radius.

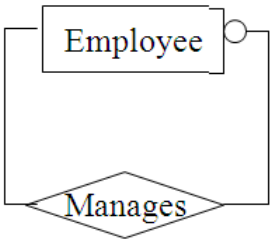


Single-valued: having only one value rather than a set of values. Example: *Place Of Birth* – single string value.

Multi-valued: having a set of values rather than a single value. Example : *Courses-Enrolled* attribute for student , *EmailAddress* attribute for student, *PreviousDegree* attribute for student.

DEGREE OF A RELATIONSHIP

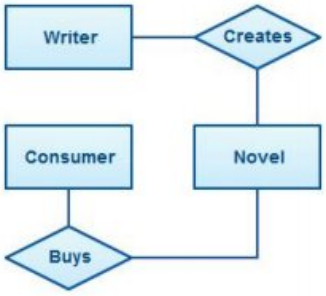
A relationship's degree indicates the number of entities that participate in the relationship.



A **unary relationship** exists when an association is maintained within a single entity. (Recursive relationship)

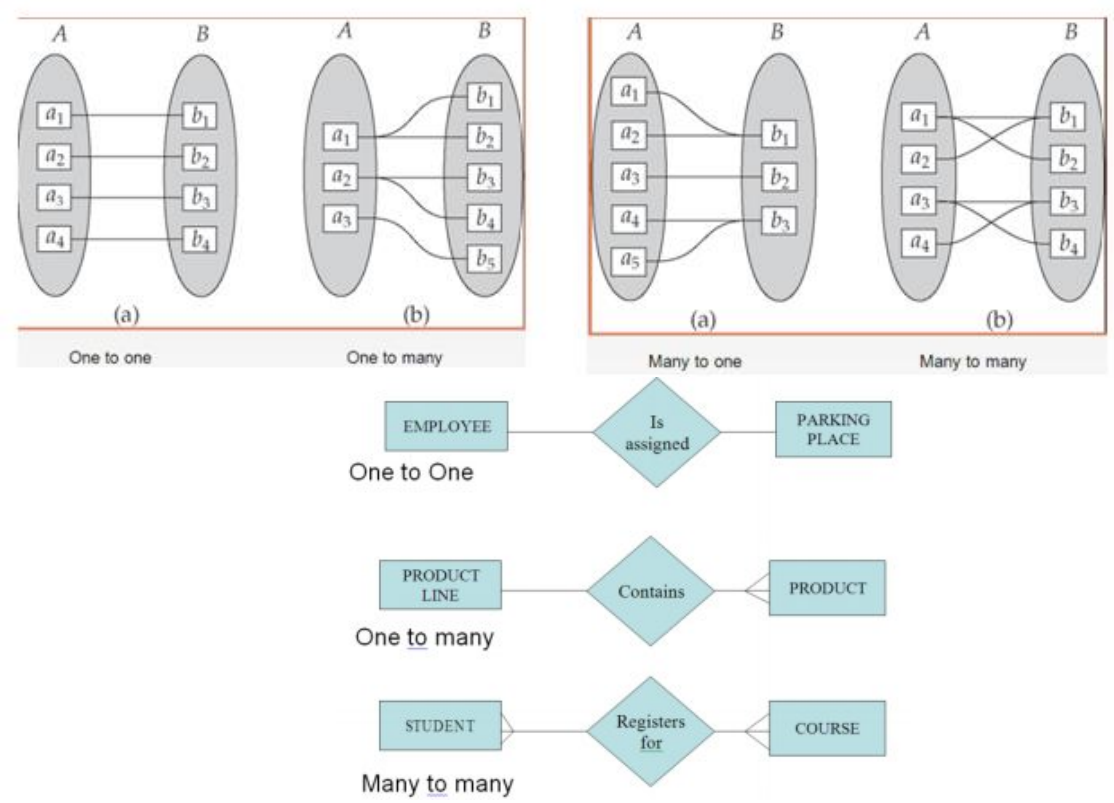
A **binary relationship** exists when two entities are associated. (Most common).

Three entities participate in a **ternary relationship**.



Connectivity

The term connectivity is used to describe the relationship classification (e.g., one-to-one, one-to-many, and many-to-many).



Cardinality

Cardinality expresses the specific number of entity occurrences associated with one occurrence of the related entity.

When the minimum number is zero, the relationship is usually called optional and when the minimum number is one or more, the relationship is usually called mandatory.

- Normally, the minimum cardinality will be 0 or 1, and the maximum cardinality will be 1 or *. Thus, only the (0, 1), (1, 1), (0,*), (1,*) cardinalities are common in practice.
- To understand a relationship, one must know the cardinality specifications on both sides.
- The maximum cardinalities on each side are used to distinguish between many-to-many, one-to-many / many-to-one, and one-to-one relationships.

BENEFITS OF ER DIAGRAM

ER diagrams constitute a very useful framework for creating and manipulating databases. First, ER diagrams are easy to understand and do not require a person to undergo extensive training to be able to work with it efficiently and accurately. Designers can use ER diagrams to easily communicate with developers, customers, and end users, regardless of their IT proficiency. Second, ER diagrams are readily translatable into relational tables which can be used to quickly build databases. In addition, ER diagrams can directly be used by database developers as the blueprint for implementing data in specific software applications. Lastly, ER diagrams may be applied in other contexts such as describing the different relationships and operations within an organization.

SUMMARY

- Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures
- Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, i.e., the application of engineering to software.
- Software Engineering deals with cost effective solutions to practical problems by applying scientific knowledge in building software artifacts in the service of mankind.
- The software products are divided in two categories: (i) Generic products (ii) Customized products.
- The prototyping model is applied when detailed information related to input and output requirements of the system is not available
- The spiral model starts with an initial pass through a standard waterfall lifecycle, using a subset of the total requirements to develop a robust prototype.
- The spiral model is similar to the waterfall model as software requirements are understood at the early stages in both the models.
- RAD Model or Rapid Application Development Model is similar to incremental model and waterfall model.

- This data model is another method of representing real world objects.
- The requirement process typically consists of three basic tasks: problem or requirement analysis, requirements specification, and requirements validation.
- An SRS establishes the basis for agreement between the client and the supplier on what the software product will do.
- **CORRECTNESS**: An SRS is correct if every requirement included in the SRS represents something required in the final system.
- An SRS is unambiguous if and only if every requirement stated has one and only one interpretation.
- Consistency in the SRS is essential to achieve correct results across the system.
- A data dictionary lists all data items appearing in the DFD model of a system
- Entities are connected to each other by relationships. It indicates how two entities are associated. A diamond notation with name of relationship represents as written inside.
- Attributes are specific properties of entities.
- A relationship's degree indicates the number of entities that participate in the relationship.
- Cardinality expresses the specific number of entity occurrences associated with one occurrence of the related entity.

REVIEW QUESTIONS

PART – A (2 Marks)

1. Define software engineering.
2. Comment on the statement “software does not wear out”
3. List out any two Software Myths?
4. Write down the categories of software products.
5. What are the different phases of SDLC?
6. Why waterfall model gets failures?
7. State any two advantages of waterfall model.
8. Write down any four SDLC methods.
9. What are the end products of each step of SDLC?
10. List down the task regions in the spiral model
11. What is the most important feature of spiral model?
12. List out the phases of RAD model.
13. Define : Correctness and Completeness
14. What are the two types of requirements?
15. What is SRS document?
16. What are the different symbols used in DFD?
17. What is context diagram?
18. State any two uses of SRS document.
19. What are the different components of SRS?
20. State the component of E-R diagram.
21. Define the term cardinality.
22. Define the term optimality.
23. List down the different types of attributes.
24. What is an attribute?
25. Give an example of composite attribute.
26. Give an example of multivalued attribute.
27. Differentiate Flowchart and DFD?
28. What is data dictionary?
29. Name some of the notations used in data dictionary.
30. What is E-R diagram?
31. What are the similarities between data flow diagram and E-R diagram?
32. What are functional requirements?

33. "Carpenter makes chair" draw ER diagram.
34. What is unary degree relationship?
35. Draw an ER diagram to illustrate one to one and one to many relationship
36. What is the input/output for requirement engineering?
37. List down any two requirement elicitation techniques.
38. How the operating procedures are classified?
39. Why documents and documentation have are very important?
40. State any four rules for flow of data in DFD.
41. A software project is considered as very simple and the customer is in the position of giving all the requirements at the initial stage, Which process model would you like to prefer for developing the project.

PART – B (3 Marks)

1. Write down any three of software characteristics
2. List out the operating procedures.
3. Define prototyping model. Write down the steps carried out for this model.
4. Write down any three reasons for using prototyping model.
5. Mention any three values of Good SRS
6. Define traceability. Mention its types.
7. Discuss the problems in SRS (Any three)
8. What are the tools available for Requirement gathering.
9. List out the three types of attributes?
10. Explain about degree of relationship.
11. State any three advantages of Data Dictionary.
12. Draw an E-R diagram having unary degree relationship.

PART – C (5 Marks/ 10 Marks)

1. Write down the need for software engineering.
2. Distinguish between a program Vs software product.
3. Draw the schematic diagram of the waterfall model and also discuss each phase in brief.
4. What are the major advantages of constructing a working prototype before developing actual product?
5. Explain iterative enhancement model with the help of suitable example.
6. What are the major phases in the spiral model of software development? Explain.
7. Explain RAD model in detail.

8. Write short notes on comparisons between various software development Life cycle models.
9. Describe the various steps of requirements engineering. List out the requirements elicitation techniques. Which one is most popular and why?
10. Draw the ER diagram for a hotel reception disk management
11. State some advantages and disadvantages of data flow diagram
12. Consider a student admission of XYZ university, which is to be automated. For this syntax, make DFD of 2 and 3 levels. Also draw an ER diagram.
13. Explain software myths.
14. What are the advantages and disadvantages of (i) Waterfall model (ii) Iterative enhancement model.
15. What are the symbols used in a date dictionary? Explain the use of each symbol.
16. Explain prototype model with their advantages and disadvantages.
17. Write down the major characteristic of software. Illustrate with a diagram that the software does not wear out.
18. List down the important issues which a SRS document must address.
19. What are the different types of attributes used in ER diagram? Explain them by giving the example.
20. Give the format of SRS document.

UNIT – II	

SOFTWARE DESIGN AND PLANNING

OBJECTIVES

At the end of the unit, the students will be able to

- Define software design
- Understand the objectives of software design
- Understand architectural and modular design.
- Know the various programming style.
- Understand the different types of project metrics.
- Understand different software estimation techniques.
- Know the reasons for poor and inaccurate estimation.
- Describe CASE.
- Understand the architecture of CASE environment.
- Know about the CASE tools.

2.1 SOFTWARE DESIGN

2.1.1 DEFINITION

Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language.

Design activities can be broadly classified into two important parts: (i) Preliminary (or high-level) design and (ii) Detailed design.

2.1.2 OBJECTIVES OF SOFTWARE DESIGN

Correctness: A good design should correctly implement all the functionalities identified in the SRS document.

Verifiability: Design should be correct and it should be verified for correctness. Verifiability is concerned with how easily the correctness of the design can be checked. Various verification techniques should be easily applied to design.

Completeness: Completeness requires that all the different components of the design should be verified i.e., all the relevant data structure modules, external interfaces and module interconnections are specified.

Traceability: Traceability is an important property that can get design verification. It requires that the entire design element must be traceable to the requirements.

Efficiency: Efficiency of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system consumes less processor time and memory.

Simplicity: Simplicity is the most important quality criteria for software systems. Maintenance of software system is usually quite expensive. The design of the system is one of the most important factors affecting the maintainability of the system.

2.1.3 PROCESS OF SOFTWARE DESIGN

Software design is the process of applying various software engineering techniques to develop models to define a software system. It provides sufficient details for actual realization of the software. The objective of software design is to translate user requirements into an implemental program. Figure 2.1 shows the process of software design.

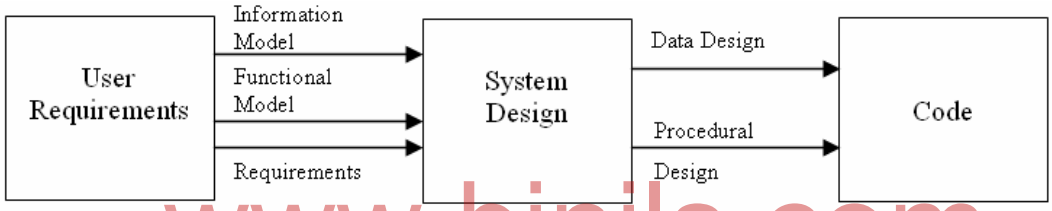


Fig No 2.1 Process of Software Design

During the process of software design, **Information model** is translated to data design. **Functional model and behavioral model** are translated to architectural design, which defines major component of the software.

Software design forms the foundation for implementation of a software system and helps in the maintenance of software in future too.

High level design gives a holistic view of the software to be built, whereas l**ow level** refinements of the design are very closely related to the final source code. **A good design** can make the work of programmer easy and hardly allow the programmer to forget the required details. Sufficient time should be devoted to design process to ensure good quality software.

The following are some of the fundamentals of design:

- The design should follow a hierarchical organization.
- Design should be modular which are logically partitioned into modules which are relatively independent and perform independent task.
- Designs leading to interface that facilitate interaction with external environment.
- Stepwise refinement to more detailed design which provides necessary details for the developer of code.
- Modularity is encouraged to facilitate parallel development, but at the same time, too many modules lead to the increase of effort involved in integrating the modules.

Data design is the first and the foremost activity of system design. Data describes a real-world information resource that is important for the application. Data describes various entities like customer, people, asset, student records etc.,

Identifying data in system design is an iterative process. At the highest level, data is defined in a very vague manner. A high-level design describes how the application handles these information resources. When expanding the application to the business needs or business processes, we tend to focus more on the details.

Data Design
Architectural design
Modular design
Human Computer Interface design

Fig No 2.2. Technical aspects of Design

The primary objective of data design is to select logical representation of data items identified in requirement analysis phase. Fig 2.2. Shows the technical aspects of design. The description for each item of data typically includes the following:

- Name of the data item
- General description of the data item
- Characteristics of data item
- Ownership of the data item
- Logical events, processes, and relationships.

DATA STRUCTURE

A data structure defines a logical relationship between individual elements of a group of data. The structure of information affects the design of procedures/algorithms. Proper selection of data structure is very important in data designing. Simple data structure like a scalar item forms the building block of more sophisticated and complex data structures.

A scalar item is the simplest form of data. For example, January (Name of the Month), where as the collection of months in a year form a data structure called a vector item.

2.1.4 ARCHITECTURAL DESIGN

The objective of architectural design is to develop a model of software architecture which gives overall organization of program module in the software product. Software architecture encompasses two aspects of structure of the data and hierarchical structure of the software components. The following figure shows how a single problem can be translated to a collection of solution domains

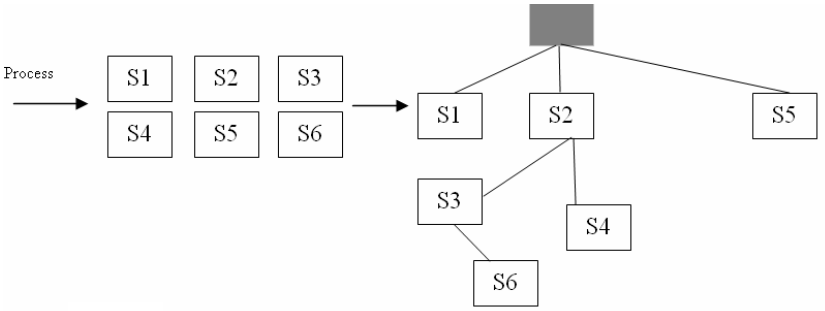


Fig No 2.3 Problem, Solutions and Architecture

Architectural design defines organization of program components. It does not provide the details of each components and its implementation. The following figure 2.4. shows the architecture of Examination Management System.

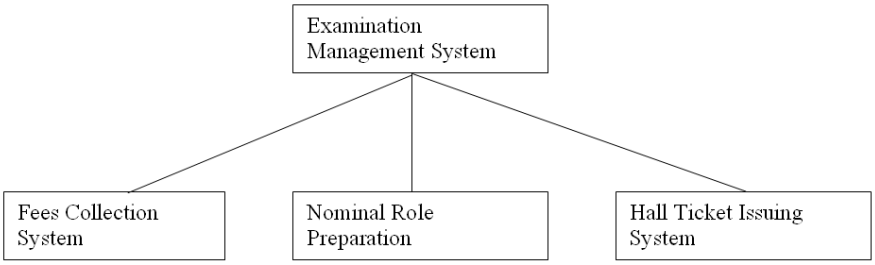


Fig No 2-4 Architecture of Examination Management System.

The architectural design is also used to control relationship between modules. One module may control another module or may be controlled by another module. These characteristics are defined by the fan-in and fan-out of a particular module. The organization of module can be represented through a tree like structure.

Consider the following architecture of a software system.

The number of level of component in the structure is called depth and the number of component across the horizontal section is called width. The number of components which controls a said component is called **fan-in** i.e., the number of incoming edges to a component. The number of components that are controlled by the module is called **fan-out** i.e., the number of outgoing edges.

Figure: Fan-in and Fan-out

S0 controls three components, hence the fan-out is 3. S2 is controlled by two components, namely, S1 and S5. Hence the fan-in is 2.

2.1.5 MODULAR DESIGN

Modular design facilitates future maintenance of software. Modular designs have become well accepted approach to build software product.

Software can be divided into relatively independent, named and addressable component called a module. Modularity is the only attribute of a software product that makes it manageable and maintainable. Dividing the software into modules helps software developer to work on independent modules that can be later integrated to build the final product. In a short, it encourages parallel development effort thus saving time and cost.

During designing of software, one must keep a balance between number of modules and integration cost. Although, more numbers of modules make the software product more manageable and maintainable. At the same time, where the number of modules increases, the cost of integrating the modules also increases.

Important properties of a modular system

- Each module is a well-defined subsystem useful to others.
- Each module has a well-defined single purpose.
- Modules can be separately compiled and stored in library.
- Modules can use other modules.
- Modules should be easier to use than build.
- Modules should have a simple interface.

Advantages of Modular Systems

- Modular systems are easier to understand and explain.
- Modular systems are easier to document.
- Programming individual modules is easier because the programmer can focus on just one small, simple problem rather than a large complex problem.
- Testing and debugging individual modules is easier .
- Well – Composed modules are more reusable because they are more likely to comprise part of a solution to many problems.
- A good module should be easy to extract from one program and insert into another.

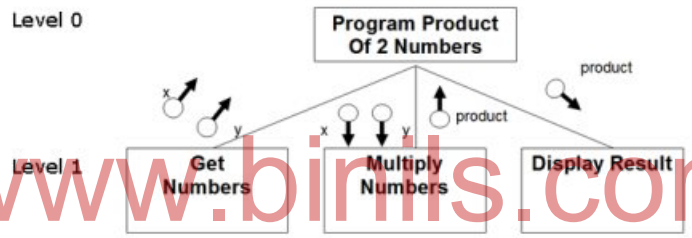
2.1.6 STRUCTURE CHART

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

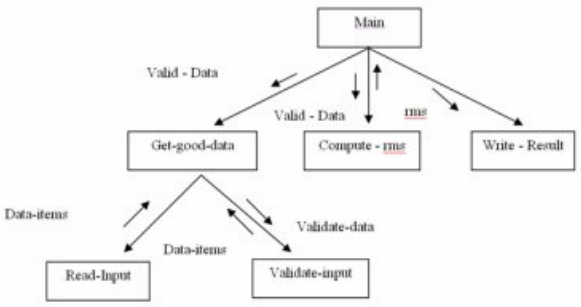
SYMBOLS USED IN A STRUCTURE CHART

- **Rectangular boxes:** Represents a module.
- **Module invocation arrows:** Control is passed from one module to another module in the direction of the connecting arrow.
- **Data flow arrows:** Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
- **Library modules:** Represented by a rectangle with double edges.
- **Selection:** Represented by a diamond symbol.
- **Repetition:** Represented by a loop around the control flow arrow.

Modules at the top level call the modules at the lower level. The connections between modules are represented by lines between the rectangular boxes. The components are generally read from top to bottom, left to right. Modules are numbered in hierarchical numbering scheme. In any structure chart there is one and only one module at the top called the root.



Example: A software system called average calculating software reads three integral numbers from the user in the range between 100 and 1000 and determines the average of the three input numbers and then displays it.



Structure Chart vs. Flow Chart

Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

• It is usually difficult to identify the different modules of the software from its flow chart representation.

• Data interchange among different modules is not represented in a flow chart.

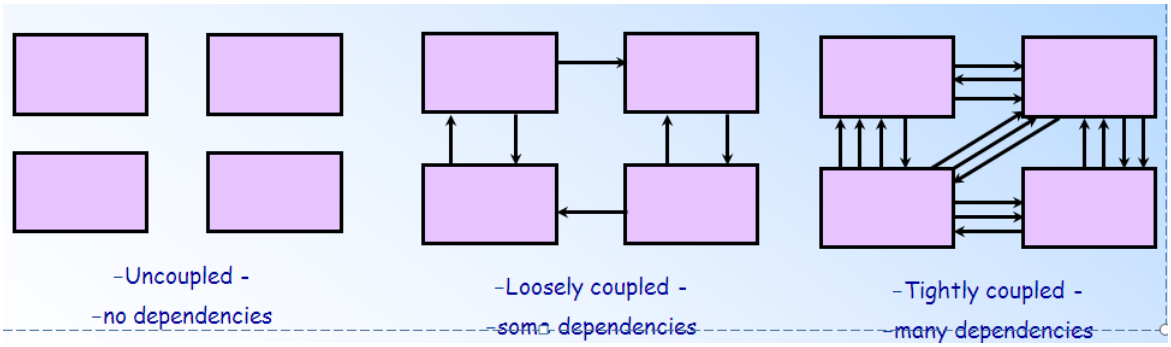
• Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

2.1.7 COUPLING AND COHESION

COUPLING

Coupling is the measure of the degree of interdependence between modules.

Type of coupling: Different types of coupling are content, common, external, control, stamp and data.



The strength of coupling from lowest coupling (best) to highest coupling (worst) is given below.

- Data coupling (Best)
- Stamp coupling
- Control coupling
- External coupling
- Common coupling
- Content coupling (Worst)

Data coupling: Only simple data is passed between modules 1 to 1 correspondence of items exists

Stamp coupling: Occurs when complete data structures are passed from one module to another.

Example: The print routine of the customer billing accepts a customer data structure as an argument, parses it, and prints the name, address, and billing information.

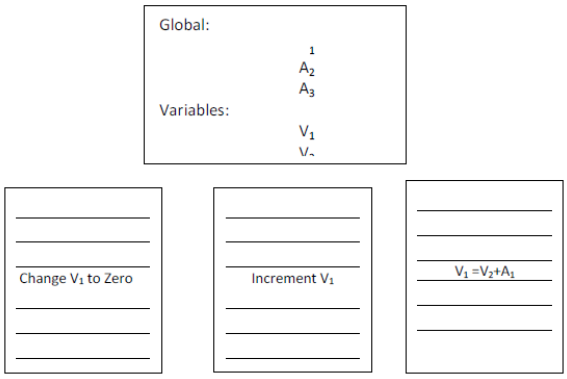
Control coupling: One module directs the execution of another module by passing the necessary control information accomplished by means of flags that set by one module and reacted upon by the dependent module

Example: Module p calls module q and q passes back flag that says it cannot complete the task, then q is passing data.

External coupling: Modules communicate through an external medium (such as file)

Common coupling:

- Two modules have shared data
- Occurs when a number of modules reference a global data area



Module X Module Y Module Z

Example: Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks.

Each source process writes directly to global data store. Each sink process reads directly from global data store.

Content coupling (should be avoided):

- One component references contents of another.
- Module directly affects the working of another module
- Occurs when a module changes another module's data or when control is passed from 1 module to the middle of another (as in a jump)

Example: Part of program handles lookup for customer. When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

What problems arise if two modules have high coupling?

***Ans**: A system with high coupling means there are strong interconnections between its modules. If two modules are involved in high coupling, it means their interdependence will be very high. Any changes applied to one module will affect the functionality of the other module. Greater the degree of change, greater will be its effect on the other. As the dependence is higher, such change will affect modules in a negative manner and in-turn, the maintainability of the project is reduced. This will further reduce the reusability factor of individual modules and hence lead to unsophisticated software. So, it is always desirable to have inter-connection & interdependence between modules.*

COHESION

Cohesion of a module represents how tightly the internal elements of the module are bound to one another.

Functional Cohesion: Every essential element to a single computation is contained in the component. Every element in the component is essential to the computation.

Example : i) calculate average ii) print result

Sequential Cohesion: The output of one component is the input to another. Occurs naturally in functional programming languages.

Example: Update record in data base and send it to the printer.

Communicational Cohesion: Definition: Module performs a series of actions related by a sequence of steps to be followed by the product and all actions are performed on the same data

Procedural cohesion:

Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution.

Temporal Cohesion: Performs a set of functions whose only relationship is that they have to be carried out at the same time

Example : (i) set of initialization operations (ii) Clear screen (iii) Open file (iv) Initialize total

Logical Cohesion: Perform a set of logically similar functions.

Example : function => output anything. Perform output operations: (i) Output text to screen (ii) Output line to printer (iii) Output record to file

Coincidental Cohesion:

- Grouped into modules in a haphazard way
- no relationship between component

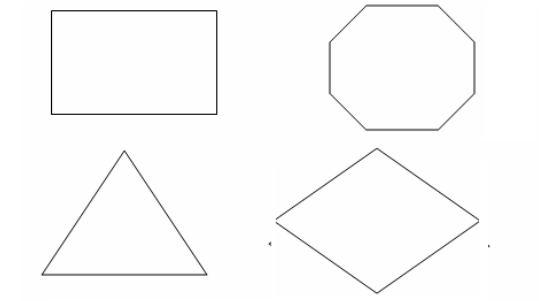


Fig No 2-7. High Cohesive and loosely coupled System (Desirable)

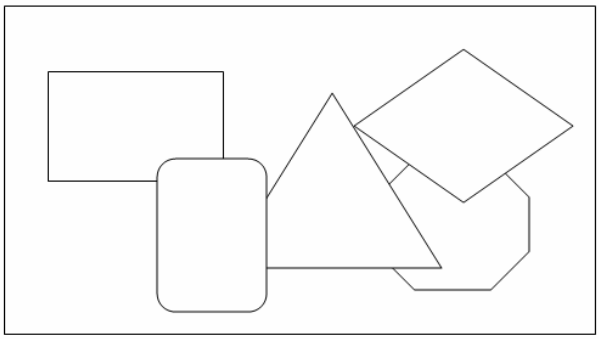


Fig No 2.8 Low Cohesive and Highly coupled System (Undesirable)

What problems are likely to arise if a module has low cohesion?

Cohesion measure the relative functional strength of a module. It represents the strength of bond between the internal elements of the modules. More tightly the elements are bound to each other, higher will be the cohesion of the module. In practice, designers should avoid a low level of cohesion when designing a module. Generally, low cohesion results in high coupling and, hence, changes in one module results in errors in other modules as well.

2.1.8 INTERFACE DESIGN

Interface design is one of the most important parts of software design. The user interaction with the system takes place through various interfaces provided by the software product.

The following are the advantages of a Graphical User Interface (GUI):

- Various information can be displayed and allow the user to switch to different task directly from the present screen.
- Reduces typing effort of the user.
- Provides key-board shortcut to perform frequently performed tasks.
- Simultaneous operations of various task without loosing the present context.
- Any interface design is targeted to users of different categories.
- Expert user with adequate knowledge of the system and application.
- Average user with reasonable knowledge.
- Novice user with little or no knowledge.

2.1.9 DESIGN OF HUMAN COMPUTER INTERFACE

Human Computer Interface (HCI) design is gaining significance as the use of computers is growing. A personal desktop PC has following interface for humans to interact.

A key board, A Computer Mouse , A Touch Screen (if equipped with) and a program on your windows machine that includes icons of various programs, disk drives, and folders.

Fig 2-9 The Process of Interface Design

The process of HCI design beings with a creation of a model of system function as perceived by the end user. The designer must consider the types of users ranging from novice user, knowledgeable but intermittent user and expert frequent user. Accommodating expectation of all types of user is important.

Each type of user expects the screen layout to accommodate their desires, novices needing extensive help where as expert user want to accomplish the task in quickest possible time. For example providing a command such as ^P(control P) to a specific report as well as a printer icon to do the same job for the novice user.

Rules for Human Computer Interface Design

Consistency: Interface is designed to ensure consistent sequences of actions for similar situations. Terminology should be used in prompts, menus, and help screens should be identical. Color scheme, layout and fonts should be consistently applied throughout the system.

Enable expert users to use shortcuts: Use of short cut increases the productivity. Shortcuts are used to increase the pace of interaction with use of, special keys and hidden commands.

Feedback. A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request. If required, the user should be periodically informed about the progress made in processing his command. For example, if the user specifies a file copy/file download operation, a progress bar can be displayed to display the status. This will help the user to monitor the status of the action initiated.

Error prevention and handling common errors: Screen design should be such that users are unlikely to make a serious error. Highlight only actions relevant to current context. Allowing user to select options rather than filling up details. Don't allow alphabetic characters in numeric fields. In case of error, it should allow user to undo and offer simple, constructive, and specific instructions for recovery.

Allow reversal of action: Allow user to reverse action committed. Allow user to migrate to previous screen.

Reduce effort of memorization by the user: Do not expect user to remember information. A human mind can remember little information in short term memory. Reduce short term memory load by designing screens by providing options clearly using pull-down menus and icons.

Relevance of information: The information displayed should be relevant to the present context of performing certain task.

Screen size: Consideration for screen size available to display the information. Try to accommodate selected information in case of limited size of the window.

Minimize data input action: Wherever possible, provide predefined selectable data inputs.

Help: Provide help for all input actions explaining details about the type of input expected by the system with example.

Attractiveness: A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

2.2 CODING

2.2.1 INFORMATION HIDING

Information hiding is a powerful programming technique because it reduces complexity. One of the chief mechanisms for hiding information is encapsulation -- combining elements to create a larger entity. The programmer can then focus on the new object without worrying about the hidden details. In a sense, the entire hierarchy of programming languages, from machine languages to high-level languages, can be seen as a form of information hiding.

Information hiding is also used to prevent programmers from intentionally or unintentionally changing parts of a program.

2.2.2 PROGRAMMING STYLE

The aim of programming style is to optimize the code with desired results. This must be presented in the best possible manner. Some general rules used in the programming style are:

1. NAMING: In a program, Care should be taken to name the module, process, and variables and so on. The naming style should not be cryptic and non-representative.

The name should represent the entity completely without confusion. Avoid cryptic names, unknown acronyms, or names totally unrelated to the entity. For example, purchase order should be named PO and not PRO, POrder, Vender PO and so on.

2. CONTROL CONSTRUCTS: It is desirable that as much as possible single-entry and single exist constructs used. In every language there are few control constructs, we should use a few standard control constructs rather than using a wide variety of control constructs.

3. INFORMATION HIDING: Only access functions to data should be made visible and data should be hidden behind these functions.

4. Gotos : Gotos should be used sparingly and in a disciplined manner. If a goto must be used, forward transfers (or a jump to a later statement) are more acceptable than a backward jump.

5. User Defined Type: Modern languages allow user to define types like the enumerated type. When such facilities are available, these should be used wherever applicable.

6. Nesting: In control constructs, 'if-then-else' are used to construct a control, based on given conditions. If the condition is satisfied one action is proposed; if not, then another action is proposed. If this condition-based nesting is too deep, the code becomes very complex.

7. Module Size: The module size should be uniform. Its size should not be too small or too big. If the module is too large, it is not functionally cohesive and if it is too small it might lead to unnecessary overhead. Module size should be based on the principle of cohesion and coupling.

8. Program Layout: A good layout helps to read the programs faster and to understand it better. The layout should be organized using proper indentation, blank spaces and parenthesis to enhance readability.

9. Module Interface: A module with a complex interface (has multiple functions) should be carefully examined and avoided. If a complex interface has more than five parameters then it should be carefully examined and they must be broken into multiple modules with a similar interface.

10. Robustness: A program is robust if it does something planned even for exceptional conditions. A program might encounter exceptional conditions in such forms as incorrect input, the incorrect value of some variable, and overflow. If such situations do arise, the program should not just "crash" or "core dump"; it should produce some meaningful message and exit gracefully.

11. Side-effects: When a module is invoked, it sometimes has side effects of modifying the program state beyond the modification of parameter, listed in the module interface definition. Such side effects should be avoided wherever possible and if a module has side effects, they should be properly documented.

2.2.3 INTERNAL DOCUMENTATION

Internal documentation is the one that talks in detail about **how** the code does whatever it function is. It describes the data structures, algorithms, and control flow in the programs.

Internal documentation is usually formed by:

1. Name, type, and purpose of each variable and data structure used in the code
2. Brief description of algorithms, logic, and error-handling techniques
3. Information about the required input and expected output of the program
4. Assistance on how to test the software
5. Information on the upgrades and enhancements in the program.

External documentation, on the other hand, focuses on **what** the problem to be solved is, and which approach is used in solving it. This can be a graph detailing the execution, a list of algorithms, dependencies, as well as meta-data such as the author, the date, and so forth

2.2.4 MONITORING AND CONTROL FORM CODING

Code reviews are mainly designed to detect defects that are originated during coding phase; they can also be used to detect defects in detached design. It was started with the purpose of detecting defects in the code.

After the successfully completion of code, code inspection and reviews are held. Activities like code reading, symbolic execution and static analysis should be performed and defects found by these techniques corrected before starting the code reviews. The main aim of this is to save human time and effort.

The main motivation for this is to save human time and effort, which would otherwise be spent in detecting errors that a compiler or static analyzer can detect.

Code defects can be divided into two groups: logic and control defects and data operations and computations defects. Examples of logic and control defects are unreachable code, incorrect predicate, infinite loops, improper nesting of loops, and unreferenced labels etc.,

Examples of data operations and computations defects are incorrect access of array components, missing validity tests for external data, improper initialization, misuse of variables, etc.

2.2.5 STRUCTURED PROGRAMMING

Structured programming refers to a general methodology of writing good programs. A good program is one, which has the following properties:

- It should perform all the desired actions.
- It should be reliable i.e., perform the required actions within acceptable margins of error.
- It should be clear i.e., easy to read and understand.
- It should be easy to modify.
- It should be implemented within the specified schedule and budget.

Structured programs have the single-entry single-exit property. This feature helps in reducing the number of paths for flow of control. If there are arbitrary paths for the flow of control, the program will be difficult to read, understand, debug and maintain.

A program is of two types: 'Static Structure' and 'Dynamic Structure'. The static structure is the structure of the text of the program, which is usually just a linear organization of statements of the program. The dynamic structure of the program is the sequence of statements executed during the execution of the program.

Both static and dynamic structure is a sequence of statements. The only difference is that the sequence of statements in static structure is fixed whereas in dynamic structure it is not fixed. That means dynamic sequence of statements can change from execution to execution.

Objectives of Structured Programming:

The goal of structured programming is to ensure that the static structure and the dynamic structures are the same. That is, the objective of structured programming is to write programs so that the sequence of statements executed during the execution of a program is the same as the sequence of statements in the text of that program. As the statements in the program text are linearly organized, the objective of structured programming becomes developing programs whose control flow during execution is linearized and follows the linear organization of the program text.

Clearly, no meaningful program can be written as a sequence of simple statements without any branching or repetition (which also involves branching). Control flow can be achieved by making use of structured constructs. In structured programming, a statement is not a simple assignment statement, it is a structured element.

Principles of Structured Programming:

All structured program methods are based upon the two fundamental principles 'stepwise refinement' and "three structured control constructs". The objective of program design is to transform the required function of the program into a set of instructions, which can easily be translated into chosen programming language. The process of stepwise refinement is such an approach that the stated program function is broken down into subsidiary functions in progressively increasing levels of detail until the lowest level functions are achievable in the programming language.

The second principle of structured program design is that any program can be constructed using only three structured control constructs. The constructs are selection, iterations and sequence. Any program independent of technology platform can be written using these constructs, i.e. Selection, repetition, sequence. These structures are basis to structured programming.

An iteration is a program component which has only one part, occurring zero or more times. The number of times the sub-component will be executed depends upon when the condition becomes false. This situation can occur on first entry or after many repetitions of the program.

Key Features of Structured Programming:

The key feature of a structured program is that , it has a single entry and single exit. We can therefore Hence the program statement can be studied in sequence. The most commonly used single entry and single exit statement are:

Selection: if customer type is X then price is Rs.100 else price is Rs.90

Iteration: While customer type is X do use price formula P1. Repeat P1 till type is X.

Sequencing: Task 1, Task 2, Task 3.

Extensive use of these statements in the program construct creates a linear flow. If readability and verification are the essence of a good construct, then a structured program is the method to achieve it.

Advantages of Structured programming:

It is very convenient to put logic systematically into the program. Due to the case of handling complex logic, the user, reader and programmer understand the program easily. It is easy to verify, conduct reviews and test the structured programs in an orderly manner. If errors are found, they are easy to locate and correct.

2.3 SOFTWARE PLANNING

2.3.1 SOFTWARE METRICS - DEFINITION

Metrics deal with measurement of the software process and the software product. Metrics quantify the characteristics of a process or a product. Metrics are often used to estimate project cost and project schedule. Example: Lines of code (LOC), pages of documentation, number of man-months, number of test cases, number of input forms.

2.3.2 TYPES OF METRICS

Metrics can be broadly divided into three categories namely, project metrics, product metrics and process metrics.

Project Metrics: Project metrics describe the project characteristics and execution. Examples: (i) Number of Software developers (ii) Staffing pattern over the life cycle of the software (iii) Cost and Schedule (iv) Productivity

Product metrics: Product metrics are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program or the number of pages of documentation produced.

Process Metrics: Process Metrices describe the effectiveness and quality of the processes that produces the software product. Examples are: (i) Effort required in the process (ii) Time to produce the product (iii) Number of defects found during testing (iv) Maturity of the process.

Another way of classification of metrics is primitive metrics and derived metrics.

Primitive metrics are directly observable quantities like lines of code (LOC), number of man-hours etc., Derived metrics are derived from one or more of primitive metrics like lines of code per man-hour, errors per thousand lines of code.

2.3.4 PRODUCT AND PROJECT METRICS

Product Metrics

In software development process, a working product is developed at the end of each successful phase. Each product can be measured at any stage of its development. Metrics are developed for these products so that they can indicate whether a product is developed according to the user requirements. If a product does not meet user requirements, then the necessary actions are taken in the respective phase.

Product metrics help software engineer to detect and correct potential problems before they result in catastrophic defects. In addition, product metrics assess the internal product attributes to know the efficiency of the following.

- Analysis, design, and code model
- Potency of test cases
- Overall quality of the software under development.

Project Metrics

Project metrics enable the project managers to assess current projects, track potential risks, identify problem areas, adjust workflow, and evaluate the project team's ability to control the quality of work products.

Project metrics serve two purposes. (i) They help to minimize the development schedule by making necessary adjustments to avoid delays and alleviate potential risks and problems. (ii) These metrics are

used to assess the product quality on a regular basis-and modify the technical issues, if required. As the quality of the project improves, the number of errors and defects are reduced, which in turn leads to a decrease in the overall cost of a software project.

Often, the first application of project metrics occurs during estimation. Here, metrics collected from previous projects act as a base from which effort and time estimates for the current project are calculated. As the project proceeds, original estimates of effort and time are compared with the new measures of effort and time. This comparison helps the project manager to monitor (supervise) and control the progress of the project.

As the process of development proceeds, project metrics are used to track the errors detected during each development phase. For example, as software evolves from design to coding, project metrics are collected to assess quality of the design and obtain indicators that in turn affect the approach chosen for coding and testing. Also, project metrics are used to measure production rate, which is measured in terms of models developed, function points, and delivered lines of code.

2.3.5 FUNCTION POINT METRICS

The function point metrics is used to measure the functionality delivered by the system, estimate the effort, predict the number of errors, and estimate the number of components in the system. Function point is derived by using a relationship between the complexity of software and the information domain value. Information domain values used in function point include the number of external inputs, external outputs, external inquires, internal logical files, and the number of external interface files.

2.3.6 SOFTWARE PROJECT ESTIMATION

Software project estimation is the process of estimating various resources required for the completion of a project. Effective software project estimation is an important activity in any software development project. Underestimating software project and understaffing is often leads to low quality deliverables, and the project misses the target deadline leading to customer dissatisfaction and loss of credibility to the company. On the other hand, overstaffing a project without proper control will increase the cost of the project and reduce the competitiveness of the company.

2.3.7 STEPS FOR ESIMATION

STEPS FOR ESTIMATION

- **Estimating the size of project:** There are many procedures available for estimating the size of a project which are based on quantitative approaches like estimating Lines of Code or estimating the functionality requirements of the project called Function Point.
- **Estimating effort based on man-month or man-hour:** Man-month is an estimate of personal resources required for the project.

- **Estimating schedule in calendar** days/months/year based on total man-month required and manpower allocated to the project.
  Duration in calendar month = Total man-months / Total manpower allocated.

- **Estimating total cost of the project** depending on the above and other resources.

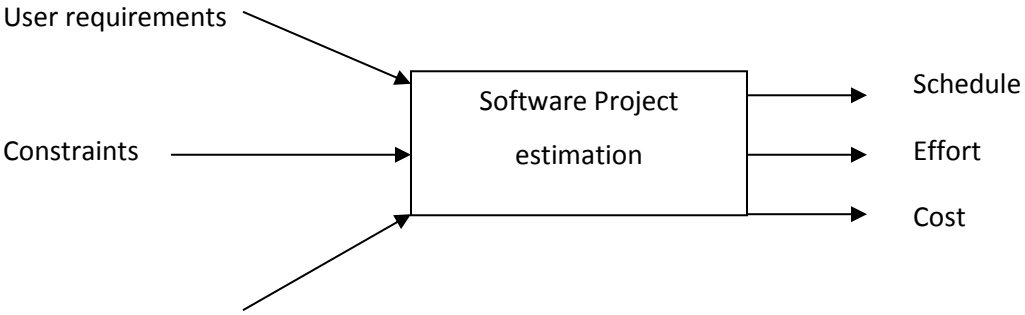


Fig 2.10 Software project estimation

ESTIMATING THE SIZE

Customer's requirements and system specification forms a baseline for estimating the size of a software. At a later stage of the project, system design document can provide additional details for estimating the overall size of software.

- *The ways to estimate project size can be through past data from an earlier developed system. This is called estimation by analogy.*
- The other way of estimation is through product feature/functionality. The system is divided into several subsystems depending on functionality, and size of each subsystem is calculated.

ESTIMATING EFFORT

Once the size of software is estimated, the next step is to estimate the effort based on the size. The estimation of effort can be made from the organizational specifics of software development life cycle. The development of any application software system is more than just coding of the system. Depending on deliverable requirements, the estimation of effort for project will vary. Efforts are estimated in number of man-months.

- The best way to estimate effort is based on the organization's own historical data of development process. Organizations follow similar development life cycle for developing various applications.
- If the project is of a different nature which requires the organization to adopt a different strategy for development then different models based on algorithmic approach can be devised to estimate effort.

ESTIMATING SCHEDULE

The next step in estimation process is estimating the project schedule from the effort estimated. The schedule for a project will generally depend on human resources involved in a process. Efforts in man-

months are translated to calendar months. Schedule estimation in calendar-month can be calculated using the following model [McConnel]:

$$\text{Schedule in calendar-month} = 3.0 * (\text{man-months})^{1/3}$$

The parameter 3.0 is variable, used depending on the situation which works best for the organization.

Estimating Cost

The cost of a project is derived not only from the estimates of effort and size but from other parameters such as hardware, travel expenses, telecommunication costs, training cost etc., should also be taken into account.

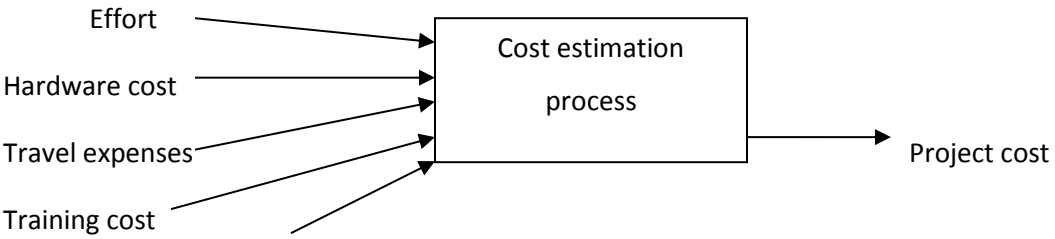


Fig No 2.11Cost estimation process

There is always some uncertainty associated with all estimation techniques. The accuracy of project estimation will depend on the following:

- Accuracy of historical data used to project the estimation
- Accuracy of input data to various estimates
- Maturity of organization's software development process.

2.3.8 REASON FOR POOR AND INACCURATE ESTIMATION

- Requirements are imprecise. Also, requirements change frequently
- The project is new and is different from past project handled.
- Non-availability of enough information about past projects.
- Estimates are forced to be based on available resources.

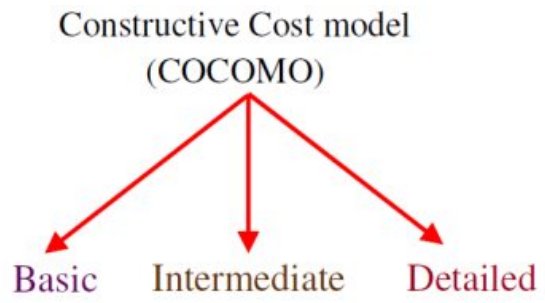
2.3.9 PROJECT ESTIMATION GUIDELINES

- Preserve and document data pertaining to organization's past projects.
- Allow sufficient time for project estimation especially for bigger projects.
- Prepare realistic developer-based estimate. Associate people who will work on the project to reach at a realistic and more accurate estimate.
- Use software estimation tools.
- Re-estimate the project during the life cycle of development process.
- Past Analysis mistakes in the estimation of projects.

2.3.10 MODELS FOR ESTIMATION - COCOMO MODEL

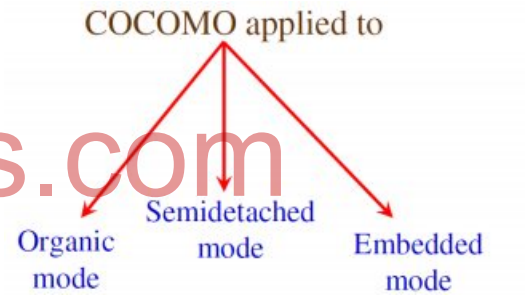
COCOMO Model

COCOMO stands for **Constructive Cost Model**. It was introduced by Barry Boehm. It provides the following three level of models:



- **Basic COCOMO:** A single – value model that computes software development cost as a function of estimate of LOC.
- **Intermediate COCOMO:** This model computes development cost and effort as a function of program size (LOC) and a set of cost drivers.
- **Detailed COCOMO:** This model computes development effort and cost which incorporates all characteristics of intermediate level with assessment of cost implication on each step of development (analysis, design, testing etc.,).

This model may be applied to three classes of software projects as given below:



Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

The Comparison of three COCOMO Models

BASIC COCOMO MODEL

Basic COCOMO model takes the form

$E = a (KLOC)^b$

$D = c (E)^y$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a, b, c and y are given in table .

Software Project	a	b	c	y
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

When effort and development time are known, the average staff size to complete the project may be calculated as:

Average staff size (SS) = E/D Persons

When project size is known, the productivity level may be calculated as:

Productivity (P) = KLOC /E

Intermediate Model

Cost drivers

(i) Product Attributes

_ Required s/w reliability
_ Size of application database
_ Complexity of the product

ii) Hardware Attributes

_ Run time performance constraints
_ Memory constraints
_ Virtual machine volatility
_ Turnaround time

(iii) Personal Attributes

_ Analyst capability
_ Programmer capability
_ Application experience
_ Virtual m/c experience
_ Programming language experience

(iv) Project Attributes

_ Modern programming practices
_ Use of software tools
_ Required development Schedule

Intermediate COCOMO equations

$E = a (KLOC)^b \times EAF$

$D = c (E)^y$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a, b, c and y are given in table .

Software Project	a	b	c	y
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

EAF is the multiplication of ratings of 15 cost drivers.

What are the methods of cost estimation?

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

Non- Algorithmic models: Estimation in these models depends on the prior experience and domain knowledge of the project manager. These models do not use mathematical equations to estimate the cost of a software project. Various non-algorithmic models are expert judgement, estimation by analogy and price to win.

2.3.12 AUTOMATED TOOLS FOR ESTIMATION

The estimation tools, which estimate cost and effort, allow the project managers to perform what if analysis. Estimation tools may only support size estimation or conversion of size to effort and schedule to cost. There are more than dozens of estimation tools available. But, all of them have the following common characteristics:

- Quantitative estimates of project size (e.g. LOC).
- Estimates such as project schedule, cost.
- Most of them use different models for estimation.

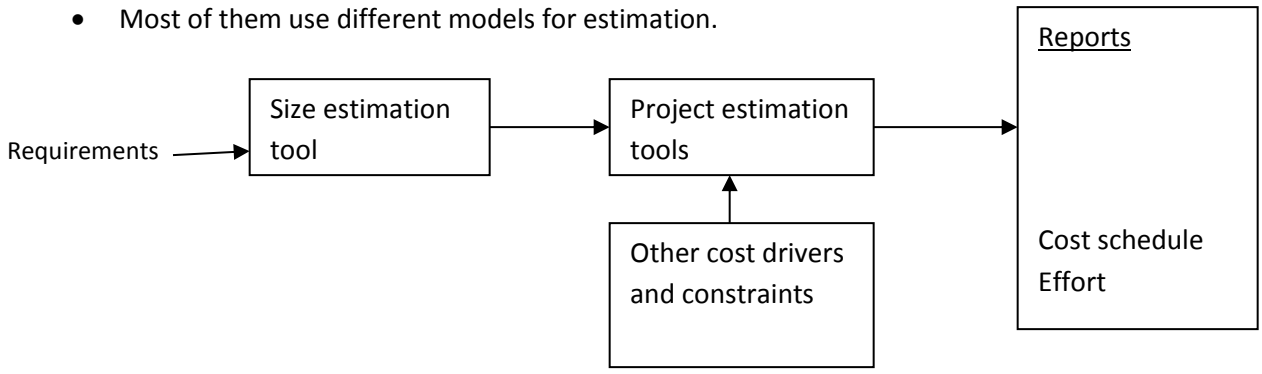


Fig No 2-12 Typical structure of estimation tools

No estimation tool is the solution to all estimation problems. One must understand that the tools are just to help the estimation process.

Problems with Models:-

Most models require estimate of software product size. However, software size is difficult to predict early in the development lifecycle. Many models use LOC for sizing, which is not measurable during requirements analysis or project planning. Although, function points and object points can be used earlier in the lifecycle, these measures are extremely subjective.

Size estimates can also be very inaccurate. Methods of estimation and data collection must be consistent to ensure an accurate prediction of product size. Unless the size metrics used in the model are the same as those used in practice, the model will not yield accurate results. The following table gives some estimation tools:

Automated Estimation Tools

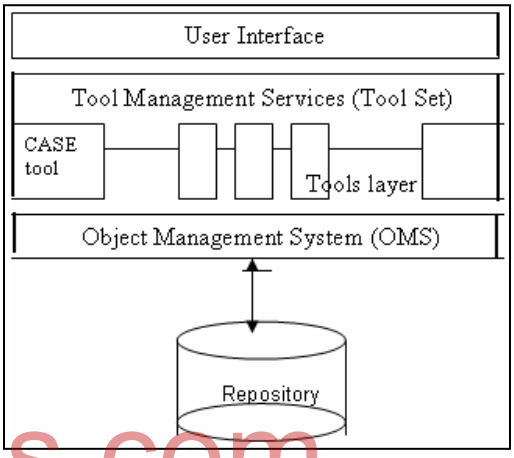
Tool	Functionality / Remark
EstimatorPal	EstimatorPal is a software tool that assists software developers and is used to estimate the effort required on various activities. This tool facilitates use of the following estimation techniques – Effort estimation tools which supports functions point analysis technique, objects points technique , use case points technique and task based estimation technique
Estimate easy use case	Effort estimation tool based on use cases.
USC COCOMO II	Based on COCOMO
ESTIMACS	Provides estimates of the effort, duration, cost and personnel requirements for maintenance and new application development projects.
Checkpoint	Guides the user through the development of a software project estimate and plan.
Function Point Workbench	Automated software estimation tools, implementing function point sizing techniques, linking project estimation to project management initiatives, and collecting historical project data to improve future estimation efforts.
ESTIMATE Professional	Based on Putnam, COCOMO II
SEER-SEM	Predicts, measures and analyses resources, staffing schedules, risk and cost for software projects
ePM.Ensemble	Support effort estimation, among other things.
CostXpert	Based on COCOMO

2.4 CASE

2.4.1 CASE AND ITS SCOPE

CASE stands for Computer Aided Software Engineering. CASE is a tool which aids a software engineer to maintain and develop software. The workshop for software engineering is called an integrated project support environment (IPSE) and the tool set that fills the workshop is called CASE. CASE tools are the software engineering tools that permit collaborative software development and maintenance. CASE is an automated support tool for the software engineers in any software engineering process.

2.4.2 ARCHITECTURE OF CASE ENVIRONMENT



The architecture of CASE environment is illustrated in figure no 2-13. 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:

1. **User Interface.** It provides a consistent framework for accessing different tools. So it is easier for the user to interact with different tools and reduces learning time of how the different tools are used.

2. **Tools Management Services (Tools Set).** The tools set section holds the different types of improved quality tools. The tools layer incorporates a set of tools management services with the CASE tool themselves. Tools Management Services (TMS) control the behavior 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.

3. **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., 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.

4. **Repository.** It is the CASE database and the access control functions that enable the OMS to interact with the database. The word CASE repository is referred in different ways such as project database, IPSE database, data dictionary, CASE database and so on.

2.4.3 BUILDING BLOCKS FOR CASE

The building blocks for CASE are illustrated in figure 2-14.

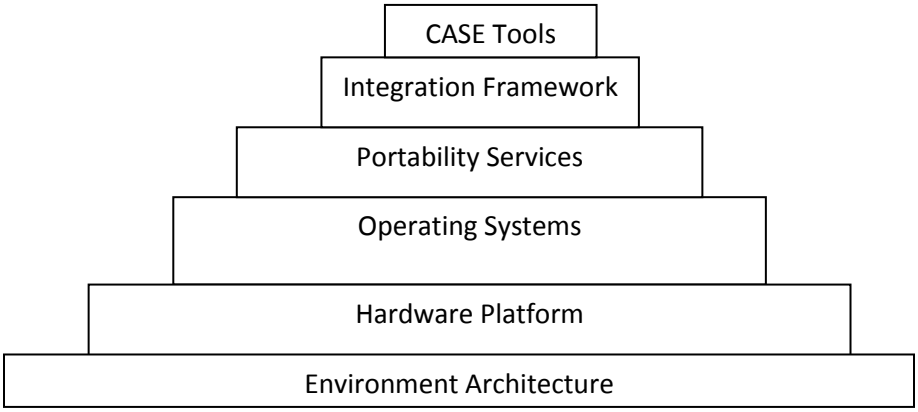


Fig No 2-14 CASE Building blocks

1. **Environment Architecture.** The environment architecture, composed of the hardware platform and operating system support including networking and database management software, lays the groundwork for CASE but the CASE environment itself demands other building blocks.
2. **Portability Services.** A set of portability service provides a bridge between CASE tools and their integration framework and the environment architecture. These portability services allow the CASE tools and their integration framework to migrate across different hardware platforms and operating systems without significant adaptive maintenance.
3. **Integration framework.** It is a collection of specialized programs that enables individual CASE tools to communicate with one another, to create a project database.
4. **Case Tools.** CASE tools are used to assist software engineering activities (like analysis modeling, code generation etc.) either communicating with other tools, project database (integrated CASE environment) or as point solutions.

2.4.4 CASE SUPPORT IN SOFTWARE LIFE CYCLE

There are various types of support that provides during the different phases of a software life cycle.

1. **Prototyping Support.** The prototyping is used to understand the requirements of complex software products, to market new ideas and so on. The prototyping CASE tools requirements are as follows:
    - Define user interaction
    - Define the system control flow
    - Store and retrieve data required by the system
    - Incorporate some processing logic

Prototyping tools support the following features:

- Prototyping CASE tool is used to develop graphical user interface (GUI) development. The user should be allowed to define all data entry forms, menus and control.
- Integrate well with the data dictionary of a CASE environment.
- It should be able to integrate with the external user-defined modules written in high-level languages.
- The user should be able to define the sequence of states through which a created prototype can run.
- The prototype should support mock up run of the actual system and management of the input and output data.
- The prototype should support mock up run of the actual system and management of the input and output data.

2. **Structured Analysis and Design.** A CASE tool should support one or more of the structured analysis and design techniques. It should also support making of the fairly complex diagrams and preferably through a hierarchy of levels. The tool must also check the incompleteness, inconsistencies and anomalies across the design and analysis through all levels of analysis hierarchy.

   Analysis and design tools enable a software engineer to create models of the system to be built. The models contain a representation of data, function, and behavior (at the analysis level) and characterizations of data, architectural, component level, and interface design. By performing consistency and validity checking on the models, analysis and design tools provide a software engineer with some degree of insight into the analysis representation and help to eliminate errors before they propagate into the design, or worse, into implementation itself.

3. **Code Generation.** A support expected from a CASE tool during the code generation phase comprises the following:
   - The CASE tool should support generation of module skeletons or templates in one or more popular programming languages.
   - The tool should generate records, structures, class definitions automatically from the contents of the data dictionary in one or more popular programming languages.
   - It should be able to generate database tables for relational database management systems.
   - The tools should generate code for user interface from prototype definitions for X- Windows and MS Window based applications.

4. **Test CASE Generator.** The CASE tool for test case generator should have following features:
   - It should support both design and requirement testing.
   - It should generate test set reports in ASCII formats, which can be directly imported into the test plan document.

Under testing phase, Test management tools are used to control and coordinate software testing for each of the major testing steps. Testing tools manage and coordinate regression testing, perform comparison that ascertain differences between actual and expected output. In addition to the functions noted, many test management tools also serve as generic test drivers. A test driver reads one or more test cases from a testing file, formats the test data to conform to the needs of the software under test, and then invokes the software to be tested.

2.4.5 OBJECTIVES OF CASE

1. **Improve Productivity.** Most organization use CASE to increase the speeds with which systems are designed and developed. Tools increase the analysts productivity by reducing the time needed to document, analyze, and construct information system.
2. **Improve information System Quality.** When tools improve processes, they usually improve the results as well.

   - 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.
   - Shortens the overall construction process
   - Improve software portability across environments.
   - Through reverse engineering and re-engineering, CASE products extend the file of existing systems.

   Despite the various driving forces (objectives) for the adoption of CASE, there are many resisting forces also that preclude many organizations from making investment in CASE.

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 compared to moving quickly into coding.
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 early stage of the project.
   - CASE tools cannot easily share information between tools
   - Lack of methodology standards within organizations, CASE products forces analysts to follow a specific methodology for system development
   - Lack of confidence in CASE products

Despite these issues, in 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.

2.4.6 CHARECTERISTICS OF CASE TOOLS

A Standard Methodology: A CASE tool must support a standard software development methodology and standard modeling techniques. In the present scenario most of the CASE tools are moving towards UML.

Flexibility: Flexibility in use of editors and other tools. The CASE tool must offer flexibility and the choice for the user of editor's development.

Strong Integration: The CASE tools should be integrated to support all the stages, this implies that if a change is made at any stage, for example, in the model, it should get reflected in the code documentation and all related design and other documents. This providing a cohesive environment for software development.

Integration with testing Software: The CASE tools must provide interfaces for automatic testing tools that take care of regression and other kind of testing software under the changing requirements.

Support for reverse Engineering: A CASE tools must be able to generate complex models from already generated code.

On line Help: The CASE tools provide an online tutorial.

Documentation Support: The delivered documents should be organized graphically and should be able to incorporate text and diagrams from the central repository. This helps in producing up-to-date documentation. The CASE tool should integrate with one or more of the commercially available desktop publishing packages. It should be possible to text, graphics, and table data dictionary reports to the DTP package in standard forms such as PostScript.

Data Dictionary Interface: The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

2.4.7 LIST OF CASE TOOLS

Application	Case Tool	Purpose of Tool
1. Planning	Excel spreadsheet, MS-Project, PERT/CPM Network, Estimation tools	Functional Application: Planning, Scheduling, control
2. Editing	Diagram editors, Text editors, Word Processors	Speed and Efficiency
3. Testing	Test data Generators, File Comparators	Speed and Efficiency

4. Prototyping	High level Modeling language, User Interface Generators	Confirmation and Certification of RDD and SRS
5. Documentation	Report Generators, Publishing imaging, PPT presentation	Faster structural documentation with quality of presentation
6. Programming and Language Processing Integration	Program Generators, Code Generators, Compilers, Interpreters Interface, Connectivity	Programming of high quality with no errors, System Integration
7. Templates	---	Guided Systematic development
8. Re-engineering tools	Cross reference systems, program re-structuring systems	Reverse- engineering to find structure, design and design information
9. Program analysis tool	Cross reference generators Static analyzers, dynamic analyzers	Analyses risks, functions, features

2.4.8 CATEGORIES, ADVANTAGES AND DISADVANTAGES OF CASE TOOLS

The schematic diagram of CASE tools is drawn in fig 2-15.

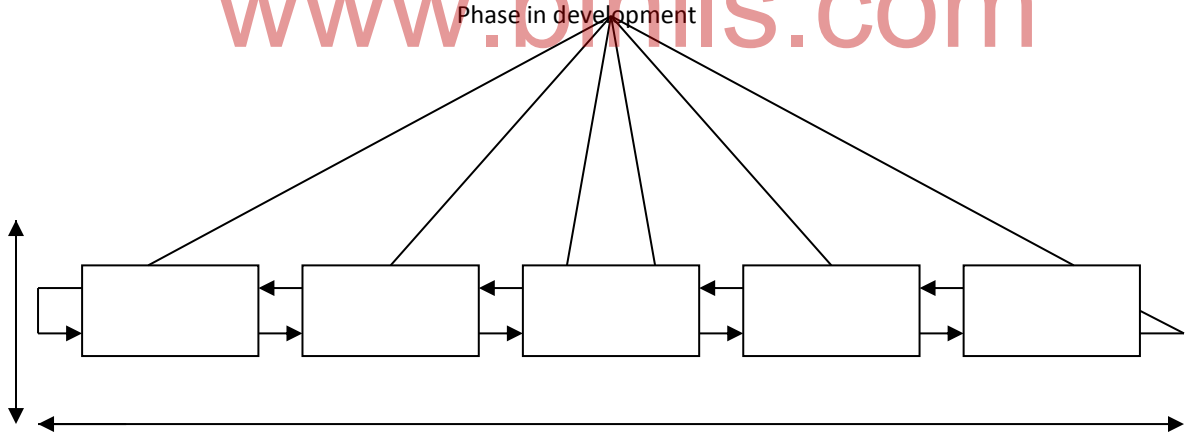


Fig No 2-15. Categories of CASE tools

Horizontal Tools

CASE tools are divided into the following two categories.

1. Vertical CASE tools
2. Horizontal CASE tools

1. **Vertical CASE Tools.** Vertical CASE tools provide support for certain activities within a single phase of the software life cycle. here are two subcategories of vertical CASE tools

i. *First Category.* It is the set of tools that are within one phase of life cycle. These tools are important so that development in each phase can be as quick 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 not move on the next phase as appropriate.

2. **Horizontal CASE Tools.** These tools support 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

a) UPPER CASE Tools/ Front-end CASE Tools. CASE tools designed to support the analysis and design phases of SDLC. All the analysis, design and specification tools are front-end tools. These tools also include computer-aided diagramming tools oriented towards a particular programming design methodology, more recently including object-oriented design.

The general types of upper CASE tools are listed below:

- **Diagramming tools:** Diagramming tools enable system process, data and control structures to be represented graphically. They strongly support analysis and documentation of application requirements.
- **Form and Report generator tools:** They support the creation of system forms and reports in order to show how systems will "look and feel" to users.
- **Analysis tools:** Analysis tools enable automatic checking for incomplete, inconsistent, or incorrect specification in diagrams, forms and reports.

b) Lower CASE or Back-End Tools. CASE tools designed to support the implementation and maintenance phases of SDLC. All the generator, translation and testing tools are back-end tools. The general types of Lower CASE tools are :

- **Code Generators:** Code generators automate the preparation of computer software. Code generation is not yet perfect. Thus, the best generator will produce approximately 75 percent of the source code for an application. Hand coding is still necessary.
- **Cross life cycle CASE or Integrated Tools.** CASE tools used to support activities that occur across multiple phases of the SDLC. While such tools include both front-end and back-end capabilities, they provide an efficient environment for the creation, storage, manipulation, and documentation of systems.
- **Reverse Engineering Tools.** These tools build bridges from lower CASE tools to upper CASE tools. They help in the process of analyzing existing applications, performance and database code to create higher level representations of the code.

ADVANTAGES OF CASE TOOLS

- Improved Productivity
- Better documentation
- Improved accuracy
- Intangible benefits
- Improved quality
- Reduced lifetime maintenance
- Opportunity to non-Programmers
- Reduced cost of software
- Produce high quality and consistent documents
- Impact on the style of a working of company
- Reduce the drudgery in a software engineer's work
- Increase speed of processing
- Easy to program software
- Improved co-ordination among staff members who are working on a large software project
- An increase in project control though better planning, monitoring and communication.

DISADVANTAGES OF CASE TOOLS

1. **Purchasing of Case Tools is Not an Easy Task.** Its cost is very high. Due to this reason, small software development firm do not invest in case tools.
2. **Learning Curve.** In general cases programmer productivity may fail in initial phases of implementation as user need time to learn this technology.
3. **Tool Mix.** It is important to make proper selection of case tools to get maximum benefit from the case tools, so wrong selection may lead to wrong result.

SUMMARY

- Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language.
- The objectives of software design are: Correctness, Verifiability, Completeness , Traceability, Efficiency and Simplicity
- The objective of architectural design is to develop a model of software architecture which gives overall organization of program module in the software product.
- Software can be divided into relatively independent, named and addressable component called a module.
- A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules
- Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:(i) It is usually difficult to identify the different modules of the software from its flow chart representation. (ii) Data interchange among different modules is not represented in a flow chart. (iii) Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.
- Coupling is the measure of the degree of interdependence between modules.
- *The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component.*
- Interface design is one of the most important parts of software design. The user interaction with the system takes place through various interfaces provided by the software product.
- Information hiding is also used to prevent programmers from intentionally or unintentionally changing parts of a program.
- Internal documentation is the one that talks in detail about **how** the code does whatever it function is. External documentation, on the other hand, focuses on **what** the problem to be solved is, and which approach is used in solving it.
- Code reviews are mainly designed to detect defects that are originated during coding phase; they can also be used to detect defects in detached design.
- Code detects can be divided into two groups: logic and control detects and data operations and computations defects.
- Examples of logic and control defects are unreachable code, incorrect predicate, infinite loops, improper nesting of loops, and unreferenced labels etc.,
- Examples of data operations and computations defects are incorrect access of array components, missing validity tests for external data, improper initialization, misuse of variables, etc.
- Structured programs have the single-entry single-exit property. This feature helps in reducing the number of paths for flow of control

- Metrics deal with measurement of the software process and the software product. Metrics quantify the characteristics of a process or a product. Metrics are often used to estimate project cost and project schedule.
- Metrics can be broadly divided into three categories namely, project metrics, product metrics and process metrics.
- Function point is derived by using a relationship between the complexity of software and the information domain value.
- Software project estimation is the process of estimating various resources required for the completion of a project.
- *The ways to estimate project size can be through past data from an earlier developed system. This is called estimation by analogy.*
- COCOMO stands for Constructive Cost Model. It was introduced by Barry Boehm. It is the best known and most thoroughly documented of all software cost estimation models.
- Intermediate COCOMO model is expected to give more accurate estimate than the basic COCOMO model
- CASE stands for Computer Aided Software Engineering. CASE is a tool which aids a software engineer to maintain and develop software.
- CASE tools are divided into the following two categories.(i) Vertical CASE tools and (ii) Horizontal CASE tools

REVIEW QUESTIONS

PART – A (2 Marks)

1. Define the term “Software Design”.
2. What do you mean by depth and width with respect to architecture design.
3. What is fan-in & fan-out?
4. What is a module?
5. Write any two uses of structure chart?
6. Give the format of a structure chart.
7. Define coupling. What are the types of coupling?
8. Define cohesion. What are the types of cohesion?
9. Write the difference between tightly coupling and loosely coupling.
10. What is interface design?
11. What do you mean by software metric?
12. Expand the term LOC.
13. How the efforts are estimated?
14. Write down the formula to calculate “Schedule in calendar-month”.

15. Expand the term “COCOMO” and “CASE”
16. What is the use of CASE?

PART – B (3 Marks)

13. How the modules are classified according to the activation mechanism.
14. Explain the symbols used in a structure chart.
15. What are the different types of coupling?
16. What are the disadvantages of high coupling?
17. What are the disadvantages of low coupling?
18. State any three advantages of a graphical User Interface.
19. What is primitive metrics and derive metrics?
20. List down the benefits of using function points.
21. What are the points to be considered for the accuracy of project estimation.
22. State some of the reason for poor and inaccurate estimation.
23. Give the project estimation guidelines.
24. What are the common characteristics of automated tools for estimation?
25. Give some examples. Which can be automated by using CASE.
26. What are the features of Test CASE generator?

PART – C (5 Marks/ 10 Marks)

1. What are the main objectives of software design? Explain.
2. What are the main objectives of software design? Explain.
3. Briefly explain about architectural design.
4. State any five advantages of modular design.
5. What are the different types of coupling? Explain.
6. What are the points to be considered in function point metrics?
7. Explain interface design with example.
8. What are the rules for human computer interface (HCI)?
9. What are steps for estimation?
10. State some of the reasons which make the task of cost estimation difficult.
11. List down the objectives of CASE.
12. Explain different categories of case tools.
13. Discuss the advantages and disadvantages of CASE tools.
14. Explain the process of software design.
15. Briefly explain the following software design : a) Architectural design b) Modular design

16. Explain structural chat with an example.
17. Explain the various types of coupling and various types of cohesion.
18. Briefly explain about interface design.
19. List down the elements of good interface design.
20. Write down the rules for human computer interface design.
21. Briefly explain different types of metrics.
22. Briefly explain the steps used for software project estimation.
23. Explain COCOMO Model.
24. Write short notes on: a) Putnam's Model and b) Statistical Model.
25. Briefly explain Automated Estimation tools.
26. Explain the architecture of CASE Environment.
27. What are the building blocks for CASE? Explain.
28. Explain CASE support in software life cycle.
29. Explain the characteristics of CASE tools.

UNIT – III

SOFTWARE MAINTENANCE & RISK MANAGEMENT

OBJECTIVES

After reading this unit, the student will be able to

- Understand what is software maintenance.
- Analyze the need for software maintenance.
- Learn about the various software configuration management activities.
- Understand categories of software maintenance.
- Identify the factors affecting the effort.
- Identify and manage risks.
- Learn about how detect, avoid, control and recover from risks.
- Analyze the sources of risks.
- Categorize the types of risks.
- Understand the basic concept of project scheduling.
- Know about the different scheduling methods.
- Learn the basic concepts of PERT, GANT chart and other scheduling methods.

INTRODUCTION

Software maintenance is the modification of a software product after its deployment with the objective of correcting errors, enhancing performance and other attributes or adapting the product to changes in the environment. It is observed through various studies that as years pass by the cost spent over maintaining software increases.

Risk management in software engineering is related to the various future harms that could be possible on the software due to some minor or non-noticeable mistakes in software development project or process. “Software projects have a high probability of failure so effective software development means dealing with risks adequately. Risk management is the most important issue involved in the software project development.

In this unit, we will discuss about the concepts of software maintenance, various software configuration management, risk management

3.1 SOFTWARE MAINTENANCE

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updations done after the delivery of software product. delivery of the software.

3.1.1 SOFTWARE AS AN EVOLUTION ENTITY

Lehman and Belady have analysed the characteristics of evolution of several software products. They have expressed their observations in the form of laws. The important laws are given below.

Lehman's first law: A software product must change continually or become progressively less useful.
Lehman's second law: The structure of a program tends to degrade as more and more maintenance is carried out on it.
Lehman's third law: Over a program's lifetime, its rate of development is approximately constant.

3.1.2 SOFTWARE CONFIGURATION MANAGEMENT ACTIVITIES

Four component of a good Configuration Management System

1. **Configuration Identification** - It is the process of identification of Configuration Items (CI) and developing a method to uniquely identify each individual CI. It helps answers the following questions:
    - Which items are placed under configuration management?
    - What are the components of the product?
    - What is the structure (or configuration) of components in the product?
    - What are the versions of the configuration items?
2. **Configuration Control** - It is the activity of managing the product (or project's deliverables) and related documentation, throughout the lifecycle of the product. It answers the questions:
    - Which items are controlled?
    - How are changes controlled?
    - Who controls the changes?
3. **Configuration Status Accounting** - It involves the recording and reporting of all the changes to the configuration items. It tells us:
    - What is the status of proposed changes?
    - What changes have been made?
    - When were the changes made?
    - What components were affected by the change?
4. **Configuration verification and audit** - It is the process of verifying the correctness of the product and its components in order to ensure conformance to requirements. It also means verifying the correctness of the Configuration Status Accounting information. It helps to:
    - Ensure that all the configuration items have been correctly identified and accounted for.
    - Ensure that all the changes have been registered, assessed, approved, tracked, and correctly implemented.
    - Measure the effectiveness of the configuration management process.

CONFIGURATION IDENTIFICATION

The project manager normally classifies the object associated with a software development into three main categories: *controlled, pre-controlled, and uncontrolled*. Controlled objects are those, already put under configuration control. Pre-controlled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not and will not be subject to configuration control. Controllable objects include both controlled and pre-controlled object. Typical controllable objects include:

- Requirements specification document
- Designing documents
- Tools used to build the system, such as compilers, linkers, lexical analyzers, parsers, etc.
- Source code for each module
- Test cases
- Problem reports

Configuration management plan is written during the project and the planning phase lists all controlled objects.

CONFIGURATION CONTROL

Configuration control is the process of managing changes to controlled objects. The configuration control system prevents unauthorized changes to any controlled object. In order to change a controlled object such as a module, a developer can get a private copy of the module by a reserve operation. Once an object is reserved it does not allow anyone else to reserve this module until the reserved module is restored. Thus, by preventing more than one engineer to simultaneously reserve a module, the problems associated with concurrent access are solved.

The CCB is a group of people responsible for configuration management. The CCB evaluates the request based on its effect on the project, and the benefit due to change.

An important reason for configuration control is, people need a stable environment to develop a software product.

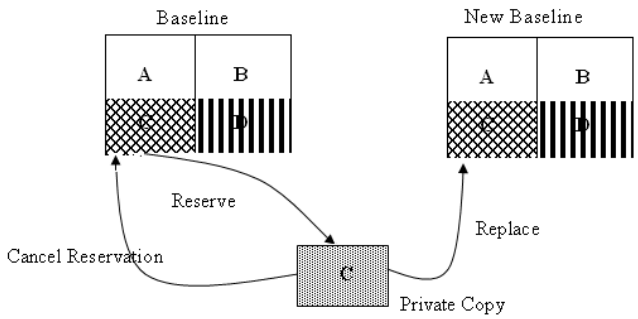


Fig No 3.1. Configuration Control

Suppose you are trying to integrate module A with the modules B and C, you cannot make progress if developer of module C keeps changing it; this is especially frustrating if a change to module C forces you to recompile A. As soon as a document under configuration control is updated, the updated version is frozen and is called a *baseline*, as shown fig 3.1.

CONFIGURATION ACCOUNTING

Configuration accounting can be explained by two concepts:

STATUS ACCOUNTING

Once the changes in baselines occur, some mechanisms must be used to record how the system evolves and what its current state is. This task is accomplished by status accounting. Its basic function is to record the activities related to the other SCM functions. Configuration status reporting (sometimes called status accounting) is an SCM task that answers the following questions:

(a) What happened?
(b) When did it happen?
(c) Who did it?
(d) What else will be affected?

A typical Configuration Status Report might include:

- A list of the configuration items that comprise a baseline
- The date when each version of each configuration item was baselined
- A list of the specifications that describe each configuration item
- The history of baseline changes including rationales for change
- A list of open change requests by configuration item
- Deficiencies identified by configuration audits
- The status of works associated with approved change requests by CI.

The key elements under status accounting are shown in fig 3.2

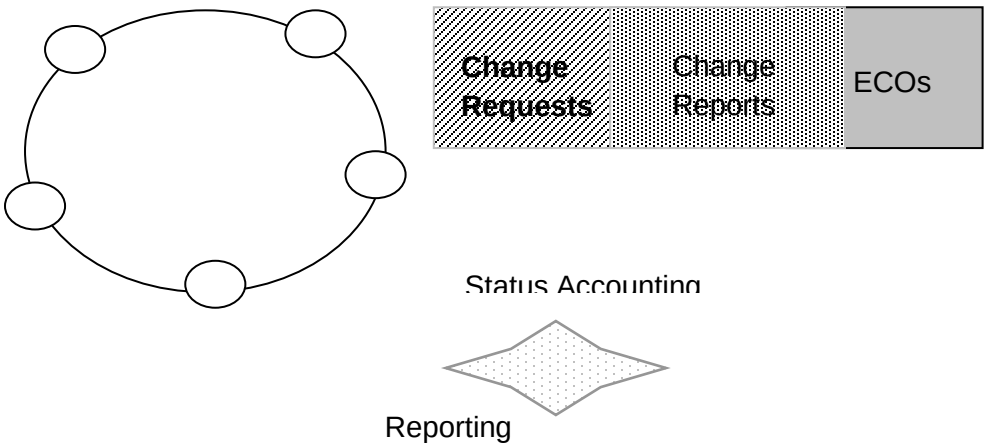


Fig No 3.2. Status Accounting

CONFIGURATION AUDIT

A software configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:

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

3.1.3 CHANGE CONTROL PROCESS

Change control combines human procedures and automated tools to provide mechanism for the control of change. The change control process is illustrated schematically in Figure 3.3.

A *change request* is submitted by user and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a *change report,* which is used by a *change control authority* (CCA).

A CCA is a person or group who makes a final decision on the status and priority of the change. An *engineering change order* (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints to be respected, and the criteria for review and audit. The object to be changed is "checked out" of the project database, the change is made, and appropriate SQA activities are applied. The object is then "checked in" to the database and appropriate version control mechanisms are used to create the next version of the software.

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

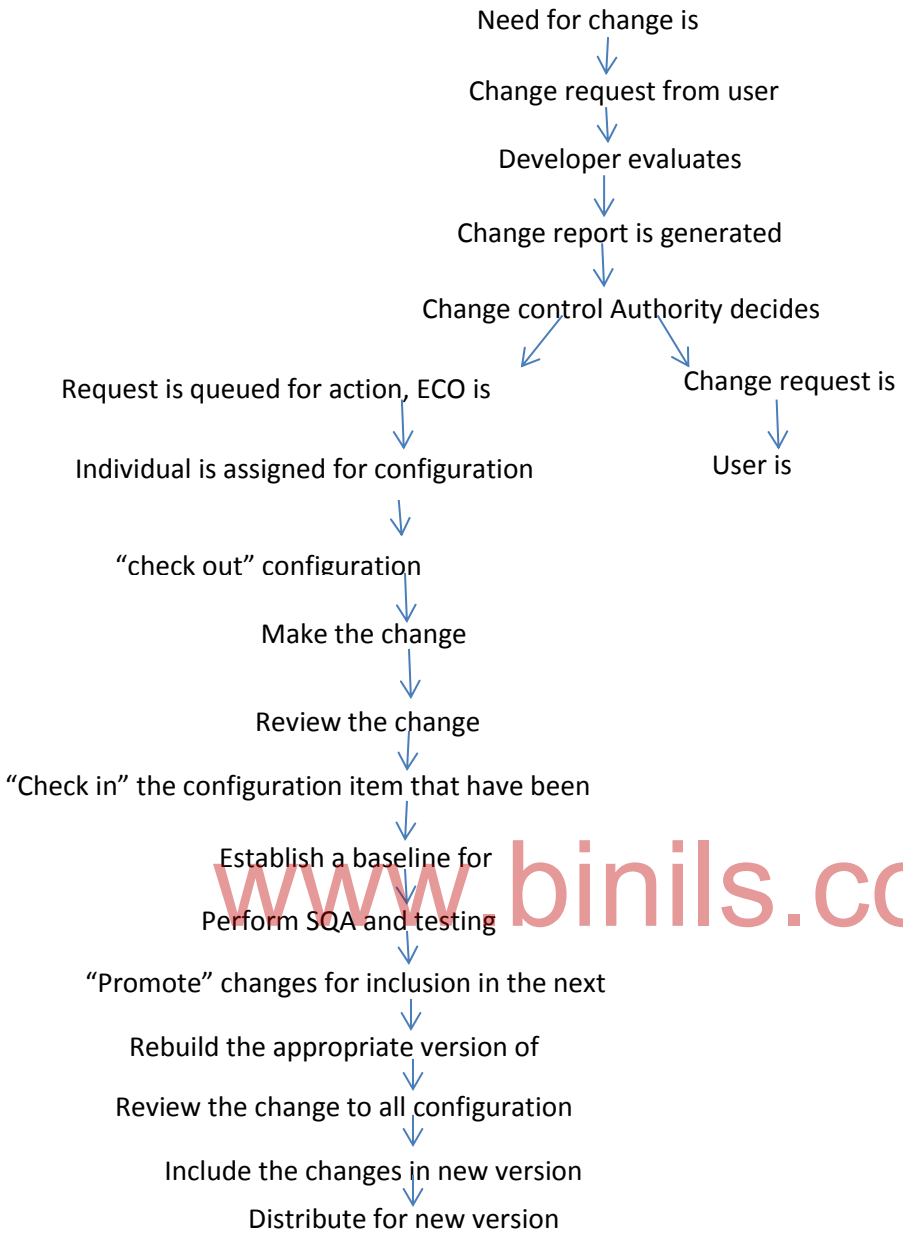


Fig.No 3.3. Change control Process

3.1.4 SOFTWARE VERSION CONTROL

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. During the process of software evolution, many objects are produced, for example, files, electronic documents, paper documents, source code, executable code and bitmap graphics. A version control tool is the first stage towards being able to manage multiple versions. Once it is in place, a detailed record of every version of the software must be kept. This includes the following components:

- Name of each source code component, including the variations and revisions
- The versions of the various compilers and linkers used

- The name of the software staff who constructed the component
- The date and the time at which it was constructed.

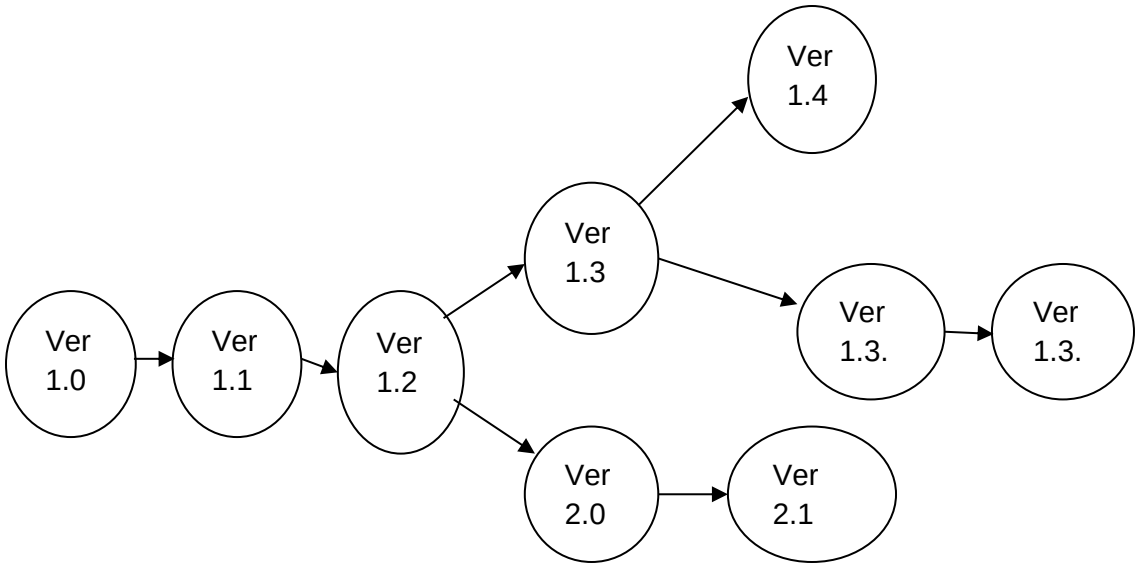


Fig.No 3.4 An Evolutionary Graph for a Different Version of an Item

The above evolutionary graph in figure 3.4 shows the evolution of a configuration item during the development life cycle. The initial version of the item is given version number Ver 1.0. Subsequent changes to the item which could be mostly fixing bugs or adding minor functionality is given as Ver 1.1 and Ver 1.2. After that, a major modification to Ver 1.2 is given a number Ver 2.0 at the same time; a parallel version of the same item without the major modification is maintained and given a version number 1.3.

3.1.5 NEED FOR MAINTENANCE

Software Maintenance process is expensive and risky and is very challenging. Software maintenance is essential for the following reasons

- Changes in user requirement with time
- Program / System Problem
- Changing hardware / Software environment
- To improve system efficiency
- To modify the components
- To test the resulting product to verify the correctness of changes
- To eliminate any unwanted side effects resulting from modification
- To fine-tune the software
- To optimize the code to run faster
- To review the standards and efficiency
- To make the code easier to understand and work with
- To eliminate any deviations from specifications.

The actions carried out during the software maintenance are:

- Correct errors
- Correct requirements and design flaws
- Improve the design
- Make enhancements
- Interface with other systems
- Convert to use other hardware, can be used
- Retire systems
- Maintaining control over the system's day-to-day functions
- Maintaining control over system modification
- Perfecting existing acceptable functions
- Preventing system performance from degrading to unacceptable levels

3.1.6 CATEGORIES OF MAINTENANCE

Maintenance may be **classified** into the four categories as follows:

- **Corrective -** reactive modification to correct discovered problems.
- **Adaptive –** modification to keep it usable in a changed or changing environment.
- **Perfective –** improve performance or maintainability.
- **Preventive –** modification to detect and correct latent faults.

- **Corrective Maintenance** - This includes modifications and updations done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.
- **Adaptive Maintenance** - This includes modifications and updations applied to keep the software product up-to date and tuned to the ever-changing world of technology and business environment.
- **Perfective Maintenance** - This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
- **Preventive Maintenance** - This includes modifications and updations to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future

As maintenance is very costly and very essential, efforts have been done to reduce its costs. One way to reduce the costs is through maintenance management and software modification audit. Software modification consists of programs rewriting, system level up gradation.

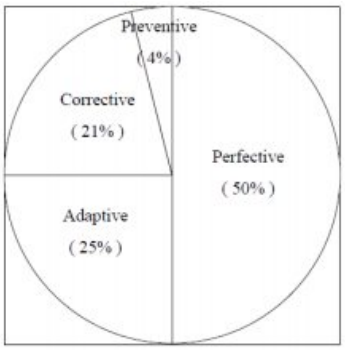


Fig.No 3.5 Percent of Total Maintenance

3.1.7 MAINTENANCE COST

Software maintenance takes up a higher proportion of IT budgets than new development (roughly two-thirds maintenance, one-third development). More of the maintenance budget is spent on implementing new requirements than on fixing bugs. The maintenance costs will obviously vary from one organization. Evolving the system to cope with new environments and new or changed requirements consumes most maintenance effort.

In the 1970s, cost of software system's budget was spent on development. The ratio of development money to maintenance at 40 to 60 percent of the full life-cycle of a system (i.e., from development through maintenance to eventual retirement or replacement). However, this cost may vary widely from one application domain to another.

It is advisable to invest more effort in early phases of software life cycle to reduce the maintenance costs. The defect repair ratio increases heavily from analysis phase to implementation phase and given table.

Defect repair ratio

Phase	Ratio
Analysis	1
Design	10
Implementation	100

3.1.8 FACTORS AFFECTING THE EFFORT

There are many other factors that contribute to the effort needed to maintain a system. These factors can include the following:

- Application Type
- System novelty
- Turnover and maintenance staff availability
- System life span
- Dependence on a changing environment
- Hardware characteristics
- Design quality
- Code quality
- Documentation quality
- Testing quality

MODELING MAINTENANCE EFFORT

1. Belady and Lehman Model.

This model indicates that the effort and cost can increase exponentially if poor software development approach is used and the person or group that used the approach is no longer available to perform maintenance. Belady and Lehman capture these effects in an equation:

$$M = P + Ke^{(c-d)}$$

M is the total maintenance effort expanded for a system, and P represents wholly productive efforts: analysis, evaluation, design, coding, and testing. c is complexity caused by the lack of structured design and documentation; it is reduced by d, the degree to which the maintenance team is familiar with software. Finally, K is a constant determined by comparing this model with the effort relationships on actual projects; it is called an *empirical constant* because its value depends on the environment.

The Belady – Lehman equation expresses a very important relationship among the factors determining maintenance effort. In this relation, the value of 'c' is increased if the software system is developed without use of a software engineering process. If the software is maintained without an understanding of the structure, function and purpose of the software, then the value of'd' will be low.

The result is that the cost for maintenance increases exponentially. Thus, to economize on maintenance, the best approach is to build the system using good software engineering practices and to give the maintainers time to become familiar with the software.

Example:-

The development effort for a software project is 400 person – months. The empirically determined constant (K) is 0.3. The complexity of the code is quite high and is equal to 8. Calculate the total effort expended (M) if,

(i) Maintenance team has good level of understanding of the project (d=0.9)
(ii) Maintenance team has poor understanding of project (d=0.1)

Solution

Development effort (P) = 400PM , K = 0.3 , C = 8

(i) Maintenance team has good level of understanding (d=0.9)

$M = P + Ke^{(c-d)} = 400 + 0.3\ e^{(8-0.9)} = 400 + 363.59 = 763.59$ PM

(ii) Maintenance team has poor level of understanding (d=0.1)

$M = P + Ke^{(c-d)} = 400 + 0.3e^{(8-0.1)} = 500+809.18 = 1209.18$PM

Hence, it is clear that effort increases exponentially, if poor software engineering approaches are used and understandability of the project is poor.

2. Boehm Model.

Boehm proposed a formula for estimating maintenance costs as part of his COCOMO Model. Using data gathered from several projects, this formula was established in terms of effort. Boehm used a quantity called Annual Change Traffic (ACT), which is defined as:

The fraction of a software product's source instructions which undergo change during a year either through addition, deletion or modification.

The ACT is clearly related to the number of change requests.

$$ACT = \frac{KLOC_{total}}{KLOC_{added} + KLOC_{deleted}}$$

where, $KLOC_{added}$ is the total kilo lines of source code added during maintenance. $KLOC_{deleted}$ during maintenance. Thus, the code that is changed should be counted in both the code added and the code deleted.

The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost.

Maintenance cost = ACT x development cost

Most maintenance cost estimation models, however, give only approximate results because they do not take into account several factors such as the experience level of engineers, and familiarity of engineers with the product, hardware requirements, software complexity, etc.,

Example : Annual change traffic (ACT) for a software system is 15% per year. The development effort is 500 PMs. Compute an estimate for Annual Maintenance Effort (AME). If lifetime of the project is 10 years, what is the total effort of the project?

Solution

The development effort = 500 PM

Annual Change Traffic (ACT) = 15%

Total duration for which effort is to be calculated = 10 years.

The maintenance effort is a fraction of development effort and is assumed to be constant.

AME = ACT x SDE 0.15 x 500 = 75 PM

Maintenance effort for 10 years = 10 x 75 =750PM

Total effort = 600 + 750 = = 1550 PM

3.2 RISK MANAGEMENT

Risk management is concerned with assessing the possible losses that might ensue from attacks on assets in the system, and balancing these losses against the costs of security procedures that

may reduce these losses. It is up to senior management to decide whether or not to accept the cost of security or the exposure that results from a lack of security procedures.

3.2.1 DEFINITION – SOFTWARE RISKS

A risk may be defined as a potential problem. It may or may not happen. But, it should always be assumed that it may happen and necessary steps are to be taken.

Risks can arise from various factors like improper technical knowledge or lack of communication between team members, lack of knowledge about software products, market status, hardware resources, competing software companies, etc.

3.2.2 BASICS FOR DIFFERENT TYPES OF SOFTWARE RISKS

- **Skills or Knowledge:** The persons involved in activities of problem analysis, design, coding and testing have to be fully aware of the activities and various techniques at each phase of the software development cycle. In case, they have partial knowledge or lacks adequate skill, the products may face many risks at the current stage or at later stages.
- **Interface modules:** Complete software contains various modules and each module sends and receives information to other modules and their concerned data types have to match.
- **Poor knowledge of tools:** If the team or individual members have poor knowledge of tools used in the software product, then the final product will have many risks, since it is not thoroughly tested.

- **Programming skills:** The code developed must be efficient. It must occupy less memory space and less CPU cycles to compute given tasks. The software should be able to implement various object oriented technologies and to catch exceptions in the case of errors.
- **Management Issues:** The management of the organization should give proper training to the project staff, arrange some recreation activities, give bonus and promotions and interact with all members of the project and try to solve their necessities at the best. Team members and the project manager should have healthy coordination.
- **Updates in the hardware resources:** The team should be aware of the latest updates in the hardware resources,peripherals, etc. In case the developer makes a product, and later in the market, a new product is released, the product should support minimum features. Otherwise, it is considered a risk, and may lead to the failure of the project.
- **Extra support:** The software should be able to support a set of a few extra features in the vicinity of the product to be developed.
- **Customer Risks:** Customer should have proper knowledge of the product, and should not be in a hurry to get the work done. He should take care that all the features are implemented and tested. He should take the help of a few external personnel to test the product and should arrange for demonstrations with a set of technical and managerial persons from his office.

- **External Risks:** The software should have backup in CD, tapes, etc., fully encrypted with full license facilities. The software can be stored at various important locations to avoid any external calamities like floods, earthquakes, etc. Encryption is maintained such that no external persons from the team can tap the source code.
- **Commercial Risks:** The organization should be aware of various competing vendors in the market and various risks involved if their product is not delivered on time.

3.2.3 MONITORING OF RISKS

Various risks are identified and it is maintained in the risk table monitor. A risk table monitor contains attributes like risk name, module name, team members involved, lines of code, codes affecting this risk, hardware resources, etc., If the project is continued further to 2-3 weeks, and then further the risk table is also updated. It is seen whether there is a ripple effect in the table, due to the continuity of old risks. Risk monitors can change the ordering of risks to make the table easy for computation. The following table shows a risk table monitor. It shows the risk that is being monitored.

RISK TABLE MONITOR

Sl. No.	Risk Name	Week 1	Week 2	Remarks
1.	Module Average()	Line 15, 18, 20	Line 15,25	Priority 3
2.	More memory and peripherals	Module f2(), f5() affected	Module f3() affected	Priority 1
……	……	……	……	…….

The above risk table monitor has a risk in module Average() where there is a risk in line 15, 18 and 20 in week 1. In week 2, risks are present in lines 15 and 25. Risks are reduced in week 2. The priority 3 is set. Similarly, in the second row, risk is due to more memory and peripherals, affecting module f2(), f5() in week-1. After some modifications in week 2, module f3() is affected and the priority is set to 1.

3.2.4 RISK MANAGEMENT

Risk management is a set of practices and support tools to identify, analyze and treat risks explicitly and making the software product error free. Treating risks means understanding it better, avoiding or reducing it. Risk management tries to reduce the probability of risk occurring and the impact caused by risks.

A priority is given to risk and the highest priority risk is handled first. Various factors of the risk are who are the involved team members, what hardware and software items are needed, where, when and why are resolved during risk management. The risk manager does scheduling of risks. Risk management can be further categorized as follows:

1. **Risk Avoidance**
   a. Risk anticipation
   b. Risk tools
2. **Risk Detection**
   a. Risk analysis
   b. Risk category
   c. Risk prioritization
3. **Risk Control**
   a. Risk pending
   b. Risk resolution
   c. Risk not solvable
4. **Risk Recovery**
   a. Full
   b. Partial
   c. Extra / alternate features

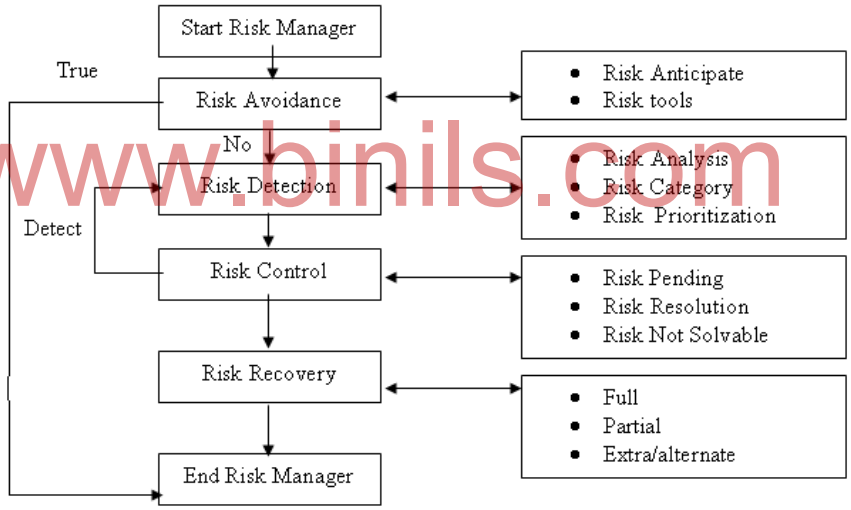


Fig.No 3.6 Risk Manager Tool

Fig. 3.6. Shows the Risk Manager Tool. From the figure, it is clear that the first phase is to avoid risk by anticipating and using tools from previous project history. In case there is no risk, risk manager halts. In case there is risk, detection is done using various risk analysis techniques and further prioritizing risks.

In the next phase, risk is controlled by pending risks, resolving risk and in the worst case, (if risk is not solved) lowering the priority. Lastly, risk recovery is done fully, partially or an alternate solution is found.

3.2.5 RISK AVOIDANCE

Risk Anticipation: Various risk anticipation rules are listed according to standards from previous projects experience, and also as mentioned by the project manager.

Risk tools: Risk tools are used to test whether the software is risk free. The tools have built-in data base of available risk areas and can be updated depending upon the type of project.

3.2.6 RISK DETECTION

The risk detection algorithm detects a risk and it can be categorically stated as:

Risk Analysis: In this phase, the risk is analyzed with various hardware and software parameters as probabilistic occurrence (pr), weight factor (wf) (hardware resources, lined of code, persons), risk exposure (pr * wf). Table shows a risk analysis table.

RISK ANALAYSIS TABLE

Sl. No.	Risk Name	Probability of Occurrence (pr)	Weight factor (wf)	Risk exposure (pr * wf)
1.	Stack Overflow	4	15	60
2.	No Password forgot option	8	150	120
…..	…..	…..	…..	…..

Maximum value of risk exposure indicates that the problem has to solved as soon as possible and be given high priority. A risk analysis table is maintained as shown above.

Risk Category*:* Risk identification can be from various factors like persons involved in the team, management issues, customer specification and feed back, environment, commercial, technology, etc. Once proper category is identified, priority is given depending upon the urgency of the product.

Risk Prioritization: Depending upon the entries of the risk analysis table, the maximum risk exposure is given high priority and has to be solved first.

3.2.7 RISK CONTROL

Once the prioritization is over, the next step is to control various risks as follows:

- **Risk Pending:** According to priority, low priority risks are pushed at the end of the queue with a view of various resources (hardware, man power, software) and in case it takes more time their priority is made higher.
- **Risk Resolution:** Risk manager makes a strong resolve how to solve the risk.

- **Risk elimination:** This action leads to serious error in software.
- **Risk transfer:** If the risk is transferred to some part of the module, then risk analysis table entries get modified. Thereby, again risk manager will control high priority risk.
- **Disclosures:** Announce the risk of less priority to the customer or display message box as a warning. And thereby the risk is left out to the user, such that he should take proper steps during data entry, etc.,
- **Risk not solvable:** If a risk takes more time and more resources, then it is dealt in its totally in the business aspect of the organization and thereby it is notified to the customer, and the team member proposes alternate solution. There is a sight variation in the customer specification after consultation.

3.2.8 RISK RECOVERY

Full: The risk analysis table is scanned and if the risk is fully solved, then corresponding entry is deleted from the table.

Partial: The risk analysis table is scanned and due to partially solved risks, the entries in the table are updated and thereby priorities are also update.

Extra/alternate features: Sometimes it is difficult to remove some risks, and in that case, we can add a few extra features, which solve the problem. Therefore, a bit of coding is done to get away from the risk. This is later documented or notified to the customer.

3.2.9 SOURCES OF RISKS

There are two major sources of risk, which are as follows:

1. **Generic Risks.** Generic risks are common to all software projects. *Generic risks* are a potential threat to every software project. For example, Requirements misunderstanding, allowing insufficient time for testing is losing key personnel etc.
2. **Project-Specific Risks.** A vendor may be promising to deliver particular software by a particular date, but he is unable to do it.

3.2.10 TYPES OF RISKS

Risks are of three types : They are (i) Technical Risks (ii) Business Risk and (iii) Project Risks

TECHNICAL RISKS

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification

ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than we thought it would be.

BUSINESS RISKS

Business risks are risks, which affect the organization developing or procuring the software. *Business risks* threaten the viability of the software to be built. For example, technology change and product competition. The top five business risks are:

- Building on excellent product or system that no one really wants(market risks)
- Building a product that no longer fits into the overall business strategy for the company(strategic risk)
- Building a product that the sales force doesn't understand how to sell.
- Losing the support of senior management due to a change in focus or a change in people (management risk.
- Losing budgetary or personnel commitment (budget risks).

PROJECT RISKS.

Project risks are risks, which affect the project schedule or resources. *Project risks* threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. This risk occurs due to conditions and constraints about resources, relationship with vendors and contractors, unreliable vendors and lack of organizational support. Funding is the significant project risk the management has to face. It occurs due to initial budgeting constraints and unreliable customer payments. For example Staff turnover, management change, hardware uninvertibility.

3.3 PROJECT SCHEDULING

3.3.1 INTRODUCTION

The process of building a schedule for any project helps really to understand as how it is done. The basic idea to get across is to divide the project into well-defined modules, determine the independence among the modules, determine the time duration for each module and assign the modules to the project team members. Each module must have a well-defined outcomes and be associated a meaningful project milestone. Scheduling has the ultimate goal of deciding on how to distribute effort over time

.

3.3.2 FACTORS AFFECTING THE TASK SET FOR THE PROJECT

- **Technical staff expertise:** All staff members should have sufficient technical expertise for timely implementation of the project. Meetings have to be conducted, weekly and status reports are to be generated.

- **Customer Satisfaction:** Customer has to be given timely information regarding the status of the project. If not, there might be a communication gap between the customer and the organization.
- **Technology update:** Latest tools and existing tested modules have to be used for fast and efficient implementation of the project.
- **Full or partial implementation of the project:** In case, the project is very large and to meet the market requirements, the organization has to satisfy the customer with at least a few modules. The remaining modules can be delivered at a later stage.
- **Time allocation:** The project has to be divided into various phases and time for each phase has to be given in terms of person-months, module-months etc.,
- **Module binding:** Module has to bind to various technical staff for design, implementation and testing phases. Their necessary inter-dependencies have to be mentioned in a flow chart
- **Milestones:** The outcome for each phase has to be mentioned in terms of quality, specifications implemented, limitations of the module and latest updates that can be implemented (according to the market strategy).
- **Validation and Verification:** The number of modules verified according to customer specification and the number of modules validated according to customer's expectations are to be specified

3.3.3 SCHEDULING METHODS

Scheduling of a software project can be correlated to prioritizing various tasks (jobs) with respect to their cost, time and duration. Scheduling can be done with resource constraint or time constraint in mind. Depending upon the project, scheduling methods can be static or dynamic in implementation.

SCHEDULING TECHNIQUES : The various types of scheduling techniques in software engineering are: (i) Work Breakdown structure (ii) Gant Chart (iii) PERT chart

3.3.4 WORK BREAKDOWN STRUCTURE

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

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

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

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

An example for work breakdown structure is shown in fig.3.7.

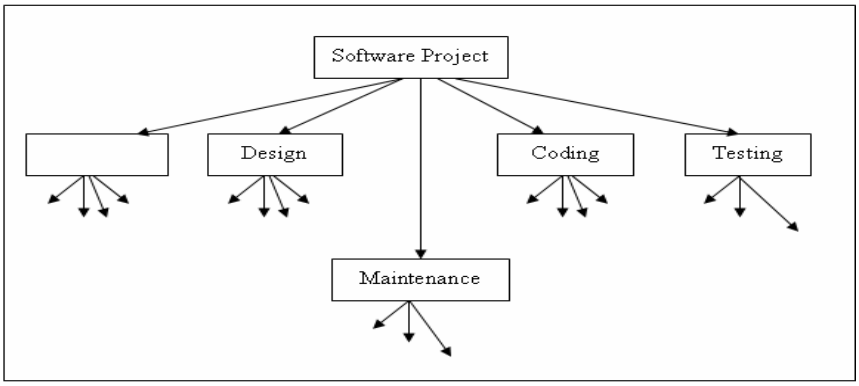


Fig.No 3.7 An example Work Breakdown Structure

The project is split into requirement and analysis, design, coding, testing and maintenance phase. Further, requirement and analysis is divided into R1, R2…..Rn; design is divided into D1,D2…..Dn; coding is divided into C1,C2…..Cn; testing is divided into T1,T2…..Tn; and maintenance is divided into M1,M2…..Mn. If the project is complex, then further sub division is carried out. Upon the completion of each stage, integration is done.

3.5 FLOW GRAPH

Various modules are represented as nodes with edges connecting nodes. Dependency between nodes is shown by flow of data between nodes. Nodes indicate milestones and deliverables with the corresponding module implemented. Cycles are not allowed in the graph. Start and end nodes indicate the source and terminating nodes of the flow.

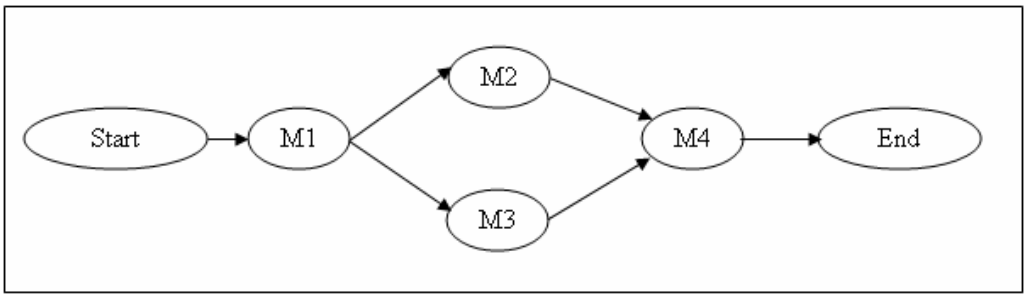


Fig.No 3.8 Flow graph

M1 is the starting module and the data flows to M2 and M3. The combined data from M2 and M3 flow to M4 and finally the project terminates. In certain projects, time schedule is also associated with each module. The arrows indicate the flow of information between modules.

3.3.6 GANT CHART

Gant chart is a graphical representation of the project. It is used in project scheduling to depict the activities of the project. This chart shows the start and end dates of each activity in the project. In addition, it shows week, month, or quarter required to complete each activity. Due to this fact, Gantt chart is also known as timeline chart. It shows information about activities in the form of horizontal bars,. Generally, a Gantt chart is prepared on a graph paper. In case a Gantt chart is big and complex, it is prepared using applications such as Microsoft Excel.

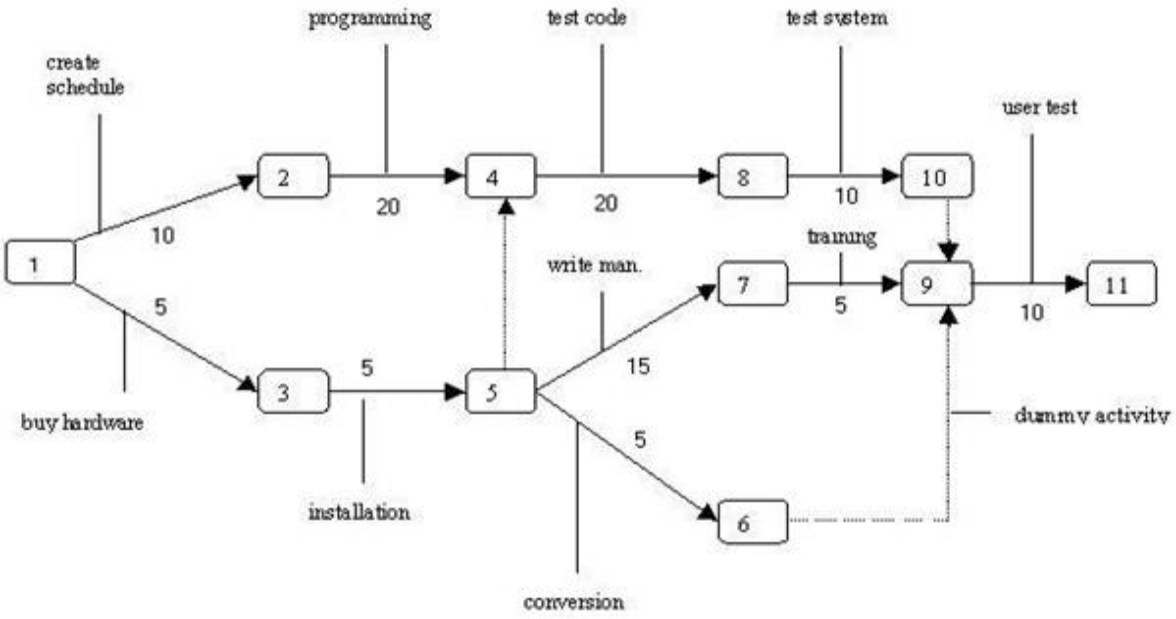
A Gantt chart helps the project manager by providing a graphical illustration of the project schedule. The advantages of using Gantt chart are:

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

3.3.7 PERT

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

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



* Numbered rectangles are nodes and represent events or milestones.
* Directional arrows represent dependent tasks that must be completed sequentially.
* Diverging arrow directions (e.g. 1-2 & 1-3) indicate possibly concurrent tasks
* Dotted lines indicate dependent tasks that do not require resources.

Fig.No 3.9 PERT Chart

SUMMARY

- A software product must change continually or become progressively less useful.
- The structure of a program tends to degrade as more and more maintenance is carried out on it.
- Over a program's lifetime, its rate of development is approximately constant.
- Configuration management is carried out with the help of three principle activities: Configuration Identification, Configuration Control, Configuration Accounting
- the object associated with a software development is classified into three main categories: controlled, pre-controlled, and uncontrolled objects
- Controlled objects are those, which are already put under configuration control and some procedures has to be followed to change them
- Pre controlled objects are not yet under configuration control, but will eventually be under configuration control.
- Configuration control is the process of managing changes to controlled objects. The configuration control system prevents unauthorized changes to any controlled object.
- Once the changes in baselines occur, some mechanisms must be used to record how the system evolves and what its current state is. This task is accomplished by status accounting.
- A software configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review.
- Change control combines human procedures and automated tools to provide mechanism for the control of change.
- Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process.
- Software maintenance is the activity associated with keeping operational computer system continuously in tune with the requirements of users and data processing operation.
- Maintenance may be classified into the four categories: Corrective, Adaptive, Perfective, Preventive
- Belady and Lehman Model indicates that the effort and cost can increase exponentially if poor software development approach is used and the person or group that used the approach is no longer available to perform maintenance.
- A risk may be defined as a potential problem. It may or may not happen. But, it should always be assumed that it may happen and necessary steps are to be taken.
- A risk table monitor contains attributes like risk name, module name, team members involved, lines of code, codes affecting this risk, hardware resources, etc.,
- Risk management makes the software product error free. Firstly, risk management takes care that the risk is avoided, and if it not avoidable, then the risk is detected, controlled and finally recovered.

- Generic risks are common to all software projects. Generic risks are a potential threat to every software project.
- Project-Specific Risks**.** A vendor may be promising to deliver particular software by a particular date, but he is unable to do it
- Risks are of three types: They are (i) Technical Risks (ii) Business Risk and (iii) Project Risks
- Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible
- Business risks are risks, which affect the organization developing or procuring the software. Business risks threaten the viability of the software to be built.
- Project risks are risks, which affect the project schedule or resources. Project risks threaten the project plan
- Scheduling of a software project can be correlated to prioritizing various tasks (jobs) with respect to their cost, time and duration. Scheduling can be done with resource constraint or time constraint in mind.
- The various types of scheduling techniques in software engineering are: (i) Work Breakdown structure (ii) Gant Chart (iii) PERT chart
- Work Breakdown Structure is the process of dividing the project into tasks and logically ordering them in a sequence. It provides a framework for keeping track of the progress of the project and for determining the cost planned for these tasks.
- Gant chart is a graphical representation of the project. It is used in project scheduling to depict the activities of the project. This chart shows the start and end dates of each activity in the project.
- The program Evaluation and Review Technique (PERT) chart is used to schedule, organize and coordinate tasks within the project. The objective of PERT chart is to determine the critical path, which comprises critical activities that should be completed on schedule.

REVIEW QUESTIONS

PART – A (2 Marks)

1. State Lehman's first and second law.
2. What is configuration control?
3. What are the two important elements of change control?
4. What are the commercial tools used in version control?
5. What are the types of maintenance?
6. What are the models used in maintenance effort?
7. What is risk? Define the term software risk.

8. List any four types for software risks.
9. What are the risk controls available?
10. List the categories of risk management.
11. What is meant by project scheduling?
12. What are the scheduling techniques available in software engineering?
13. What is i) Gantt chart ii) PERT chart?
14. What is risk table monitor?
15. What do you mean by project risk?

PART – B (3 Marks)

1. What are the three steps involved in software configuration management activities? Define them.
2. How we can classify the object associated with software development?
3. State the components of version control.
4. What is adaptive maintenance?
5. What are the factors that contribute to the effort needed to maintain a system?
6. Draw Risk manager tool.
7. What are the three types of risk?
8. What do you mean by risk prioritization?
9. What is risk recovery? Explain.
10. List the types of business risk.
11. What are the sources of risk?
12. Give an example for Risk analysis table.
13. Give an example of Risk table monitor.

PART – C (5 Marks /10 Marks)

1. What do you mean by configuration identification? Explain.
2. What is change control? Explain
3. What is version control? Explain.
4. State any five reasons for software maintenance?
5. What are the actions carried out during software maintenance?
6. What is monitoring of risks? Explain
7. What are the different types of risks? Explain
8. Explain about flow graph.

9. Give an example for PERT chart.
10. Explain the following SCM activities (a) Configuration control and (b) Configuration accounting.
11. Briefly explain change control process.
12. Explain briefly about (i) versions and releases and (ii) version and release management.
13. Explain briefly about different types of Software Maintenance.
14. Explain briefly about any one of the maintenance effort models with example.
15. Explain the basics for different types of software risks.
16. Explain briefly about (i) Risk Management and (ii) Risk Avoidance
17. Explain Boehm model and how it helps in software maintenance.
18. Explain the various factors affecting the task set for the project
19. Briefly explain (i) work break down structure (ii) Gant Chart

UNIT – IV	

SOFTWARE TESTING

OBJECTIVES

At the end of the unit, the students will be able to

- State the purpose of software Testing.
- Define Testing principles and objectives.
- Define basic terms used in testing.
- Differentiate white box testing from black box testing.
- Explain different types of white box testing and black box testing.
- Discuss different levels of testing.
- Know about Software testing strategies
- Understand the life cycle of a debugging task
- Need for software testing tool.
- List down the different categories of software testing tools.
- Define code of ethics for software professionals.

4.1 SOFTWARE TESTING

4.1.1 INTRODUCTION TO TESTING

Software testing gives an important set of methods that can be used to evaluate and assure that a program or system meets its non-functional requirements. The aim of most testing methods is to systematically and actively locate faults in the program and repair them.

4.1.2 TESTING PRINCIPLES

There are many principles that guide software testing.

- ✓ All tests should be traceable to customer requirements.
- ✓ Tests should be planned long before testing begins.
- ✓ The Pareto principle applies to software testing.
- ✓ Testing should begin “in the small” and progress toward testing “in the large”.
- ✓ Exhaustive testing is not possible.
- ✓ To be most effective, testing should be conducted by an independent third party.

4.1.3 TESTING OBJECTIVES

Testing objectives are:

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

4.1.4 TESTING ORACLES

To test any program, a description of its expected behavior is need for and a method of determining whether the observed behavior conforms to the expected behavior. For this a test oracle is needed.

A test oracle is a mechanism, which can be used to check the correctness of the output of the program for the test cases. Conceptually, testing is a process in which the test cases are given to the test oracle and the program under testing. The output of the two is then compared to determine if the program behaved correctly for the test cases,

.

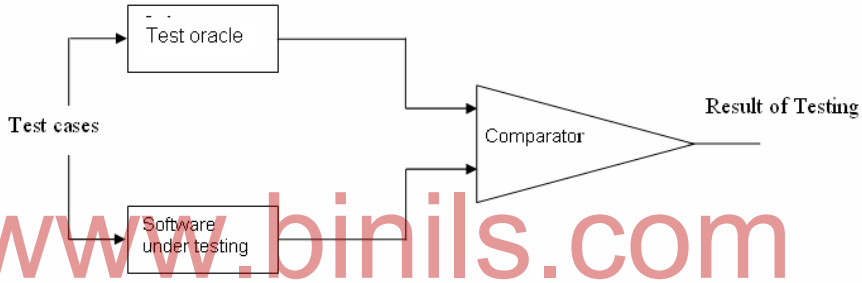


Fig 4.1 Test Oracle

Test oracles are human beings, so they may make mistakes when there is a discrepancy between the oracles and the results of a program. First verify the result produced by the oracle, before declaring that there is a fault in the program. Hence the testing is so cumbersome and expensive.

The human oracles generally use the specification of the program to decide what the "correct" behavior of the program should be. To help the oracle to determine the correct behavior, it is important that the system be unambiguously specified and the specification itself is error free.

4.1.5 BASIC TERMS USED IN TESTING

ERROR : Error is a discrepancy between actual value of the output given by the software and the specified correct value of the output for that given input. Error can be classified into two categories 1 .**Syntax Error:** A syntax error is program statements that violate one or more rules of the language in which it is written. 2. **Logic Error:** A logic error deals with incorrect data fields, out of range terms and invalid combinations.

FAULT : Fault is a condition that causes a system to fail in performing its required function. A fault is the basic reason for software malfunction. It is also commonly called bug. Even though correct input is given to the system, when it fails then the system has a fault or bug, and needs repair.

FAILURE : Failure is the inability of the software to perform a required function to its specification. In other words, when software goes ahead in processing without showing errors or fault even though certain input and process specification are violated, then it is called software failure.

4.1.5.1 TEST CASES

Testing comes down to selecting and executing test cases. A test case for a specific component consists of three essential pieces of information:

- A Set of test inputs
- The expected results when the inputs are executed; and
- The execution conditions or environments in which the inputs are to be executed .

4.1.6 BLACK BOX AND WHITE BOX TESTING

Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester . White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.

BASIS	WHITE-BOX TESTING	BLACK-BOX TESTING
Purpose	• To test the internal structure of software • Tests the software but does not ensure the complete implementation of all the specifications mentioned in user requirements. • Addresses flow and control structure of a program	• To test the functionality of software • Concerned with testing the specifications and does not ensure that all the components of software that are implemented are tested • Addresses validity, behaviour and performance of software
Stage	• Performed in the early stages of testing	• Performed in the later stages of testing
Requirement	• Knowledge of the internal structure of a program is required for generating test cases	• No knowledge of the internal structure of a program is required to generate test cases
Test Cases	• Test cases are generated based on the internal structure or code of the module to be tested	• Internal structure of modules or programs is not considered for selecting test cases
Example	• The inner software present inside the calculator(Which is known to the developer only) is checked by giving inputs to the code	• In this testing, it is checked whether the calculator is working properly by giving inputs by pressing the buttons in the calculator

ADVANTAGES AND DISADVANTAGES OF BLACK BOX TESTING

- The main advantage of black box, test case selection can be done before the design or coding of a program. Black box test cases can also help to get the design and coding correct with respect to the specification. Black box testing methods are good at testing for missing functions or program behavior that deviates from the specification.
- The main disadvantage of black box testing is that black box test cases cannot detect additional functions or features that have been added to the code. This is especially important for systems that need to be safe.

ADVANTAGES AND DISADVANTAGES OF WHITE BOX TESTING

- The main advantage of white box testing is that it tests the internal details of the code and tries to check all the paths that a program can execute to determine if a problem occurs. White box testing can check additional functions or code that has been implemented, but not specified.
- The main disadvantage of white box testing, test cases are selected after designing, coding and testing.

4.1.7 METHODS FOR BLACK BOX TESTING STRATEGIES

In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches available to design black-box test cases:

- Equivalence class partitioning.
- Boundary value analysis

EQUIVALENCE CLASS PARTITIONING

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set. These classes are called equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.

Equivalence classes for a program can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e. [1, 10]), then the invalid equivalence classes are [-∞, 0], [11,+∞] or valid equivalence class[1,10].

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are {A, B, C}, then the invalid equivalence class is U-{A, B, C}, where U is the universe of possible input values.

EXAMPLE 1 : For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence class test suite.

There are three equivalence classes- (i) the set of negative integers, (ii) the set of integers in the range of 0 and 5000, and (iii) the set of integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes. A possible test suite can be: {-5, 500, 6000}.

EXAMPLE 2: Design the equivalence class test cases for a program that reads two integer pairs (m1, c2) and (m2, c2) defining two straight lines of the form y=mx+c. The program computes the intersection point of the two straight lines and displays the point of intersection.

The equivalence classes are the following:

- Parallel lines (m1 = m2, c1≠c2)
- Intersecting lines (m1≠m2)
- Coincident lines (m1=m2, c1=c2)

Now, selecting one representative value from each equivalence class, we get the required equivalence class test suite {(2,2)(2,5),(5,5)(7,7),(10,10)(10,10)}.

EXAMPLE 3 : Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.

The equivalence classes have been shown in Figure 4.2.. The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: {abc, aba, abcdf}.

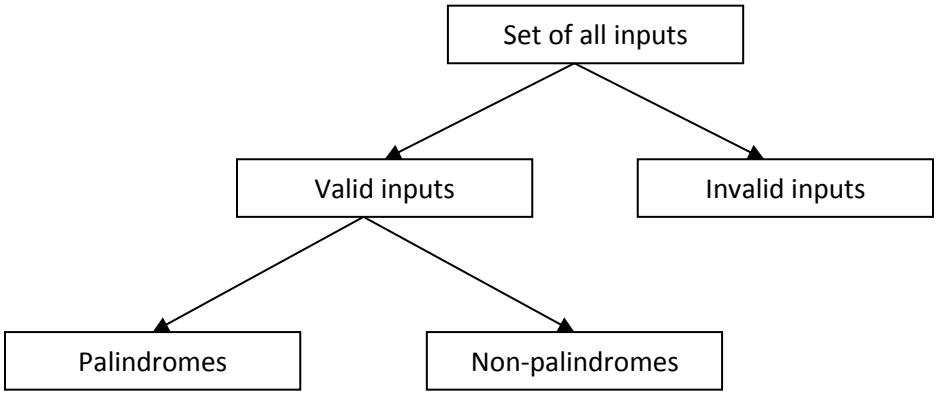


Fig No 4.2. Equivalence classes Example

BOUNDARY VALUE ANALYSIS

A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs. The reason behind programmers committing such errors might purely due to psychological factors. Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values. For those equivalence classes that are not a range of values (that is, consist of a discrete collection of values) no boundary value test cases can be defined. For an equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0, 1, 10, 11}.

Example 1: For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.

There are three equivalent classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. The boundary value-based test suite is: {0, -1, 5000, and 5001}.

Test case selection guidelines for boundary value analysis

The following set of guidelines is for the selection of test cases according to the principles of boundary value analysis. The guidelines do not constitute a firm set of rules for every case. Develop some judgment by applying these guidelines.

1. If an input condition specifies a range of values, then construct value test cases for the ends of a range, and invalid input test cases for input points just beyond the end of the range.
2. If an input condition specifies the number of values, construct test cases for the minimum and maximum values: and one beneath and beyond these values.

Differences between equivalence class partitioning and boundary value analysis techniques.

The difference between equivalence class partioning and boundary value analysis is presented as follows:

- Equivalence class partitioning: This method tests the validity of outputs by dividing the input domain into different classes of data (known as equivalence classes) using which test cases can be easily generated. Test cases are designed with the purpose of covering each partition at least

once. A test case that is able to detect every error in a specified partition is said to be an ideal test case. An equivalence class depicts valid or invalid states for the input condition. An input collection can be either a specific numeric value, a range of values, a Boolean condition, or a set of values.

- Boundary value analysis: In boundary value analysis (BVA), test cases are derived on the basis of values that lie on the edge of an equivalence partition. These values can be input or output values, either on the edge or within the permissible range from the edge of an equivalence partition.

BVA is more commonly used since it has been observed that most errors occur at the boundary of the input domain rather than in the middle of the input domain. Note that BVA complements the equivalence portioning method. The only difference is that in BVA, test cases are derived for both the input domain and the output domain, while in equivalence partitioning, test cases are derived only for the input domain.

4.1.8 METHODS FOR WHITE BOX TESTING STRATEGIES

In this approach, complete knowledge about the internal structure of the source code is required. For White-box testing strategies, the methods used are:

1. Coverage Based Testing 2. Cyclomatic Complexity 3. Mutation Testing

COVERAGE BASED TESTING

The aim of coverage based testing methods is to 'cover' the program with test cases that satisfy some fixed coverage criteria. Put another way, test cases should be chosen to exercise as much of the program as possible according to some criteria.

COVERAGE BASED TESTING CRITERIA: Coverage based testing works by choosing test cases according to well-defined 'coverage' criteria. The more common coverage criteria are the following.

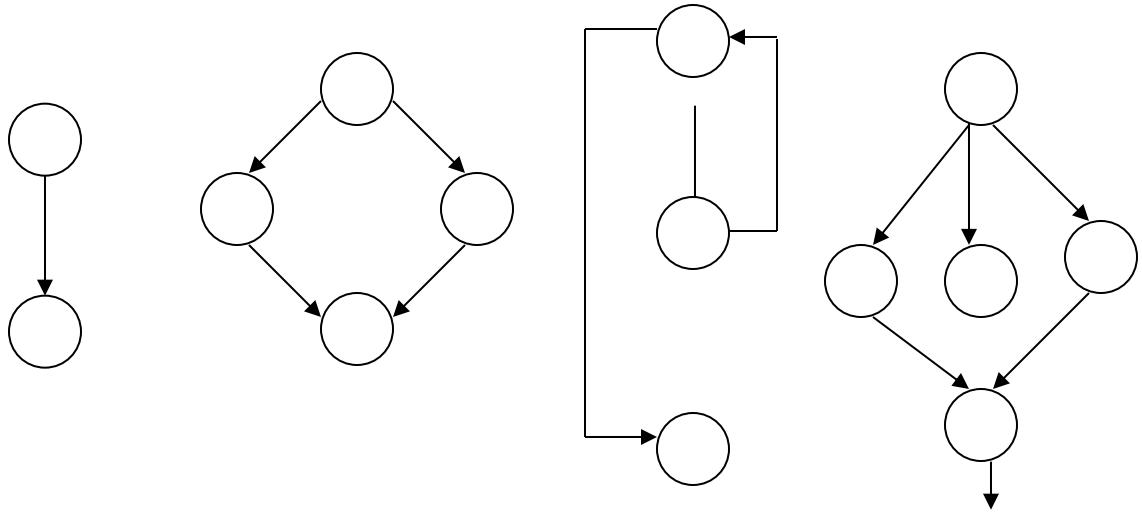
- **Statement Coverage or Node Coverage:** Every statement of the program should be exercised at least once.
- **Branch Coverage or Decision Coverage:** Every possible alternative in a branch or decision of the program should be exercised at least once. For if statements, this means that the branch must be made to take on the values *true* or *false*.
- **Decision/Condition Coverage:** Each condition in a branch is made to evaluate to both true and false.
- **Multiple condition coverage:** All possible combinations of condition outcomes within each branch should be exercised at least once.
- **Path coverage**: Every execution path of the program should be exercised at least once.

CONTROL FLOW GRAPH (CFG)

A control flow graph describes the sequence in which different instructions of a program get executed. It also describes how the flow of control passes through the program. In order to draw the control flow

graph of a program, first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph.

An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. The flow graph depicts logical control flow using the notation is shown in Figure 4.3.



Sequence If - else While loop case

Fig 4.3. The structured constructs in flow graph form

Example: Draw CFG for the program given below:

```
int sample (a,b)
int a,b;
{
while (a!=b)
if (a>b)
a= a-b;
else b=b-a;
}
return a;
}
```

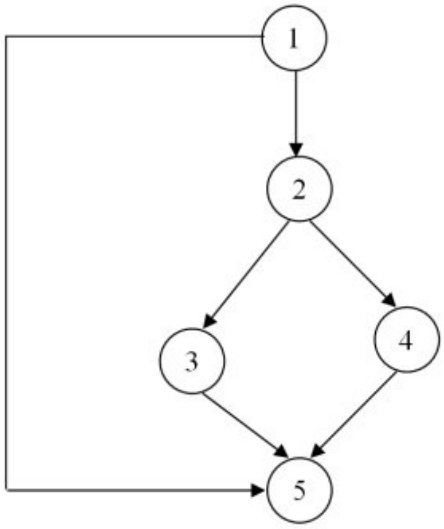


Fig No 4.4. CFG for program

In the above program, two control constructs are used; namely, while-loop and if-then-else. The complete CFG for the program is shown in the figure 4.4.

Cyclomatic Complexity: This technique is used to find the number of independent paths through a program. If CFG of a program is given, the Cyclomatic Complexity V(G) can be computed as: V(G) = E – N + 2, where N is the number of nodes of the CFG and E is the number of edges in the CFG. For the given example , the Cyclomatic Complexity = 6 – 5 + 2 = 3. Therefore, the cyclomatic complexity of the flow graph shown in Figure 4.4. is 3.

The following are the properties of Cyclomatic Complexity:

- V(G) is the maximum number of independent paths in graph G
- Inserting and deleting functional statements to G does not affect V(G)
- G has only one path if and only if V(G) = 1.

More important, the value for *V*(*G*) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

MUTATION TESTING

Mutation testing is a white-box testing method in which errors are “purposely” inserted into a program (under test) to verify whether the existing test case is able to detect the error. In this testing, mutants of the program are created by making some changes in the original program. The objective is to check whether each mutant produces an output that is different form the output produced by the original program.

In mutation testing, test cases that are able to “kill “all the mutants should be developed. This is accomplished by testing mutants with the developed set of test cases. There can be two possible outcomes when the test cases test the program – either the test case detects the failure or fails to detect faults. If faults are detected, then necessary measures are taken to correct them.

When no failures are detected, it implies that either the program is absolutely correct or the test case is inefficient to detect the failure. Therefore, it can be concluded that mutation testing is conducted to determine the effectiveness of a test case. That is, if a test case is able to detect these “small” faults, (minor changes) in a program, then it is likely that the same test case will be equally effective in finding real faults.

To perform mutation testing, several steps are followed:

1. Create mutants of a program
2. Check both the program and its mutants using test cases.
3. Find the mutants that are different from the main program. A mutant is said to be different from the main program if it produces an output which is different from the output produced by the main program.
4. Find mutants that equivalent to the main program. A mutant is said to be equivalent to the main program if it produces the same output as that of the main program

5. Compute the mutation score using the following formula:

   M = D / (N-E) ;

   where

   M = Mutation Score;

   N = Total Number of mutants of the program;

   D = Number of Mutants different from the main program;

   E = Total number of mutants that are equivalent to the main program.

6. Repeat steps 1 to 5 till the mutation score is “1”

However, mutation testing is very expensive to run on large programs. Thus certain tools are used to run mutation tests on large programs.

4.1.9 TEST ACTIVITIES

Requirement Analysis: Testing should begin in the requirements phase of the software development life cycle.

Design Analysis: During the design phase, testers work with developers in determining what aspects of a design are testable and under what parameters should the testers work.

Test Planning: Test Strategy, Test Plan(S).

Test Development: Test procedures, Test scanners, Test cases, Test Scripts to use in testing software

Text Execution: Testers execute the software based on the plans and tests and report any errors found to the development team.

Test Reporting: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.

Retesting the Defects: Defects are once again tested to find whether they got eliminated or not.

[[

4.1.10 TEST PLAN

Test plan is a document consisting of different test cases designed for different testing objects and different testing attributes. The plan puts the tests in logical and sequential order as per the strategy chosen, top-down or bottom-up. The test plan is a matrix of test and test cases listed in order of its execution.

The table below shows the matrix of test and test cases within the test.

Project Name …….. Project ID …….

Project Manager …….. Q.A. Manager …….

Test Plan

Test									
Test ID		Test	1	2	3	…	N	Planned Date	
ID	Tester	Name						Completed	Successful

Test ID, test name, test cases are designed well before the development phase and have been designed to those who conduct the tests. A test plan states the items to be tested, the level of testing, sequence of testing and how the test strategy will be applied to the testing of each item and describes the test environment.

4.2. LEVELS OF TESTING

Mainly, Software goes through three levels of testing:

• Unit testing	• Integration testing	• System testing.

4.2.1 UNIT TESTING

In unit testing individual components are tested to ensure that they operate correctly. On the smallest unit of software design, each component is tested independently without other system component. There are number of reasons for using in support of unit testing.

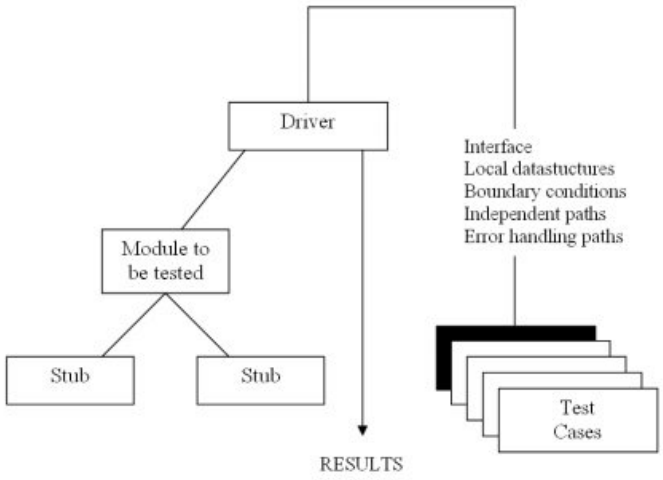
- Since the size of the single module is small , we can locate an error fairly easily.
- The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.
- Confusing interactions of multiple errors in widely different parts of the software are eliminated.

UNIT TEST PROCEDURE

In the most applications, the driver is nothing more than a “main program” that accepts the test case data, passes such data to the component (to be tested) and prints relevant results. Stubs serve to replace the module that is sub ordinate (called by) to the component to be tested. A stub uses subordinate modules interface, may do minimal data manipulation, prints the verification of entry and returns control to the models undergoing the testing.

Drivers and stubs represents overhead. i.e. both are software that must be written but that is not delivered with the final software product.

If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step.



Unit testing is simplified when a component with high cohesion. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

4.2.2 INTEGRATION TESTING

Integration testing is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and preceded system testing. Integration testing takes, modules that have been checked out by unit testing, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers its output. *The purpose of Integration testing is to verify functional, performance and reliability requirements placed on major design items.*

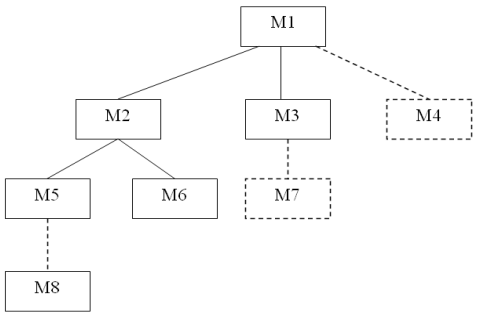
OBJECTIVES OF INTEGRATION TESTING

The primary objectives of integration testing is to test the module interfaces in order to ensure that there are no errors in the parameter passing, when one module invokes another module. During integration testing different modules of a system are integrated in a planned manner using an integration plan.

The integration plan specifies the step and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested.

TOP- DOWN INTEGRATION TESTING

Top-Down integration testing is an incremental approach to the construction of program structure. Modules are integrated by moving downward through the control hierarchy beginning with the main control module.



Top-Down Integration Testing

Depth-First Integration:

This would integrate all components on a major control path of a structure. For example, selecting the M1, M2 and M5 would be integrated first. Next M8 or M6 would be integrated. Then, the central and right-hand control paths are built.

Breadth-First Integration:

This incorporates all components directly subordinate at each level, moving across the structure horizontally. From figure, components M2, M3 and M4 would be integrated first. The next control level M5, M6, and so on, follows.

Bottom-up Integration Testing

In this testing, individual modules to be integrated are starting from the bottom and then moving upwards in the hierarchy. That is, bottom-up integration testing combines and tests the modules present at the lower levels proceeding towards the modules present at higher levels of the control hierarchy.

Some of the low-level modules present in software are integrated to form clusters (collection of modules).A test driver that coordinates the test case input and output is written and the clusters are tested. After the clusters have been tested, the test drivers are removed and the clusters are integrated ,moving upwards in the control hierarchy.

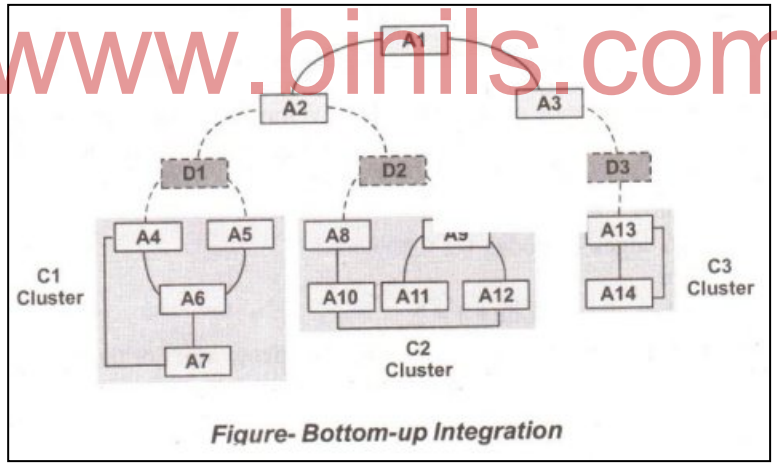


Figure- Bottom-up Integration

Five Steps of Integration Process

- The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
- Depending on the integration approach selected (ie., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
- Tests are conducted as each component is integrated.
- On completion of each set of tests, another stub is replaced with the real components.
- Regression testing may be conducted to ensure that new errors have not been introduced.

4.2.3 SYSTEM TESTING

The sub-system is integrated to make up the entire system. The testing process is concerned with finding errors that result from unanticipated interactions between sub-systems and system components. It is also concerned with validating that the system needs its functional and non-functional requirements.

Types of system testing: (i) Alpha testing (ii) Beta testing and (iii) Acceptance testing

1. **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the development organization.

The alpha test is conducted at the developer's site by the customer under the project team's guidance. In this test, users test the software on the development platform and point out errors for correction. However, the alpha test, because a few users on the development platform conduct it, has limited ability to expose errors and correcting them. Alpha tests are conducted in a controlled environment. Once the alpha test is over, the software product is ready for transition to the customer site for implementation and development.

2. Beta Testing. Beta testing is the system testing performed by a selected group of friendly customers.

Beta tests are conducted at customer site in an environment where the software is exposed to a number of users. The developer may or may not be present while the software is in use. So, beta test is a real life software experience without actual implementation. In this test, users record their observations, mistakes, errors and so on and report them periodically.

Differences between Alpha Testing and Beta Testing

S.No	Alpha Testing	Beat Testing
1.	In Alpha testing, users test the software at the developers site.	The software is tested at the users site.
2.	Alpha testing assesses the performance of the software in the environment in which it is developed.	Beta testing is 'live' testing and is conducted in an environment, which is not controlled by the developer .That is, this testing is performed without any interference from the developer.
3.	Alpha testing identifies all the errors present in the software and checks whether all the functions mentioned in the requirements are implemented properly in the software.	Beta testing evaluates the entire documentation of the software and checks whether the software is operating successfully in user environment.

3. **Acceptance Testing.** Acceptance testing is the system testing performed by the customer to determine whether to accept or reject the delivery of the system.

When customer 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.

4.3 SOFTWARE TESTING STRATEGIES

4.3.1 STATIC TESTING STRATEGIES

Static testing is the systematic examination of program structure for the purpose of showing that certain properties are true regardless of the execution path the program may take. Consequently, some static analyses can be used to demonstrate the absence of some faults from a program. Static testing represents actual behavior with a model based upon the programs semantic features and structure. Some of the strategies are,

- Formal technical reviews
- Code Walkthrough
- Code inspection
- Compliance with design and coding standards

4.3.2 FORMAL TECHNICAL REVIEWS

A review can be defined as " A meeting at which the software element is presented to project personnel, managers, users, customers or other interested parties for comment or approval "

The purpose of any review is to discover errors in analysis, design, and coding, testing and implementation phase of software development cycle. The other purpose of review is to see whether procedures are applied uniformly and in a manageable manner.

OBJECTIVE FOR REVIEWS

- To ensure that the software element confirms to its specifications.
- To ensure that the development of the software element is being done as per plans, standards and guidelines applicable for the project.
- To ensure that the changes to the software elements are properly implemented and affect only those system areas identified by the change specification.

TYPES OF REVIEWS

Reviews are basically of two types, informal technical review and formal technical review.

Informal Technical Review: Informal meeting and informal desk checking.

Formal Technical Review

Formal technical review is a software quality assurance activity performed by software engineering practitioners to improve software product quality. The product is scrutinized for completeness, correctness, consistency, technical feasibility, efficiency, and adherence to established standards and guidelines by the client organizations.

The review meeting

The meeting should consist of two to five people and should be restricted to not more than 2 hours. The aim of the review meeting is to review the product / work and the performance of people. The project leader contacts the review leader for the review. The review leader asks the reviewer to perform an independent review of the product / work before the scheduled FTR.

Result of FTR

- **Meeting decision**
  1. Whether to accept the product / work without any modification
  2. Accept the product / work with certain changes
  3. Reject the product / work due to error
- **Review summary report**
  1. What was reviewed?
  2. Who reviewed it?
  3. Findings of the review
  4. Conclusion

4.3.3 CODE WALKTHROUGH

A code walk through is an informal analysis of code as a cooperative, organized activity by several participants. The analysis is based mainly on the game of "playing the computer". That is participants select some test cases and simulate execution of the code by hand.

4.3.4 CODE INSPECTION

A code inspection, originally introduced by Fagan (1976) at IBM, is similar to a walkthrough but is more formal. Fagan's experiment, three separate inspections were performed: one following design, but prior to implementation, one following implementation, but prior to unit testing and one following unit testing. The inspection following unit testing was not considered to be cost effective in discovering errors.

The following is a list of some classical programming errors, which can be checked during code inspection:

- Use of un-initialized variables.
- Jumps into loops.
- Non – terminating loops.
- Incompatible assignments

- Arrays indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatches between actual and formal parameters in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.

DIFFERENCE BETWEEN WALKTHROUGH AND INSPECTION / REVIEW

- A walkthrough is less formal and has only a few steps, whereas inspections and reviews are more formal and logically sequential with many steps.
- Both processes are undertaken before actual development, and hence they are conducted on documents such as development plan, SOW, RDD and SRS design document and broad WBS to examine their authenticity, completeness, correctness, and accuracy.
- Both are costly but the cost incurred is comparatively much lower than the cost of repair at a later stage in the development cycle.

Differences between Code Walkthrough and Code Inspection

S.No	Code Walkthrough	Code Inspection
1.	Walkthrough is an informal review, which is used to evaluate the product. No preparation is made in advance and due to this, walkthroughs are termed as an unplanned activity.	Code inspection is a formal and systematic examination of the source code to detect errors.
2.	Walkthrough is quick.	Inspection takes more time.
3.	In a walkthrough, the author describes the product to the peers (group of individuals other than the author). The peers involved in this review do not have any specified responsibility.	In an inspection, the inspection team consisting of moderator, reader, recorder and author checks the source code for errors. Each member of the inspection team has a specified role to play.
4.	In a walkthrough, the author gets feedback from the peers in an informal way,	During inspection, the whole procedure is carried out in a formal manner.
5.	Walkthroughs are not considered efficient because they leave several errors unnoticed. In addition, no record of errors is maintained. It results in difficulties in verifying the correctness of the product after walkthroughs are 'over'.	Two checklists namely, code inspection checklist and inspection error list are prepared for recording the result of the code inspection. The code inspection checklist contains a summary of all errors of different types found in the software code. The inspection error list provides the details of each error that requires network.

4.3.5 DEBUGGING

DEFINITION

Debugging occurs as a consequence of successful testing. Debugging refers to the process of identifying the causes for defective behavior of a system .

CHARACTERISTICS OF BUGS

1. The symptoms and the cause may be geographically remote. ie., the symptoms may be appears in one part of a program, while the cause may actually be located at a sight that is far removed.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptoms may actually be caused by non errors (eg. round-off inaccuracies)
4. The symptoms may be caused by human errors.(ie., not easily traced)
5. The symptoms may be result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (eg a real-time application in which input ordering is indeterminate).
7. The symptoms may be due to causes that are distributed across a number of tasks running on different process.

LIFE CYCLE OF A DEBUGGING TASK

The following are various steps involved in debugging:

a. Defect identification / confirmation:

- A problem is identified in a system and a defect report is created.
- Defect assigned to a software engineer.
- The engineer analysis the defect report and performing the following actions:
  - What is expected / desired behavior of the system?
  - What is the actual behavior?
  - Is this really a defect in the system?
  - Can the defect be reproduced?

b. Defect analysis:

If the software engineer concludes that the defect is genuine, the focus shifts to understanding the root cause of the problem. This is often the most challenging step in any debugging task, particularly when the software engineer is debugging complex software.

Many engineers debug by using a debugging tool, generally a debugger and try to understand the root cause of the problem by following the execution of the program step-by-step. This approach may eventually yield successfully. However, in many situations, it takes too much time, and in some cases is not feasible, due to the complex nature of the program(s).

c. Defects resolution:

Once the root cause of a problem is identified, the defect can then be resolved by making an appropriate change to the system, which fixes the root cause

DEBUGGING APPROACHES

The following are some of the approaches that are used by the programmer for debugging:

Brute force method: This is the common method of debugging but is the least efficient method. In this approach, print statement are inserted throughout the program. The programmer may trace the value printed and locate the statement containing the error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

Backtracking: This is also a common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large for complex programs, limiting the use of this approach.

Cause elimination method: In this approach, once a failure is observed, the symptoms of the failure are noted. Based on the failure symptoms, the causes which could possibly have contributed to the symptoms is developed and tests are conducted to estimate each. A related technique of identification of the error from the symptom is the software fault tree analysis.

4.4 SOFTWARE TESTING TOOLS

4.4.1 NEED FOR TOOLS

Software testing tools is one that is used to find the right efficiency of the software and will provide with a way of checking on the software that user is using so that it is error free.

4.4.2 CLASSIFICATION OF TOOLS

Software testing tools can be categorized by the testing activity or the process they are utilized in, e.g. test planning, test execution, data comparison, defect capture, etc. Software tools are classified as follows:

- Function/ Regression tools
- Performance / Load testing tools.
- Testing process Management Tools

4.4.3 FUNCTION / REGRESSION TESTING TOOLS

Functional testing is a process used within software development in which software is tested to ensure that it conforms with all requirements. Its focus is on validating features, database access, security and functionality of the Application under Test. It uses external interfaces, including Application programming interfaces (APIs), Graphical user interfaces (GUIs), and Command line interfaces (CLIs).

The main objective of functional testing is to verify that each function of the software application operates in accordance with the written requirement specifications. FT life cycle includes Test Requirements Gathering, Test Planning, Test Strategy, Test Execution, Defect Management, Test Results Reporting, Test Metrics collection, Analysis and Improvement

Types of Function test include Unit, Smoke, Sanity, Integration, White box, Black Box, UAT and Regression Testing.

Functional Testing Tools includes *JMeter, SoapUI, and Watir.*

Regression Testing

Regression Testing is always used to verify that modified code does not break the existing functionality of the application and works within the requirements of the system.

- It is required when there is any change in requirements and code is modified according to the requirement
- New feature is added to the software
- Defect fixing
- During Release cycles (Alpha, Beta etc…)

Regression Testing techniques include Retest all, Regression Test Selection and Prioritization of test cases.

Important / Popular Test Tools for Functional and Regression Testing.

1) Selenium - (Open Source Tool)
2) UFT / QTP (Commercial Tool)
3) RFT (Commercial Tool)
4) SoapUI - (Open Source Tool)-SmartBear
5) Watir - (Open Source Tool)
6) SilkTest (Commercial Tool)
7) TestComplete - (Commercial Tool)
8) Cucumber (Open Source Tool)

4.4.4 PERFORMANCE / LOAD TESTING TOOLS

Performance testing tools are basically for system level testing, to see whether or not the system will stand up to a high volume of usage. A **load testing** is to check that the system can handle its expected number of transactions.

The purpose of the performance testing is to measure characteristics, such as response times, throughput or the mean time between failures (for reliability testing). This can be done in different ways depending on the tool, such as different user profiles, different types of activity, timing delays and other parameters. Adequately replicating the end-user environments or user profiles is usually key to realistic results.

If the performance is not up to the expected standard, then some analysis needs to be performed to see where the problem is and to know what can be done to improve the performance.

Features or characteristics of performance-testing tools are:

- To generate load on the system to be tested;
- To measure the timing of specific transactions as the load on the system varies;
- To measure average response times;
- To produce graphs or charts of responses over time

4.4.5 TESTING PROCESS MANGEMENT TOOLS

The features of **test management tools** are given below. Some tools will provide all of these features; others may provide one or more of the features.

- To manage the tests (like, keeping track of the same kind of data for a given set of tests, knowing which tests need to run in a common environment, number of tests planned, written, run, passed or failed);
- Scheduling of tests to be executed (manually or by a test execution tool);
- Managing the testing activities (time spent in test design, test execution, whether we are on schedule or on budget);
- Interfaces to other tools, such as:
    - *test execution tools (test running tools);*
    - *incident management tools;*
    - *requirement management tools;*
    - *configuration management tools;*
- Traceability of tests, test results and defects to requirements or other sources;
- To log the test results (note that the test management tool does not run tests but could summarize results from test execution tools that the test management tool interfaces with);
- To prepare progress reports based on metrics (quantitative analysis), such as:
    - *tests run and tests passed;*
    - *incidents raised, defects fixed and outstanding.*

This information can be used to monitor the testing process and decide what actions to take (test control). The tool also gives information about the component or system being tested (the test object). Test management tools help to collect, organize and communicate information about the testing on a project

4.4.6 BENEFITS OF TOOLS

- **Reduction of repetitive work:** People tend to make mistakes when doing the same task over and over. Examples of this type of repetitive work include running regression tests, entering the same test data again and again (can be done by a test execution tool), checking against coding standards (which can be done by a static analysis tool) or creating a specific test database (which can be done by a test data preparation tool).

- **Greater consistency and repeatability:** People have tendency to do the same task in a slightly different way even when they think they are repeating something exactly. A tool will exactly reproduce what it did before, so each time it is run the result is consistent.

- **Objective assessment:** If a person calculates a value from the software or incident reports, by mistake they may omit something, or their own one-sided preconceived judgments or convictions may lead them to interpret that data incorrectly. Using a tool means that subjective preconceived notion is removed and the assessment is more repeatable and consistently calculated.

- **Ease of access to information about tests or testing:** Information presented visually is much easier for the human mind to understand and interpret. For example, a chart or graph is a better way to show information than a long list of numbers – this is why charts and graphs in spreadsheets are so useful. Special purpose tools give these features directly for the information they process.

4.4.7 RISK ASSOCIATED WITH TOOLS

Risks include:

- **Unrealistic expectations from the tool:** Unrealistic expectations may be one of the greatest risks to success with tools. The tools are just software and there are many problems associated with any kind of software. It is very important to have clear and realistic objectives for what the tool can do.
- **People often make mistakes by underestimating the time, cost and effort for the initial introduction of a tool:** Introducing something new into an organization is hardly straightforward. Once you purchase a tool, you want to have a number of people being able to use the tool in a way that will be beneficial. There will be some technical issues to overcome, but there will also be resistance from other people – both need to be handled in such a way that the tool will be of great success.
- **People frequently miscalculate the time and effort needed to achieve significant and continuing benefits from the tool:** Mostly in the initial phase when the tool is new to the people, they miscalculate the time and effort needed to achieve significant and continuing benefits from the tool.

- **Mostly people underestimate the effort required to maintain the test assets generated by the tool:** Generally, people underestimate the effort required to maintain the test assets generated by the tool. Because of the insufficient planning for maintenance of the assets that the tool produces there are chances that the tool might end up as 'shelf-ware', along with the previously listed risks.

- **People depend on the tool a lot (over-reliance on the tool):** Since there are many benefits that can be gained by using tools to support testing like reduction of repetitive work, greater consistency and repeatability, etc. people started to depend on the tool a lot. But the tools are just a software they can do only what they have been designed to do (at least a good quality tool can), but they cannot do everything. A tool can definitely help, but it cannot replace the intelligence needed to know how best to use it, and how to evaluate current and future uses of the tool.

4.4.8 SELECTING TOOLS

Choosing an automated software testing tool is an important step, and one which often poses enterprise-wide implications. Here are several key issues, which should be addressed when selecting an application testing solution. Below are some basic guidelines for selecting and evaluating Software Testing tools for different phases:

Test Planning and Management

A robust testing tool should have the capability to manage the testing process, provide organization for testing components, and create meaningful end-user and management reports. It should also allow users to include non-automated testing procedures within automated test plans and test results. A robust tool will allow users to integrate existing test results into an automated test plan. Finally, an automated test should be able to link business requirements to test results, allowing users to evaluate application readiness based upon the application's ability to support the business requirements.

Testing Product Integration

Testing tools should provide tightly integrated modules that support test component reusability. Test components built for performing functional tests should also support other types of testing including regression and load/stress testing. All products within the testing product environment should be based upon a common, easy-to-understand language. User training and experience gained in performing one testing task should be transferable to other testing tasks. Also, the architecture of the testing tool environment should be open to support interaction with other technologies such as defect or bug tracking packages.

Internet/Intranet Testing

A good tool will have the ability to support testing within the scope of a web browser. The tests created for testing Internet or intranet-based applications should be portable across browsers, and should automatically adjust for different load times and performance levels.

Ease of Use

Testing tools should be engineered to be usable by non-programmers and application end-users. With much of the testing responsibility shifting from the development staff to the departmental level, a testing tool that requires programming skills is unusable by most organizations. Even if programmers are responsible for testing, the testing tool itself should have a short learning curve.

GUI and Client/Server Testing

A robust testing tool should support testing with a variety of user interfaces and create simple-to manage, easy-to-modify tests. Test component re-usability should be a cornerstone of the product architecture.

Load and Performance Testing

The selected testing solution should allow users to perform meaningful load and performance tests to accurately measure system performance. It should also provide test results in an easy-to-understand reporting format.

Methodologies and Services

For those situations that require outside expertise, the testing tool vendor should be able to provide extensive consulting, implementation, training, and assessment services. The test tools should also support a structured testing methodology.

4.4.9 DIFFERENT CATEGORIES OF TOOLS

Classification of different types of test tools according to the test process activities:

The tools are grouped by the testing activities or areas that are supported by a set of tools, for example, tools that support management activities, tools to support static testing, etc.

Following are the classification of different types of test tools according to the test process activities. The **'(D)'** written after the types of tool indicates that these tools are **mostly used by the developers.** The various types of test tools according to the test process activities are:

1. Tool support for management of testing and tests:

- *Test management tools*
- *Requirements management tools*
- *Incident management tools*
- *Configuration management tools*

2. Tool support for static testing:

- *Review process support tools*
- *Static analysis tools (D)*
- *Modelling tools (D)*

3. Tool support for test specification:

- *Test design tools*
- *Test data preparation tools*

4. Tool support for test execution and logging:

- *Test execution tools*
- *Test harness/ Unit test framework tools (D)*
- *Test comparators*
- *Coverage measurement tools (D)*
- *Security tools*

5. Tools support for performance and monitoring:

- *Dynamic analysis tools (D)*
- *Performance testing, Load testing and stress-testing tools*
- *Monitoring tools*

4.4.11 EXAMPLES FOR COMMERCIAL SOFTWARE TESTING TOOL

1) Open Source Tools

a) Test Management tools

- TET (Test Environment Toolkit)
    - The goal behind creating the Test Environment Toolkit (TET) was to produce a test driver that accommodated the then current and anticipated future testing needs of the test development community. To achieve this goal, input from a wide sample of the community was used for the specification and development of TET's functionality and interfaces.
- TETware
    - The TETware is the Test Execution Management Systems which allows to do the test administration, sequencing of test, reporting of the test result in the standard format (IEEE Std 1003.3 1991) and this tool is supports both UNIX as well as 32-bit Microsoft Windows operating systems, so portability of this is with test cases you developed. The TETware tools allow testers to work on a single, standard, test harness, which helps you to deliver software projects on time. This is easily available for download on ftp download.
- Test Manager
    - The Test Manager is an automated software testing tool is used in day to days testing activities. The Java programming language is used to develop this tool. Such Test Management tools are used to facilitate regular Software Development activities, automate & mange the testing activities. Currently Test Manager 2.1.0 is ready for download. If you want to learn more information of Test Manager,
- RTH
    - RTH is called as "Requirements and Testing Hub". This is a open source test management tool where you can use as requirement management tool along with this it also provides the bug tracking facilities.

b) Functional Testing Tools

- Selenium
- Soapui
- Watir
- HTTP::Recorder
- WatiN
- Canoo WebTest

- Webcorder
- Solex
- Imprimatur
- SAMIE
- Swete
- ITP
- WET
- WebInject
- Katalon Studio

c) Load Testing Tools

- Jmeter
- FunkLoad

2) Proprietary/Commercial tools

a) Test Management tools

- HP Quality Center/ALM
- QA Complete
- T-Plan Professional
- Automated Test Designer (ATD)
- Testuff
- SMARTS
- QAS.TCS (Test Case Studio)
- PractiTest
- Test Manager Adaptors
- SpiraTest
- TestLog
- ApTest Manager
- DevTest

b) Functional Testing Tools

- QuickTest Pro
- Rational Robot
- Sahi
- SoapTest
- Badboy
- Test Complete
- QA Wizard
- Netvantage Functional Tester
- PesterCat
- AppsWatch
- Squish
- actiWATE
- liSA
- vTest
- Internet Macros
- Ranorex

c) Load Testing Tools

- WebLOAD Professional
- HP LoadRunner
- LoadStorm
- NeoLoad
- Loadtracer
- Forecast
- ANTS – Advanced .NET Testing System
- vPerformer
- Webserver Stress Tool
- preVue-ASCII
- Load Impact

4.5 CODE OF ETHICS FOR SOFTWARE PROFESSIONALS

Each person has some goals to achieve in the life both personal and professional goals. Every software professional should be aware of all the ethical issues connected with software engineering. those, code of ethics was formed by IEEE in 1999.

4.5.1 HUMAN ETHICS

Every human being has certain responsibilities towards the society and the surrounding environment. As good citizens of the world, it is the duty to contribute something to make the world a better place. In order to achieve this, certain principles have to be followed. It is the responsibility of software professionals to follow set of rules and regulations set by the Government, for the growth, prosperity and security of the country. Ethical issues are highly subjective and have to be guided by one's conscience.

4.5.2 PROFESSIONAL ETHICS

Professionals like doctor, engineer, chartered accountant has the highest responsibility and the greater role in designing the future for the rest of the society. Every professional has to follow the code of ethics. Each member of the professional body of IEEE is expected to follow the code of ethics created or formed by it.

Some of the ethical codes to be followed are,

1. To accept responsibility in making engineering decisions with safety, health and welfare of the public and to disclose the factors that endanger the public.
2. To avoid the conflicts of interests whenever possible.
3. To be honest and realistic in stating estimations based on available data.
4. To reject bribery in all forms.
5. To improve the understanding of technology.
6. To accept the honest criticism of technical work to acknowledge and correct errors.
7. To assist colleges and co-workers in their professional development and to support them in following the code of ethics.

4.5.3 ETHICAL ISSUES IN SOFTWARE ENGINEERING

Ethical issues play a big role in the analysis and development of software products. It implicates and indicates the existence of professional codes of practices.

4.5.4 CODE OF ETHICS AND PROFESSIONAL PRACTICE

Software products are developed and maintained by humans. Most society of professionals have a code of ethics to which it should be followed by all its members. The two major societies for computer professionals are Association for Computing Machinery (ACM) and the computer society of the Institute of Electrical and Electronics Engineers (IEEE - CS) jointly approved the software engineering code of ethics and professionals practice which is the standard software engineering.

indicates the existence of professional codes of practices.

4.5.5 . SOFTWARE ENGINEERING CODE OF ETHICS AND PROFESSIONAL PRACTICE

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. **PUBLIC** - Software engineers shall act consistently with the public interest.

2. **CLIENT AND EMPLOYER** - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. **PRODUCT** - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. **JUDGMENT** - Software engineers shall maintain integrity and independence in their professional judgment.

5. **MANAGEMENT** - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. **PROFESSION** - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. **COLLEAGUES** - Software engineers shall be fair to and supportive of their colleagues.

8. **SELF** - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

4.5.5 ETHICAL ISSUES

Ethics is the question of right versus wrong and then doing what is right.

It is a simple ethical problem as well as “right vs right” and “right vs wrong” dilemmas where the employee knows that the employer is motivated to act unethically with respect to the customer.

It deals with the moral principles that help the individual to make the right decision in a specific ethical problem to act accordingly.

SUMMARY

- Software testing gives an important set of methods that can be used to evaluate and assure that a program or system meets its non-functional requirements.
- A test oracle is a mechanism, different from the program itself, which can be used to check the correctness of the output of the program for the test cases.
- Error is a discrepancy between actual value of the output given by the software and the specified correct value of the output for that given input.
- **Syntax Error:** A syntax error is program statements that violate one or more rules of the language in which it is written. **Logic Error:** A logic error deals with incorrect data fields, out of range terms and invalid combinations.
- Fault is a condition that causes a system to fail in performing its required function. Failure is the inability of the software to perform a required function to its specification.
- Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester. White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
- The following are the two main approaches available to design black-box test cases: Equivalence class partitioning and Boundary value analysis
- For White-box testing strategies, the methods used are: 1. Coverage Based Testing 2. Cyclomatic Complexity 3. Mutation Testing
- Test plan is a document consisting of different test cases designed for different testing objects and different testing attributes
- In unit testing individual components are tested to ensure that they operate correctly.
- *The purpose of Integration testing is to verify functional, performance and reliability requirements placed on major design items.*
- **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the development organization.
- **Beta Testing.** Beta testing is the system testing performed by a selected group of friendly customers.

- Acceptance testing is the system testing performed by the customer to determine whether to accept or reject the delivery of the system.
- Reviews are basically of two types, informal technical review and formal technical review. **Informal Technical Review:** Informal meeting and informal desk checking. **Formal Technical Review:** A formal software quality assurance activity through various approaches such as structured walkthroughs, inspection, etc.
- A code walk through is an informal analysis of code as a cooperative, organized activity by several participants.
- The basic difference between the two is that a walkthrough is less formal and has only a few steps, whereas inspections and reviews are more formal and logically sequential with many steps.
- Debugging occurs as a consequence of successful testing. Debugging refers to the process of identifying the causes for defective behavior of a system .
- Software testing tools can be categorized by the testing activity or the process they are utilized in, e.g. test planning, test execution, data comparison, defect capture, etc.
- Software tools are classified as follows: Function/ Regression tools , Performance / Load testing tools , Testing process Management Tools
- Functional testing is a process used within software development in which software is tested to ensure that it conforms with all requirements.
- Regression Testing is always done to verify that modified code does not break the existing functionality of the application and works within the requirements of the system.
- **Performance testing tools** are basically for system level testing, to see whether or not the system will stand up to a high volume of usage. A **load testing** is to check that the system can handle its expected number of transactions.

REVIEW QUESTIONS

PART - A (2 Marks Questions)

1. What is software testing? What is test oracle?
2. What are the types of errors? Define the term “Fault”
3. What is black box testing?
4. What are the two main approaches available to design black-box test cases?
5. What are the methods used for white box -testing?
6. What is cyclomatic complexity?
7. Write down the formula for finding the cyclomatic complexity of the control flow graph.
8. Define the term “ Mutant”
9. What are the three levels of system testing?
10. What is a driver? What is a stub?

11. What is the objective of integration testing?
12. What is alpha testing? What is beta testing?
13. What are the different types of static testing strategies?
14. Define the term “Review” . What are the two types of reviews?
15. Define the term “Debugging”.
16. How the software classification tools are classified?
17. Give examples for function testing and Regression testing tools.

PART - B (3 Marks Questions)

1. State any three testing objectives.
2. What are the three pieces of information contained in a test case?
3. State the main difference between black box testing and white box testing.
4. State any three common coverage criteria.
5. List the guidelines and objectives of testing phase.
6. What is cyclometric complexity?
7. What are the two possibilities for running a mutant program?
8. State any three reasons for choosing unit testing.
9. What is integration testing? What is the main purpose of integration testing?
10. What are the objectives of for review?
11. What are the programming errors, which can be checked during code inspection?
12. Write down the differences between walkthrough and inspection / Review.
13. What are the benefits of software testing tools?
14. What is human ethics? What is professional ethics?

PART - C (5/10 Marks Questions)

1. What are the basic terms used in testing? Explain.
2. Discuss elaborately about the advantages and disadvantages of black box testing and white box testing.
3. Explain equivalence class portioning with example.
4. Explain boundary value analysis strategy of black box testing.
5. Briefly explain about control flow graph.
6. Briefly explain about mutant testing.
7. What are the different types of system testing? Briefly explain about them.
8. Briefly explain the following (i) Code walkthrough and (ii) Code Inspection
9. Explain the life cycle of a debugging task.
10. Briefly explain about debugging approaches.

11. Briefly explain about some of the commercial software testing tools.
12. Write down the differences between walkthrough and inspection / Review.
13. What are the benefits of software testing tools?
14. Explain about the selection of software testing tools.
15. Explain different categories of software testing tools?
16. Explain about the risks involved in using software testing tools.
17. Explain about the Software Engineering Code of ethics and Professional Practice

UNIT – V	

SOFTWARE RELIABILITY AND QUALITY ASSURANCE

OBJECTIVES

At the end of the unit, the students will be able to

- Define Software quality Assurance
- Differentiate verification and validation.
- Classify software qualities.
- Explain SEI - CMM model.
- Understand the need, benefits and uses of ISO Certification.
- Define reliability terminologies
- Explain different types of reliability growth modelling
- Know the purpose, characteristics and applications of reverse engineering.
- Differentiate re-engineering from forward engineering.

5.0 INTRODUCTION

Software becomes more reliable over time, instead of wearing out. It becomes obsolete as the environment for which it was developed changes. Hardware redundancy allows us to make a system reliable as we desire, if we use large number of components with given reliability.

5.1 SOFTWARE QUALITY ASSURANCE

5.1.1 VERIFICATION AND VALIDATION

Verification and validation is not the same thing although they are easily confused. The difference between them is expressed by Boehm :

- 'Validation: Are we building the right product?'
  - The software should confirm to its specification
- 'Verification: Are we building the product right?'
  - The software should do what the user really requires

VERIFICATION

Verification is the process of determining whether the output of one phase of software development confirms to that of its previous phase. (OR)

Verification involves checking of function and non-functional requirements that the software confirms to its specification.

VALIDATION

Validation is the process of determining whether a fully developed system confirms to its requirements specifications. (OR)

Validation is an analysis process that is carried out after checking conformance of the system to its specification.

5.1.2 SOFTWARE QUALITY ASSURANCE

The aim of the Software Quality Assurance (SQA) process is to develop high-quality software product. *Software Quality Assurance is a set of activities designed to evaluate the process by which software is developed and / or maintained.*

Quality assurance is a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product confirms to established technical requirements

The purpose of a software quality assurance group is to provide assurance that the procedures, tools, and techniques used during product development and modification are adequate to provide the desired level of confidence in the work products.

The process of Software Quality Assurance

1. Defines the requirements for software controlled system fault / failure detection, isolation, and recovery.
2. Reviews the software development processes and products for software error prevention and / or controlled change to reduced functionality states; and
3. Defines the process for measuring and analyzing defects as well as reliability and maintainability factors.

5.1.3 SOFTWARE QUALITY ASSURANCE - OBJECTIVES

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

5.1.4 SOFTWARE QUALITY ASSURANCE PLAN

A plan that defines the quality processes and procedures that should be used. This involves selecting and instantiating standards for products and processes and defining the required quality attributes of the system.

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

The quality plan should select those organizational standards that are appropriate to a particular product and development process. New standards may have to be defined if the project uses new methods and tools.

An outline structure for a quality plan includes:

1. **Product introduction:** A description of the product, its intended market and the quality expectations for the product.
2. **Product plans:** The critical release dates and responsibilities for the product along with plans for distribution and product servicing.
3. **Process descriptions:** The development and service processes that should be used for product development and management.
4. **Quality goals:** The quality goals and plans for the product including an identification and justification of critical product quality attributes.
5. **Risks and risk management:** The key risks that might affect product quality and the actions to address these risks.

Preparation of a Software quality Assurance Plan for each software project is a primary responsibility of the software quality assurance group. Software Quality Assurance Plan contains the following details..

- Purpose-scope of plan;
- List of references to other documents;
- Management, including organization, tasks and responsibilities;
- Documentation to be produced;
- Standards, practices and conventions;
- Reviews and audits;
- Testing;
- Problems reporting and corrective action;
- Tools, techniques and methodologies;
- Code, media and supplier control;
- Records collection, maintenance and retention;
- Training;
- Risk management-the methods of risk management that are to be used

5.1.5 SOFTWARE QUALITY

DEFINITION OF SOFTWARE QUALITY

Software Quality is the "Conformance to explicit stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software "

CLASSIFICATIONS OF SOFTWARE QUALITIES

There are many desirable software qualities. Some of these apply both to the product and to the process used to produce the product. The user wants the software products to be reliable, efficient, and easy to use. The producer of the software wants it to be verifiable, maintainable, portable, and extensible. The manager of the software project wants the process of software development to be productive and easy to control.

EXTERNAL VERSUS INTERNAL QUALITIES

The external qualities are visible to the users of the system. The internal qualities are those that concern the developers of the system. In general, users of the software only care about the external qualities. But internal qualities deals largely with the structure of the software-that help developers to achieve the external qualities. For example, the internal quality of verifiability is necessary for achieving the external quality of reliability.

PRODUCT AND PROCESS QUALITIES

A process is used to produce the software product. We can also attribute some qualities to the process, although process qualities often are closely related to product qualities. For example, if the process requires careful planning of system test data before any design and development of the system starts, products reliability will increase. Some qualities, such as efficiency, apply both to the product and to the process.

Product is usually refers to what is delivered to the customer. Even though this is an acceptable definition from the customer's perspective, it is not adequate for the developer who requires a general definition of a software product that encompasses not only the object code and the user manual, but also the requirements, design, source code, test data, etc. In fact, it is possible to deliver different subsets of the same product to different customers.

SOFTWARE QUALITIES ATTRIBUTES

Functionality: The capability to provide functions which meet stated and implied needs when the software is used.

Reliability: The capability to maintain a specified level of performance.

Usability: The capability to be understood, learned, and used.

Efficiency: The capability to provide appropriate performance relative to the amount of resources used.

Maintainability: The capability to be modified for purposes of making corrections, improvements, or adaptation.

Portability: The capability to be adapted for different specified environments without applying actions or means other than those provided for this purpose in the product.

IMPORTANT QUALITIES OF SOFTWARE PRODUCT

CORRECTNESS : A program is functionally correct if it behaves according to the specification of the functions it should provide (called functional requirements specifications).

RELIABILITY : Informally, software is reliable if the user can depend on it. The specialized literature on software reliability defines reliability in terms of statistical behavior-the portability that the software will operate as expected over a specified time interval.

ROBUSTNESS: A program is robust if it behaves "reasonably," even in circumstances that were not anticipated in the requirements specification

PERFORMANCE : Performance affects the usability of the system. If a software system is too slow, it reduces the productivity of the users. If a software system uses too much disk space, it may be too expensive to run. If a software system uses too much memory, it may affect the other applications that are run on the same system, or it may run slowly while the operating system tries to balance the memory usage of the different applications.

VERIFIABILITY: A software system is verifiable if its properties can be verified easily. For example, it is important to be able to verify the correctness or the performance of a software system.

REPARABILITY: A software system is repairable if it allows the correction of its defects with a limited amount of work.

EVOLVABILITY : Like other engineering products, software products are modified over time to provide new functions or to change existing functions. Indeed, the fact that software is so malleable makes modifications extremely easy to apply to an implementation.

UNDERSTANDABILITY: Some software systems are easier to understand than others. Of course, some tasks are inherently more complex than others.

INTEROPERABILITY : "Interoperability" refers to the ability of a system to coexist and cooperate with other systems. With interoperability, a vendor can produce different products and allow the user to combine them if necessary.

PRODUCTIVITY : Productivity is a quality of the software production process; it measures the efficiency of the process.

TIMELINESS : Timeliness is a process – related quality that refers to the ability to deliver a product on time. Timeliness requires careful scheduling, accurate estimation of work, and clearly specified and verifiable milestones.

VISIBILITY : A software development process is visible if all of its steps and its current status are documented clearly. Another term used to characterize this property is transparency. The idea is that the steps and the status of the project are available and easily accessible for external examination.

IMPORTANCE OF SOFTWARE QUALITY

1. **Increasing Criticality of Software:** The final customer or user is naturally anxious about the general quality of software, especially its reliability. This is increasingly the case as organizations become more dependent on their computer systems and software is used more and more in areas which are safely critical, for example to control aircraft.
2. **The Intangibility of Software:** This makes us difficult to know whether a particular task in a project has been completed satisfactorily. The results of these tasks can be made tangible by demanding that the developers produce 'deliverables' that can be examined for quality.
3. **Accumulating Errors during Software Development.** As computer system development is made up of a number of steps where the output from one step is the input to the next, the errors in the earlier deliverables will be added to those in the later steps leading to an accumulating detrimental effect, in general, the later in a project that an error is found , the more expensive is will be to fix. In addition, because the number of errors in the system is unknown the debugging phases of a project are particularity difficult to control.

5.1.6. CAPABILITY MATURITY MODEL

SEI-CMM stands for Software Engineering Institute Capability Maturity Model. This model helps in judging the maturity of software process of an organization. It also helps to identify the main process that will help in increasing maturity of these processes. It has become a standard for assessing and improving software processes.

Five levels of process maturing have been proposed for software industry by SEI-CMM. These levels are defined as follows:

1. **Level 1 (Initial).** The software process is adhoc, and even chaotic at time. The organization whose success depends on individual effort and processes are not defined and documented come under this level. Organization at this level can benefit most by improving project management, quality assurance and change control.
2. **Level 2 (Repeatable):** Company defines the basic management practices such as tracking cost; schedule and functionality are established but not the procedures of doing it. Here the efforts done previously for success of project may repeat. **Some of the characteristics of a process at this level are:** Project commitments are realistic and based on past experience with similar projects, cost and schedule are tracked and problems resolved when they arise, formal configuration control mechanisms are in place, and software project standards are defined and followed.

3. **Level 3(Defined).** The organization-wide software process includes management and engineering procedures. These procedures are well defined, documented, standardized and integrated. All projects make use of the documented and approved version of the organization process for software development and maintenance. But the process and practices are not analyzed quantitatively. In this process both the development and management processes are formal. ISO 9000 aims at achieving this level.

4. **Level 4(Managed).** At this level, the focus is on software metrics. Two types of metric are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics reflect the effectiveness of the process being used, such as the average defect correction time, productivity, the average number of defects found per hour of inspection, the average number of failures detected during testing per LOC, and so forth. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than to improve the process.

   Software processes and products are quantitatively understood; measured and controlled using detailed procedures.

5. **Level 5(Optimized).** By using the quantitative feedback from the process in place, exploring new ideas and technologies. At this level, an organization is committed to continuous process improvement. Process improvement is budgeted and planned and is an integral part of the organization's process. The organization has the means to identify weakness and strengthen the process proactively, with the goal of preventing the occurrence of defects. Best software engineering and management practices are used throughout the organization.

Except for level 1, each maturity level is characterized by several Key Process Areas (KPAs)

5.1.7 INTERNATIONAL STANDARD ORGANISATION (ISO)

INTRODUCTION TO ISO

The International Organization for Standardization (ISO) is worldwide federations of national standards bodies for some 100 countries. ISO is a non-governmental organization established in 1947.

The ISO 9000 standard specifies quality assurance elements in generic terms, which can be applied to any business regards of the product or services being offered. In order to register for one of the quality assurance system models contained in ISO 9000, third party auditors examine an organization's quality system and operations for compliance to the standard and for effective operations. Upon successful audit, the organization receives a certificate from a registered body represented by the auditors. Thereafter, semi-annual audits ensure conformance to the standard.

The ISO 9000 standards view an organization as a set of interrelated process. In order to pass the criteria for ISO 9000 compliance, the processes must address the identified areas, and document and practice them. When a process is documented, it is better understood, controlled, and improved. However, ISO 9000 does not specify how an organization should implement the quality system. Therefore, the biggest

challenge is to design and implement a quality assurance system that meets the standard and gets well with the products/service of the organization.

ISO 9001 is a quality assurance standard that is specific to software engineering. It specifies 20 standards with which an organization must comply for an effective implementation of the quality assurance system.

ISO-9000 series of standards is a set of documents dealing with quality systems that can be used for quality assurance purpose. ISO-9000 series is not just software standard. *It is a series of five related standards that are applicable to a wide variety of industrial activities, including design/development, production, installation, and servicing.*

NEED FOR ISO CERTIFICATION

- It is sign of customer confidence. This certification has become a standard for international bidding.
- It is a motivating factor to the business organizations.
- It makes the processes more focused, efficient and cost effective.
- It helps in designing high quality repeatable software products.
- It highlights weaknesses and suggests corrective measures for improvements.
- It facilitates the development of optimal process and total quality measurement.
- It emphasizes the need of proper documentation.

PROCEDURE FOR GETTING ISO 9000 Certification

The ISO 9000 registration process consists of the following stages:-

1. **Application.** Once an organization decides to go for ISO 9000 certification, it applies to a registrar for registration.
2. **Pre-assessment.** During this stage, the registrar makes a rough assessment of the organization.
3. **Document Review and Adequacy of Audit.** During this stage, the registrar reviews the documents submitted by he organization and makes suggestions for possible improvements.
4. **Compliance Audit**. During this stage, the registrar checks whether the suggestions made by it during review have been compiled with by the organization or not.
5. **Registration.** The registrar awards the ISO 9000 certificate after successful completion of all previous phases.
6. **Continued Surveillance.** The registrar continues to monitor the organizations periodically.

BENEFITS OF ISO 9000 CERTIFICATION

1. **Continuous Improvement. Business:** ISO-9000 forces an organization to focus on "how they do business". Each procedure and work instruction must be documented and thus there will be a continuous Improvement.

2. **Eliminate Variation:** Documented processes are the basis for repetition and help eliminate variation within the process. As variation is eliminated, efficiency improves. As efficiency improves , the cost of quality is reduced.
3. **Higher Real and Perceived Quality.** With the development of solid Corrective and Preventative measures, permanent, company-wide solutions to quality problems are found. This results in higher quality.
4. **Boost Employee Morale.** Employee morale is increased as they are asked to take control of their processes and document their work processes.
5. **Improved Customer Satisfaction:** Customer satisfaction grows as a company transforms from a reactive organization to a pro-active, preventative organization. It becomes a company people want to do business with.
6. **Increased Employee Participation:** Reduced problems resulting from increased employee participation, involvement, awareness and systematic employee training.
7. **Better Product and Services:** Better products and services result from Continuous Improvement Processes.
8. **Greater Quality Assurance:** Fosters the understanding that quality, in and of itself, is not limited to a quality department but is everyone's responsibility.
9. **Improved Profit Levels:** Improved profit levels result as productivity improves and rework costs are reduced.
10. **Improved Communication.** Improved communications both internally and externally which improves quality, efficiency, on time delivery and customer/supplier relations.
11. **Reduced Cost.** ISO standards results into a reduced cost of the product.
12. **Competitive Edge.** In order to serve to deal with the increased business complexity and for offering higher customer services, ISO 9000 standards adds as competitive edge.

LIMITATION OF ISO 9000 CERTIFICATION

- ISO 9000 does not provide any guideline for defining an appropriate process.
- ISO 9000 certification process is not fool proof and no international accrediting agency exists.
- ISO 9000 does not automatically lead to continuous process improvement, i.e., it does not automatically lead to TQM.

ISO 9126

ISO 9126 standard was published in 1991 to tackle the question of the definition of software quality. This 13 page document was designed as a foundation upon which further, more detailed, standards could be built.

Six software quality characteristics of ISO 9126:

- *Functionality,* which covers the functions that a software product provides to satisfy user needs;
- *Reliability,* which relates to the capability of the software to maintain its level of performance;

- *Usability,* which relates to the effort needed to use the software;
- *Efficiency,* which relates to the physical resources used when the software is executed;
- *Maintainability,* which relates to the effort needed to of make changes to the software;
- *Portability,* which relates to the ability of the software to be transferred to a different environment;

5.1.13. COMPARISON BETWEEN ISO 9000 CERTIFICATION AND SEI-CMM

- The emphasis of the SEI-CMM is on continuous process improvement whereas ISO 9000 addresses the minimum criteria for an acceptable quality system.
- The Capability Maturity Model (CMM) is a five-level framework for measuring software engineering practices, as they relate to process. On the other hand, ISO 9000 defines a minimum level of generic attributes for a quality management program.
- ISO 9000 is awarded by an international standard body. Therefore, ISO 9000 certification can be quoted by an organization in official documents, In communication with external parties and in tender quotations. However, SEI-CMM assessment is purely for internal use.
- ISO 9000 standards were basically designed to audit manufacturing / service organizations whereas CMM was developed specifically for software industry and this addresses several issues specific to software industry.
- The SEI-CMM focuses strictly on software, while ISO 9000 has a much wider scope: hardware, software, processed materials and services.
- SEI-CMM model provides a list of key process areas (KPAs) on which on organization at any maturity level needs to concentrate to take it from one maturity level to the next. Thus, it provides a way for achieving gradual quality improvement.
- SEI CMM aims for achieving Total Quality Management (TQM), which is beyond quality assurance, whereas ISO 9000 aims at level 3 (defined level) of SEI-CMM.
- ISO 9000 requires that procedures for handling storage, packaging, and delivery be established and maintained, . Replication, delivery and installation are not covered in the SEI-CMM.
- ISO 9000 requires that the product be identified and traceable during all stages of production, delivery, and installation. SEI-CMM covers this clause primarily in software configuration management.

5.2 SOFTWARE RELIABILTY

5.2.1 DEFINITION

Software reliability is defined as that software will provide failure-free operation in a fixed environment for fixed interval of time.

(or)

Reliability of a software product can also be defined as the probability of the product working correctly over a given period of time.

The software reliability curve is shown in figure 5.2

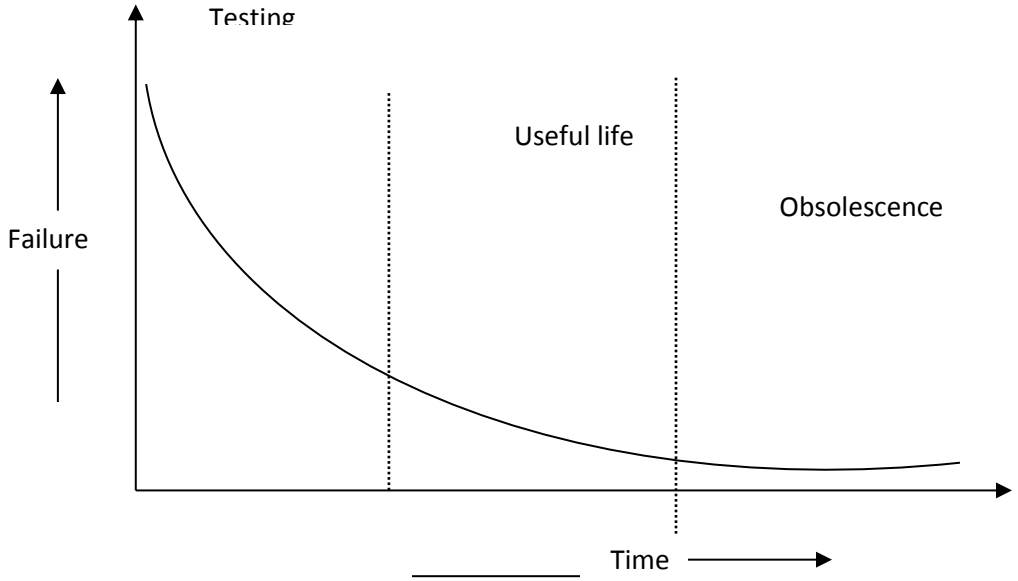


Fig No 5.2. Software Reliability Curve

Software may be retired only if it becomes obsolete. Some of the contributing factors for retiring is given below:

- Change in environment
- Change in infrastructure / technology
- Major change in requirements
- Increase in complexity
- Extremely difficult to maintain
- Deterioration in structure of the code
- Slow execution speed
- Poor graphical user interfaces.

5.2.2 RELIABILITY TERMINOLOGIES

System Failure: An event that occurs at some point in time when the system does not deliver a service as expected by its users.

System Error: Erroneous system behavior where the behavior of the system does not conform to its specification.

System Fault: An incorrect system state, i.e. a system state that is unexpected by the designers of the system.

Human error or mistake: Human behavior that results in the introduction of faults into a system.

5.2.3 CLASSIFICATION OF FAILURES

Many large systems are divided into small subsystems and each subsystem has different reliability requirements. It is easy and cheap to assess the reliability requirements of each sub-system separately.

Failure Class	Description
Recoverable	System can recover with or without operator intervention.
Transient	Occur only with certain inputs.
Unrecoverable	System cannot recover without operator intervention or in it the system may need to be restarted.
Corrupting	Failure corrupts system data.
Non-corrupting	Failure does not corrupt system data.
Permanent	Occur for all input values while invoking a function of the system.

5.2.4 RELIABILITY METRICES

Reliability metrics are used to quantitatively express the reliability of a software product. Some reliability metrics which can be used to quantify the reliability of a software product are:

1. **MTTF (Mean Time to Failure).** Components have some average lifetime and this is reflected in the most widely used hardware reliability metrics, Mean Time to failure (MTTF). The MTTF is the mean time for which a component is expected to be operational. The MTTF is the average time between observed system failures. An MTTF of 500 means that one failure can be expected every 500 time units. The time units are totally dependent on the system and it can even be specified in the number of transactions, as is the case of database query systems.
   MTTF is relevant for systems with long transactions, i.e., where system processing takes a long time. MTTF should be longer than transaction length. For example, it is suitable for computer-aided design systems where a designer will work on a design for several hours, word processing systems.

2. **MTTR (Mean Time to Repair).** Once failure occurs, some time it is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and then to fix them.

   (or)

   MTTR is the average time to replace a defective component.

   Once a hardware component fails then the failure is usually permanent, so the 'Mean Time to Repair' (MTTR) which reflects the time taken to repair or replace the component.

3. **MTBF (Mean Time Between Failures).**

$$MTBF = MTTF + MTTR$$

Thus, MTBF of 500 hours indicates than once a failure occurs, the next failure is expected to occur only after 500 hours. In this case, the time measurements are real time and not the execution time as in MTTF.

4. **POFOD (Probability of Failure on Demand).** POFOD is the likelihood that the system will fail when a service request is made. A POFOD of 0.01 means that one out of a hundred service requests may result in failure.

5. **ROCOF (Rate of Occurrences of Failure).** ROCOF is the frequency of occurrence with which unexpected behavior is likely to occur. A ROCOF of 3/100 means that three failures are likely to occur in each 100 operational time units. This metric is sometimes called the failure intensity.

6. **AVAIL(Availability).** Availability is the probability that the system is available for use at a given time. Availability of 0.995 means that in every 1000 time units, the system is likely to be available for 995 of these.

5.2.5 RELIABILITY GROWTH MODELING

A reliability growth model is a mathematical model of how software reliability improves when errors are detected and repaired. A reliability growth model can be used to predict when a particular level of reliability is likely to be attained. Thus, reliability growth modeling can be used to determine when to stop testing to attain a given reliability level. Although several reliability growth models have been proposed as follows:

JELINSKI-MORANDA MODEL

The Jelinski – Moranda model is the earliest best-known reliability model. The failure intensity function is

$$Y(t) = ¢ (N – I + 1)$$

where, ¢ = Constant of proportionally , N = Total number of errors present, I = Number of errors found by time interval t_i

This model assumes that all failures have the same failure rate. i.e., failure rate is a step function and there will be an improvement in reliability after fixing a fault. So, every failure contributes equally to the overall reliability.

Here, failure intensity is directly proportional to the number of remaining errors in a program. The relation between time and failure intensity is shown in fig 5.3.

The time interval t_1, t_2.....t_k may vary in duration depending upon the occurrence of a failure. Between $(i-1)^{th}$ and i^{th} failure, failure intensity function is $(N – I + 1)$ ¢.

Example. There are 50 errors estimated to be present in a program. We have experience 30 errors. Use Jelinski-Moranda model to calculate failure intensity with a given value of ¢ = 0.03. What will failure intensity after the experience of 40 errors?

Solution.

N = 50 errors

i = 30 failures

¢ = 0.03

We know

Y (t) = ¢ (N – i + 1)

= 0.03(50 – 30 + 1)

= 0.63 failures / CPU hr.

After 40 failures

Y (t) = 0.03 (50 – 40)

= 0.33 failures / CPU hr.

Hence, there is continuous decrease in the failure intensity when the number of failures increases.

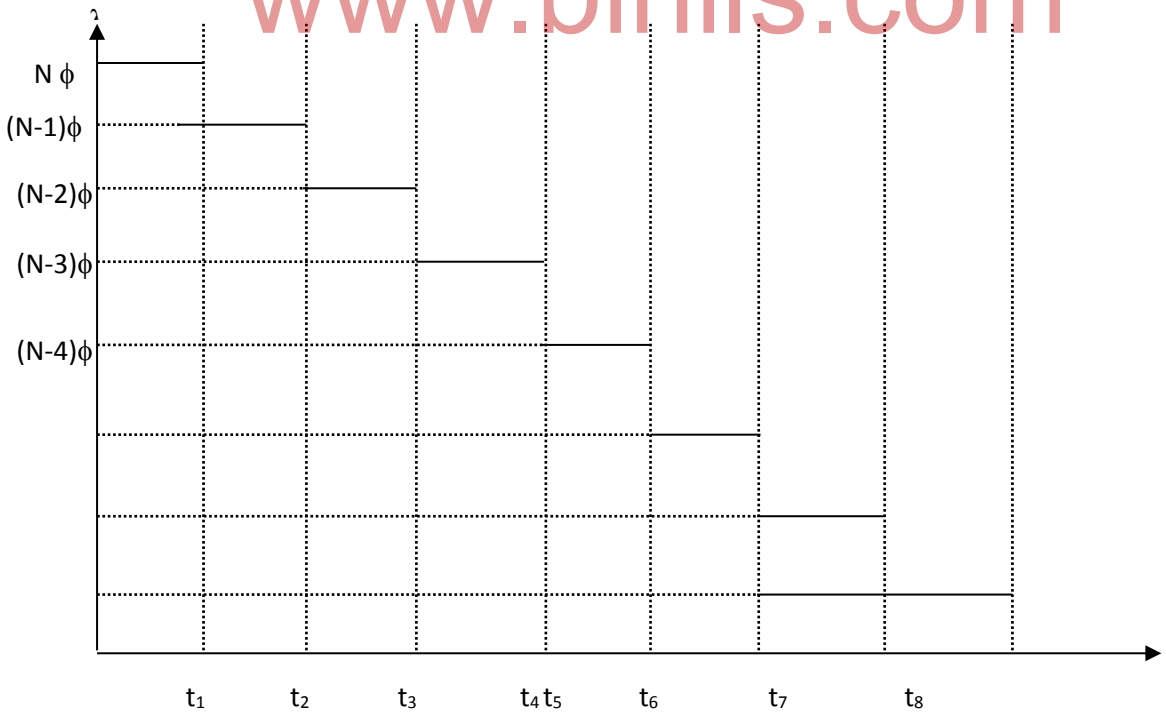


Fig No 5.2. Relation between T and Y

STEP FUNCTION MODEL

The simplest reliability growth model is a step function model. It is assumed that the reliability increases by a constant increment each time an error is detected & repaired. Such type of model is shown in fig 5.3.

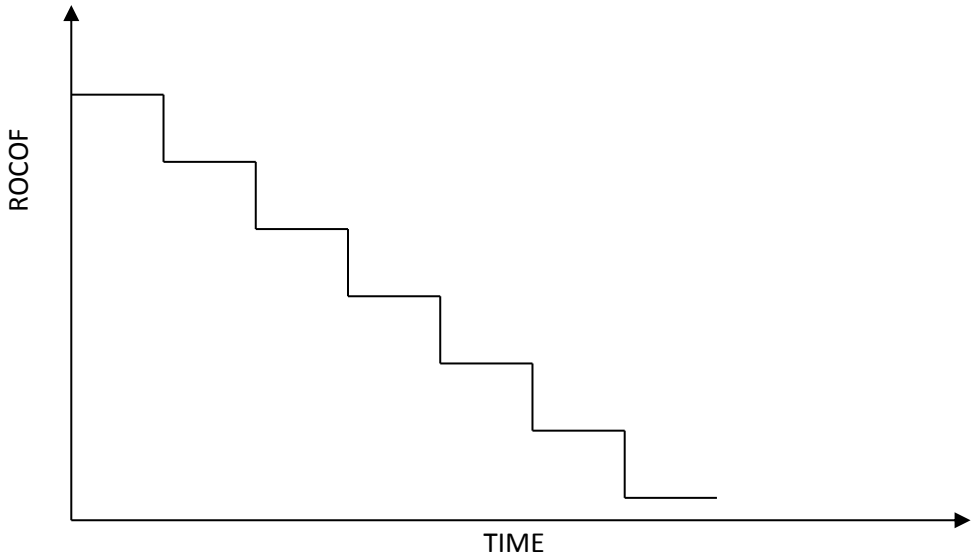


Fig No 5.3. Step function Model

However, this simple model of reliability, which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since corrections of different errors contribute differently to reliability growth.

MEASUREMENT OF RELIABILITY

Most hardware-related reliability models are predicted on failure due to. In hardware, failures due to physical wear (e.g., effect of temperature corrosion) are more likely than a design related failure. In computer-based system, a simple measure of reliability is mean time-between-failure (MTBF) and it can be expressed as:

$$MTBF = MTTF + MTTR$$

Where,

MTTF = Mean Time to failure
MTTR = Mean Time to Repair

Measurement of Availability

MTBF is a more useful measures than defects / KLOC or defects / FP. Because, an end-user is concerned with failures, not with the total error count. Because each error contained within a program does not have the same failure rate, the total error count provides little indication of the reliability of a system. In addition to reliability that a program is operating according to requirement at a given point in time and is defined as:

Availability = (MTTF ((MTTF + MTTR)) x 100

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR.

5.3 REVESRE SOFTWARE ENGINEERING

5.3.1 DEFINITION

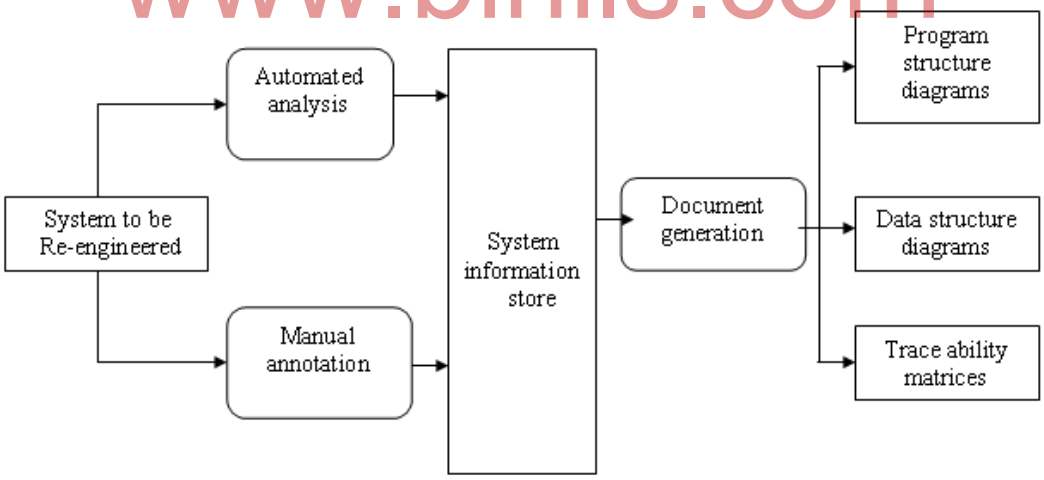
Reverse Engineering is the process followed in order to find difficult, unknown and hidden information about a software system.

5.3.2 PURPOSE OF REVERSE ENGINEERING

The main purpose of reverse engineering is to recover information from the existing code or any other intermediate documents, any activity that requires program understanding at any level may fall within the scope of reverse engineering.

5.3.3 REVERSE ENGINEERING PROCESS

The reverse engineering process is illustrated in figure 5.4.. The process starts with an analysis phase. During this phase, the system is analyzed using automated tools to discover its structure. In itself, this is not



enough to re-create the system design. Engineers then work with the system source code and its structural model. They add information to this, which they have collected by understanding the system. This information is maintained as a directed graph that is linked to the program source code. Information store browsers are used to compare the graph structure and the code and to annotate the graph with extra information. Documents of various types such as program and data structure diagrams and traceability matrices can be generated from the directed graph. Traceability matrices show where entities in the system are defined and referenced. The process of document generation is an iterative one as the design information is used to further refine the information held in the system repository.

5.3.4 CHARACTERISTICS OF REVERSE ENGINEERING

Abstraction Level. The abstraction level of a reverse engineering process refers to the sophistication of the design information that can be extracted from 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 is us for easier understanding of the program.

1. **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 a source code listing, it is relatively easy to develop a complete procedural design representations may also be derived, but it is more difficult to develop a complete set of UML diagrams .
2. **Interactivity.** 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.
3. **Directionality.** 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.
4. **Extract Abstractions.** The core of reverse engineering is an activity called extract abstractions. The engineer must evaluate the old program and from the (often un-documented) 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.

5.3.5 APPLICATIONS OF REVERSE ENGINEERING

- Program comprehension
- Re-documentation
- Recovery of design, approach and design details at any level of abstraction
- Identifying re-usable components
- Identifying components that need restructuring
- Recovery business rules.

5.4 SOFTWARE RE - ENGINEERING

5.4.1 INTRODUCTION OF RE_ENGINEERING

*Re-engineering means to re-implementing systems to make them more maintainable.*In re-engineering the functionality and system architecture remains the same but it include re-documenting, organizing and restricting, modifying and updating the system. It is a solution to the problems of system evolution.

5.4.2 PRINCIPLES OF SOFTWARE RE- ENGINEERING

The principles of re-engineering when applied to software development processes, it is called software re-engineering. It affects positively software cost, quality, service to the customer and speed of delivery. Software, whether product or system, deals with business processes making them faster, smarter and automatic in response for delivery and execution.

In software re-engineering, any one or more of the following may be resorted:

- Redefining software scope and goals.
- Redefining RDD and SRS by way of additions, deletions and extensions of functions and features.
- Redesigning the application design and architecture using new technology, upgrades, and platforms, interfacing to new technologies to make the process faster, smarter and automatic.
- Resorting to data restructuring, improving database design, code restructuring to make its size smaller and efficient in operations.
- Rewriting the documentation to more user friendly.

5.4.3 RE- ENGINEERING PROCESS

Figure 5.5 shows a re-engineering process. The input to the process is a original program and the output is a structured, modularized version of the same program. At the same time as program re-engineering, the data for system may also be re-engineered.

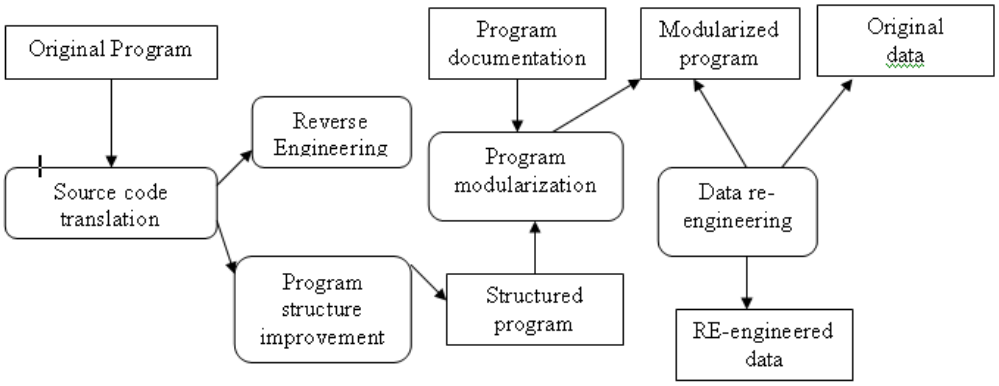


Fig No 5.5. Re-Engineering process

Re-engineering process includes the following activities:

- **Source Code Translation**: The programming language of an old program is converted into the modern version of the same language or to a new language.
- **Reverse Engineering:** The program is analyzed and important and useful information's are extracted from it which helps to document its functionality.
- **Program Structure Improvement:** Control structure of the program is analyzed and modified to make it to read and understand.

- **Program Modularization:** Redundancy of any part is removed and related parts are grouped together.
- **Data Re-engineering:** The data processed by the program is changed to reflect program changes.

5.4.3 RE- ENGINEERING PROCESS MODEL

Re-engineering of information system 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 figure 5-6. is a cyclic model.

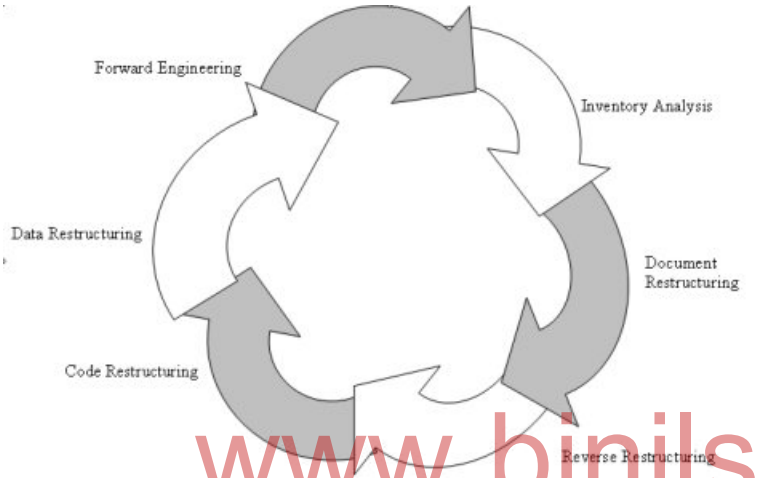


Fig No 5-6. Re-engineering Process Model

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

1. **Inventory analysis:** Every software should have an inventory of all applications. The inventory containing information that provides a detailed description (e.g., size, age business criticality) of 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.

   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.

   i. **Creating documentation is far too time consuming:**

   ii. **Documentation must be updated, but we have limited resources:-** 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**.

   iv. **Each of these options is viable.** A software organization must choose the 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 . The

"secrets" to be understanding are obscure because no specification was ever developed. Therefore, reverse engineering for software is an efforts to create a representation of the program at a higher level of abstraction. Reverse engineering tools extract data; 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 a 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 do this activity, the source code is analyzed using a restructuring tool. Violation of structured programming constructs are noted, and code is then restructured the resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

5. **Data Restructuring:** A program with weak data architecture will be difficult to adapt and enhance.Unlike code restructuring, which occurs at a relatively low levels 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. Forward engineering, also called renovation or reclamation recovers design information from existing software and the recovered information is used to alter or reconstitute the existing system to improve its overall quality.

5.4.4 DIFFERENCES BETWEEN FORWARD ENGINEERING AND RE-ENGINEERING

This difference between forward engineering ad re-engineering is shown in fig.5.7. 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 understanding and transformation of the original system.

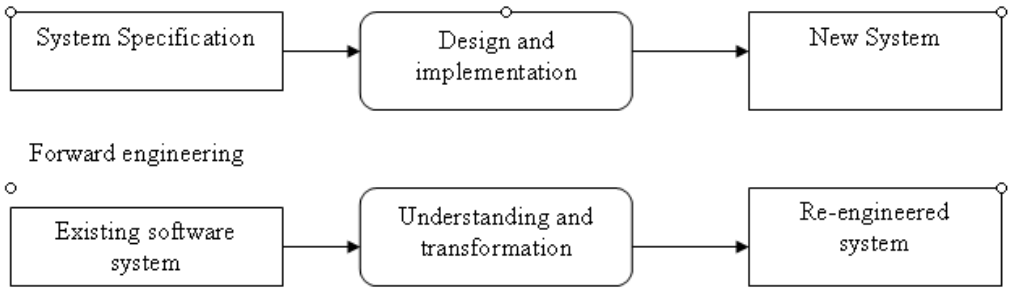


Fig No 5.7. Difference Between Forward Engineering and Re-Engineering

ADVANTAGES

- **Reduced risk:** There is a high risk in redeveloping software that is essential for an organization. Errors may be made in the system specification; there may be development problems, etc.,
- **Reduced cost:** The cost of re-engineering is significantly less than the cost of developing new software.

DISADVANTAGES

- A system can be improved by re-engineering. It isn't possible.
- Involves high addition costs.
- Although re-engineering can improve maintainability, the re-engineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

SUMMARY

- Verification is the process of determining whether the output of one phase of software development confirms to that of its previous phase.
- Validation is the process of determining whether a fully developed system confirms to its requirements specifications.
- *Software Quality Assurance is a set of activities designed to evaluate the process by which software is developed and / or maintained.*

- Software Quality is the "Conformance to explicit stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software "
- Software quality attributes are: Functionality, Reliability , Usability , Efficiency , Maintainability , Portability
- Important qualities of software produce are : Correctness , Reliability, Robustness ,Performance , Verifiability , Reparability , Evolvability , Understandability , Interoperability , Productivity , Timeliness and Visibility
- SEI-CMM stands for Software Engineering Institute Capability Maturity Model. This model helps in judging the maturity of software process of an organization.
- The ISO 9000 standard specifies quality assurance elements in generic terms, which can be applied to any business regards of the product or services being offered
- ISO 9000 standards were basically designed to audit manufacturing / service organizations whereas CMM was developed specifically for software industry and this addresses several issues specific to software industry.
- ISO 9126 standard was published in 1991 to tackle the question of the definition of software quality.

- Reliability of a software product can also be defined as the probability of the product working correctly over a given period.
- **System Failure:** An event that occurs at some point in time when the system does not deliver a service as expected by its users.
- **System Error:** Erroneous system behavior where the behavior where the behavior of the system does not conform to its specification.
- **System Fault:** An incorrect system state, i.e. a system state that is unexpected by the designers of the system.
- The MTTF is the mean time for which a component is expected to be operational. The MTTF is the average time between observed system failures.
- MTTR is the average time to replace a defective component.
- **ROCOF (Rate of Occurrences of Failure).** ROCOF is the frequency of occurrence with which unexpected behavior is likely to occur
- A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired.
- Reverse Engineering is the process followed to find difficult, unknown and hidden information about a software system.
- *Re-engineering means to re-implementing systems to make them more maintainable.*

REVIEW QUESTIONS

PART - A (2 Marks Questions)

1. Differentiate Verification with validation.
2. What is SQA? What is its use?
3. What is internal quality? What is external quality?
4. What are the steps involved in the process of SQA?
5. Define the term: “Interoperability”
6. Expand the term SEI - CMM.
7. What is ISO 9000? What is ISO 9126?
8. Define “System Failure“ and “ System Fault”
9. What is software reliability?
10. State the use of reliability metric.
11. What is reliability growth model? What is its use?
12. Expand (i) POFOD and (ii) ROCOF
13. What is reverse Engineering? Wrote down its purpose.
14. What is re-engineering?

PART - B (3 Marks Questions)

1. State any six objectives of SQA.
2. Give the outline structure of SQA plan.
3. What are software quality attributes?
4. State any three needs for ISO certification.
5. What are the six characteristics of ISO 9126?
6. State any three reasons for software retirement.
7. What are the reliability terminologies? Define them.
8. Explain about “Classification of failures”
9. Define MTTF, MTTR and MTBF.
10. Define POFOD, ROCOF and AVAIL
11. What are the applications of Reverse Engineering?
12. What are the characteristics of Reverse Engineering?
13. State the difference between forward engineering and Reverse Engineering
14. What are the advantages and disadvantages of Re-engineering.

PART - C (5/10 Marks Questions)

1. What are the contents of SQA Plan?
2. How the software qualities are classified? Explain.
3. List and explain important qualities of software product.
4. List down and explain the five levels of SEI- CMM Model.
5. List down the benefits and limitations of ISO 9000 certification.
6. Explain the procedure for getting ISO 9000 certification.
7. Compare ISO 9000 certification with SEI- CMM level.
8. Explain about reliability metrics?
9. Explain any one type of reliability growth modelling.
10. Explain re - engineering process with a neat diagram
11. Explain re - engineering process model with a neat diagram
12. Explain reverse engineering tasks.