DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS

CS308 SOFTWARE ENGINEERING AND PROJECT MANAGEMENT

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

CO1	To Analyze a problem, define and identify the computing requirements appropriate to its solution using software life cycle models.
CO2	To understand various process models and identify phases of software development.
CO3	To apply the planning phase and translate a requirement specification to a design using an appropriate software engineering methodology.
CO4	To demonstrate various coding standards and appropriate testing strategy for the given software system.
CO5	To apply different maintenance process and risk management activities.
CO6	To develop software projects based on current technology by managing resources economically and keeping ethical values.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PO 1	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**
CO1	3	3	3	-	-	3	-	-	2	2	-	3
CO2	2	3	-	-	-	-	-	-	3	-	-	-
CO3	-	-	3	3	3	3	-	-	-	-	3	-
CO4	3	-	3	-	3	-	-	-	3	-	-	-
CO5	-	-	-	3	3	-	-	-	-	-	2	-
CO6	-	-	3	3	3	3	-	-	-	3	3	3

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

MAPPING OF COURSE OUTCOMES WITH PROGRAM SPECIFIC OUTCOMES

	PSO 1	PSO 2	PSO 3
CO1	3	-	-
CO2	3	-	-
CO3	-	3	-
CO4	-	-	3
CO5	3	-	-
CO6	-	-	3

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

SYLLABUS

Course code	Course Name	L-T-P-Credits	Year of Introduction
CS308	Software Engineering and Project Management	3-0-0-3	2016

Pre-requisite: Nil

Course Objectives

- To introduce the fundamental concepts of software engineering.
- To build an understanding on various phases of software development.
- To introduce various software process models.

Syllabus

Introduction to software engineering, Software process models, Software development phases, Requirement analysis, Planning, Design, Coding, Testing, Maintenance.

Expected Outcome

The students will be able to

i. Identify suitable life cycle models to be used.
ii. Analyze a problem and identify and define the computing requirements to the problem.
iii. Translate a requirement specification to a design using an appropriate software engineering methodology.
iv. Formulate appropriate testing strategy for the given software system.
v. Develop software projects based on current technology, by managing resources economically and keeping ethical values.

References

1. Ian Sommerville, Software Engineering, University of Lancaster, Pearson Education, Seventh edition, 2004.
2. K. K.Aggarwal and Yogesh Singh, Software Engineering, New age International Publishers, Second edition, 2005.
3. Roger S. Pressman, Software Engineering : A practitioner's approach, McGraw Hill publication, Eighth edition, 2014
4. S.A. Kelkar, Software Project Management: A concise study, PHI, Third edition, 2012.
5. Walker Royce, Software Project Management : A unified frame work, Pearson Education, 1998

COURSE PLAN

Module	Contents	Hours	End Sem. Exam Marks
I	Introduction to software engineering- scope of software	07	15%

	engineering – historical aspects, economic aspects, maintenance aspects, specification and design aspects, team programming aspects. Software engineering a layered technology – processes, methods and tools. Software process models – prototyping models, incremental models, spiral model, waterfall model.		
II	Process Framework Models: Capability maturity model (CMM), ISO 9000. Phases in Software development – requirement analysis- requirements elicitation for software, analysis principles, software prototyping, specification.	06	15%
	FIRST INTERNAL EXAM		
III	Planning phase – project planning objective, software scope, empirical estimation models- COCOMO, single variable model, staffing and personal planning. Design phase – design process, principles, concepts, effective modular design, top down, bottom up strategies, stepwise refinement.	07	15%
IV	Coding – programming practice, verification, size measures, complexity analysis, coding standards. Testing – fundamentals, white box testing, control structure testing, black box testing, basis path testing, code walk-throughs and inspection, testing strategies-Issues, Unit testing, integration testing, Validation testing, System testing.	07	15%
	SECOND INTERNAL EXAM		
V	Maintenance-Overview of maintenance process, types of maintenance. Risk management: software risks - risk identification-risk monitoring and management. Project Management concept: People – Product-Process-Project.	07	20%
VI	Project scheduling and tracking: Basic concepts-relation between people and effort-defining task set for the software project-selecting software engineering task Software configuration management: Basics and standards User interface design - rules. Computer aided software engineering tools - CASE building blocks, taxonomy of CASE tools, integrated CASE environment.	08	20%
	END SEMESTER EXAM		

Question Paper Pattern

1. There will be *five* parts in the question paper – A, B, C, D, E
2. Part A
   a. Total marks : 12
   b. *Four* questions each having 3 marks, uniformly covering modules I and II;

All four questions have to be answered.

3. Part B
   a. Total marks : 18
   b. *Three* questions each having 9 marks, uniformly covering modules I and II; *Two* questions have to be answered. Each question can have a maximum of three subparts.
4. Part C
   a. Total marks : 12
   b. *Four* questions each having 3 marks, uniformly covering modules III and IV; All four questions have to be answered.
5. Part D
   a. Total marks : 18
   b. *Three* questions each having 9 marks, uniformly covering modules III and IV; *Two* questions have to be answered. Each question can have a maximum of three subparts
6. Part E
   a. Total Marks: 40
   b. Six questions each carrying 10 marks, uniformly covering modules V and VI; four questions have to be answered.
   c. A question can have a maximum of three sub-parts.

7. There should be at least 60% analytical/numerical questions.

•

QUESTION BANK

MODULE I				
Q:NO:	**QUESTIONS**	**CO**	**KL**	**PAGE NO:**
1	Explain the concept of software engineering?	CO1	K2	17
2	Differentiate between waterfall model and spiral model.	CO1	K4	31,36
3	Describe the incremental model of software development.	CO1	K2	35
4	What is meant by prototyping model?	CO1	K1	33
5	Explain the layered technology of software engineering.	CO1	K2	26
6	How does the maintenance aspect of software engineering changed over the period of time?	CO1	K3	23
7	Why the historical aspects still faces issues in scope of software engineering?	CO1	K3	22
8	Briefly describe the scope of software engineering.	CO1	K2	22
9	What are basic characteristics of good software?	CO1	K2	19
10	Illustrate the need of software engineering in the society.	CO1	K3	19

•

MODULE II				
1	Briefly explain the process framework in software engineering?	CO2	K6	38
2	Briefly describe software prototyping.	CO2	K2	57
3	List out the different analysis principles involved in elicitation.	CO2	K4	56
4	Explain the concept of ethnography.	CO2	K2	55
5	What makes use case diagram a representation for elicitation?	CO2	K3	54
6	Write short notes on process activities in requirement elicitation and analysis.	CO2	K6	49
7	Describe the phases of Software Development Life Cycle.	CO2	K2	45
8	Explain the ISO 9000 standard.	CO2	K6	43
9	How the Capability Maturity Model helps large organization?	CO2	K3	40
10	Illustrate the framework models mainly used.	CO2	K3	38

•

MODULE III				
1	Describe about detailed COCOMO model.	CO3	K2	67
2	Differentiate between top down and bottom up approaches.	CO3	K4	90,91
3	Write note on the concept of Effective Modular Design.	CO3	K6	86
4	Explain the quality attributes in design.	CO3	K5	78
5	Differentiate between basic and intermediate COCOMO model.	CO3	K4	68,69
6	Write note on different design concepts.	CO3	K6	80
7	Write notes on resources.	CO3	K3	66
8	Explain about staffing and project planning.	CO3	K5	73
9	Write notes on the characteristics and guidelines for a good design.	CO3	K2	77
10	Explain the staffing activities for a software project.	CO3	K2	75

•

MODULE IV				
1	Write notes on (a) Memory Leaks, (b) NULL dereferencing and (c) Synchronization errors.	CO4	K6	95,96
2	Briefly describe about testing objectives and testing principles.	CO4	K2	103
3	Describe briefly about integration testing along with its various strategies.	CO4	K2	113
4	Write note on system testing.	CO4	K3	117
5	Explain any five programming practice methods	CO4	K5	98
6	Explain alpha-beta testing in detail.	CO4	K5	116
7	Write short note on coding standards.	CO4	K3	101
8	Write notes on programming practices.	CO4	K6	98
9	Explain briefly about common coding errors.	CO4	K2	94
10	Explain the different techniques involved in black-box testing	CO4	K5	104
11	Explain black box testing listing its advantages and disadvantages.	CO4	K5	103

•

12	Explain about white box testing.	CO4	K5	109
	MODULE V			
1	Point out the different types of maintenance.	CO5	K4	121
2	Write note on project concept in project management.	CO5	K6	142
3	Briefly describe the five part common sense approach to software project.	CO5	K2	144
4	Explain the importance of people in 4P's of project management.	CO5	K5	133
5	Write note on risk mitigation, monitoring and management.	CO5	K3	130
6	Explain the concept of melding product and process.	CO5	K2	139
7	Write note on process decomposition.	CO5	K3	141
8	Differentiate between the concept of product and process.	CO5	K4	137, 139
9	Explain the concept of people in project management.	CO5	K2	133
10	Point out the categories of players in project management.	CO5	K4	133

•

11	Explain about the different maintenance activities.	CO5	K2	121
12	Explain different categories of software risks.	CO5	K2	125
	MODULE VI			
1	Point out the different integration and testing tools	CO6	K4	163
2	Point out the root causes of late delivery of software products.	CO6	K4	145
3	Briefly describe the building blocks of CASE.	CO6	K2	158
4	Explain the golden rules of user interface design.	CO6	K5	153
5	Write note on (a) relationship between people and effort and (b) defining a task set for software project.	CO6	K3	147, 148
6	Explain briefly about taxonomy of CASE tools.	CO6	K5	159
7	Describe the basic principles that guide software project scheduling	CO6	K2	145

•

8	Write notes on (a) Software configuration management tools (b) reengineering tools.	CO6	K3	162, 164
9	Explain the concept of make the interface inconsistent.	CO6	K2	156
10	Write note on CASE tools	CO6	K6	157
11	Define SCM with its elements.	CO6	K1	151
12	Differentiate between the concepts place the user in control and reduce the user's memory load.	CO6	K4	154, 155

•

APPENDIX 1		
CONTENT BEYOND THE SYLLABUS		
S:NO;	**TOPIC**	**PAGE NO:**
1	Top ten automated software testing tools	166

.

MODULE 1

1.1 INRTODUCTION

- **Software** is more than just a program code. A program is an executable code, which serves some computational purpose.

- Software is considered to be a 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.

- So, we can define ***software engineering*** as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures.

- The outcome of software engineering is an efficient and reliable software product.

- IEEE defines software engineering as:

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

- Without using software engineering principles it would be difficult to develop large programs.

- In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes.

- Software engineering helps to reduce this programming complexity. Software engineering principles use two important techniques to reduce problem complexity: ***abstraction*** and ***decomposition*****.**

- The principle of abstraction implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose.

- Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem.

- The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one.

- However, in this technique any random decomposition of a problem into smaller parts will not help.

- The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution.

- A good decomposition of a problem should minimize interactions among various components.

- If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

•

1.1.1 Need of 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 process 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-** The always growing and adapting nature of software hugely depends upon the environment in which the 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 Characteristics of Good Software

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

- Operational

•

- Transitional
- Maintenance

Operational

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

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

Transitional

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

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

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

- Modularity
- Maintainability

•

- Flexibility
- Scalability

1.1.3Three parties are involved in software engineering:

i. **Client:** The client is the individual who wants a product to be built (developed).

ii. **Developers:** The developers are the members of a team responsible for building that product.

iii. **User:** The user is the person or persons on whose behalf the client has commissioned the product and who will utilize the software.

1.1.4Types of software:

(a) Based on parties relationship:

i. **Internal Software:** Both the client and developers may be part of the same organization.

ii. **Contract Software:** The client and developers are members of totally independent organizations.

•

(b) Based on the functionality:

i. **Custom software:** It is written for one client.

ii. **Commercial off-the-shelf (COTS) software:** It has multiple copies and the copies are sold at much lower prices to a large number of buyers. It is developed for "the market".

iii. **Open-source software:** It is developed and maintained by a team of volunteers and may be downloaded and used free of charge by anyone.

1.2 SCOPE OF SOFTWARE ENGINEERING:

The scope of software engineering is extremely broad. In general, five aspects are involved:

- Historical Aspects
- Economic Aspects
- Maintenance Aspects
- Requirements, Analysis, and Design Aspects
- Team Development Aspects

1.2.1 Historical aspects:

- Software engineering cannot be considered as engineered since an unacceptably large proportion of software products still are being:

  - Delivered late
  - Over budget
  - With residual faults.

•

- Solution: A software engineer has to acquire a broad range of skills, both technical and managerial. These skills have to be applied to: Programming; and Every step of software production, from requirements to post_delivery maintenance.

1.2.2 Economic Aspects:

- Applying economic principles to software engineering requires the client to choose techniques that reduce long-term costs in terms of the economic sense.

- The cost of introducing new technology into an organization includes:
  - Training cost
  - A steep learning curve
  - Unable to do productive work when attending the class.

1.2.3 Maintenance Aspects:

- **Classical View of Maintenance:**
  Development-then-maintenance model.

- But this model is unrealistic due to:

  - During the development, the client's requirements may change. This leads to the changes in the specification and design.
  - Developers try to reuse parts of existing software products in the software product to be constructed.

- **Modern view of Maintenance**: It is the process that occurs when "software undergoes modifications to code and associated documentation due to a problem or the need for improvement or adaptation".

•

- That is, maintenance occurs whenever a fault is fixed or the requirements change, irrespective of whether this takes place before or after installation of the product,

- **Classical Postdelivery Maintenance:** All changes to the product once the product has been delivered and installed on the client's computer and passes its acceptance test.

- **Modern Maintenance (or just maintenance):** Corrective, perfective, or adaptive activities performed at any time.

- Classical postdelivery maintenance is asubset of modern maintenance.

- A major aspect of software engineering consists of techniques, tools, and practices that lead to a reduction in postdelivery maintenance cost.

- The importance of Postdelivery Maintenance:

  - ❖ A software product is a model of the real world, and the real world is perpetually changing.
  - ❖ As a consequence, software has to be maintained constantly for it to remain an accurate reflection of the real world.

1.2.4 Team Development Aspects:

- Team development leads to interface problems among code components and communication problems among team members.

•

- Unless the team is properly organized, an inordinate amount of time can be wasted in conferences between team members.

- It also includes human aspects, such as team organization, economic aspects, and legal aspects, such as copyright law.

1.2.5 Requirements, Analysis and Design Aspects:

- The earlier we correct a fault, the better. That is, the cost of correcting a fault increases steeply since it is directly related to what has to be done to correct a fault.

- If the mistake is made while eliciting the requirements, the resulting fault will probably also appear in the specifications, the design, and the code.

- It is crucial to check that making the change has not created a new problem elsewhere in the product. All the relevant documentation, including manuals, needs to be updated.

- The corrected product must be delivered and reinstalled.

•

1.3 SOFTWARE ENGINEERING: A LAYERED TECHNOLOGY

- The software engineering can be divided into 4 layers:

Fig: Layered technology of Software Engineering

1. A quality Process :

- Any engineering approach must rest on an quality.
- The "Bed Rock" that supports software Engineering is QualityFocus.

2. Process :

- Foundation for Software Engineering is the Process Layer
- Software Engineering process is the GLUE that holds all the technology layers together and enables the timely development of computer software.
- It forms the base for management control of software project.

•

3. Methods:

- Software Engineering methods provide the "Technical Questions" for building Software.
- Methods contain a broad array of tasks that include communication requirement analysis, design modeling, program construction testing and support.

4. Tools:

- Software engineering tools provide automated or semi-automated support for the "Process" and the "Methods".
- Tools are integrated so that information created by one tool can be used by another

.

1.4 SOFTWARE PROCESS MODEL

- A software process or software methodology is a set of related activities that leads to the production of the software.
- These activities may involve the development of the software from the scratch, or, modifying an existing system.
- Any software process must include the following four activities:

  (a) Software specification (or requirements engineering): Define the main functionalities of the software and the constrains around them.

  (b) S**oftware design and implementation**: The software is to be designed and programmed.

•

(c) Software verification and validation: The software must conforms to it's specification and meets the customer needs.

(d) Software evolution (software maintenance): The software is being mo dified to meet customer and market requirements changes.

- A process also includes the process description, which includes:

(a) Products: The outcomes of the an activity. For example, the outcome of architectural design maybe a model for the software architecture.

(b) Roles: The responsibilities of the people involved in the process. For example, the project manager, programmer, etc.

(c) Pre and post conditions: The conditions that must be true before and after an activity. For example, the pre condition of the architectural design is the requirements have been approved by the customer, while the post condition is the diagrams describing the architectural have been reviewed.

- Software process is complex, it relies on making decisions.
- There's no ideal process and most organizations have developed their own software process.
- For example, an organization works on critical systems has a very structured process, while with business systems, with rapidly changing requirements, a less formal, flexible process is likely to be more effective.

- A software process model is a simplified representation of a software process.
- Each model represents a process from a specific perspective.
- These generic models are abstractions of the process that can be used to explain different approaches to the software development.

•

- Some methodologies are sometimes known as ***software development life cycle***(SDLC) methodologies, though this term could also be used more generally to refer to any methodology.

1.4.1 Software Development Life Cycle

- SDLC is a process followed for a software project, within a software organization.
- It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software.
- The life cycle defines a methodology for improving the quality of software and the overall development process.

•

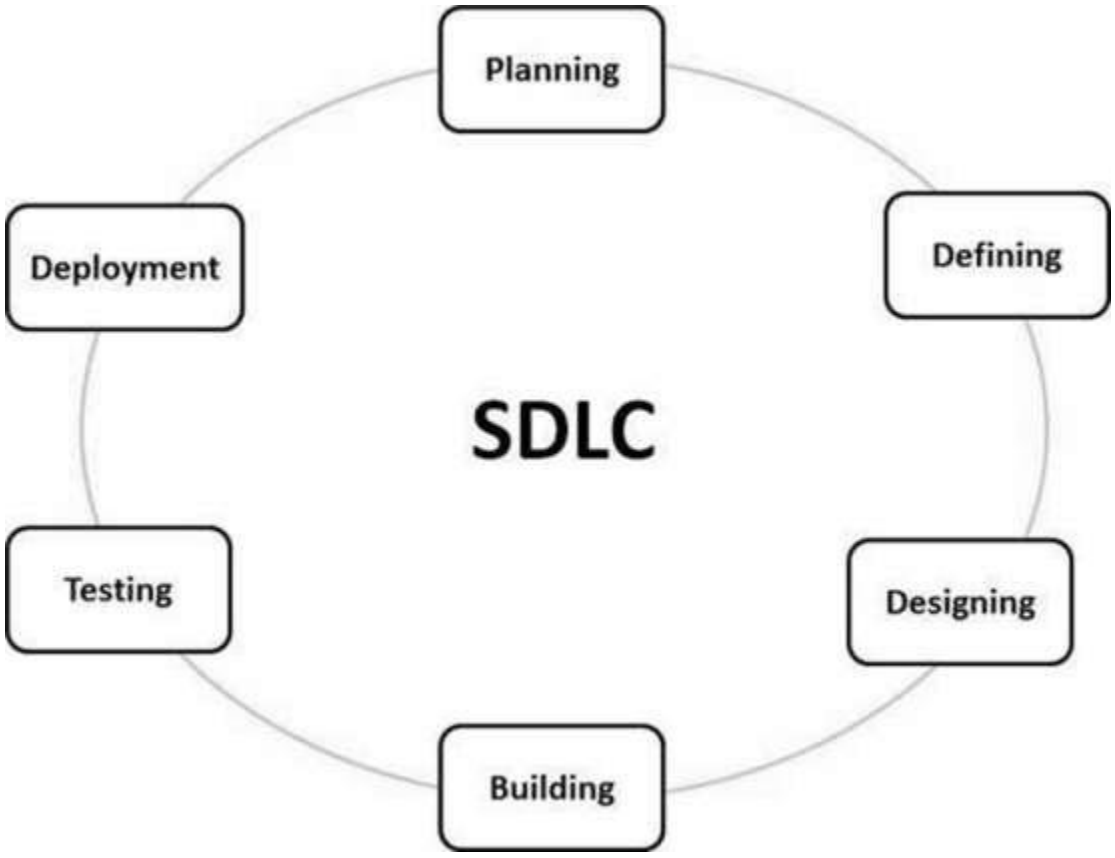


Fig: Software Development Life Cycle

(a) Planning and Requirement Analysis:

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

(b) Defining Requirements:

•

The next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through an **SRS (Software Requirement Specification)** document which consists of all the product requirements to be designed and developed during the project life cycle.

(c) Designing the Product Architecture:

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

(d) Building or Developing Product:

The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

(e) Testing the product:

This stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

(f) Deployment in the Market and Maintenance:

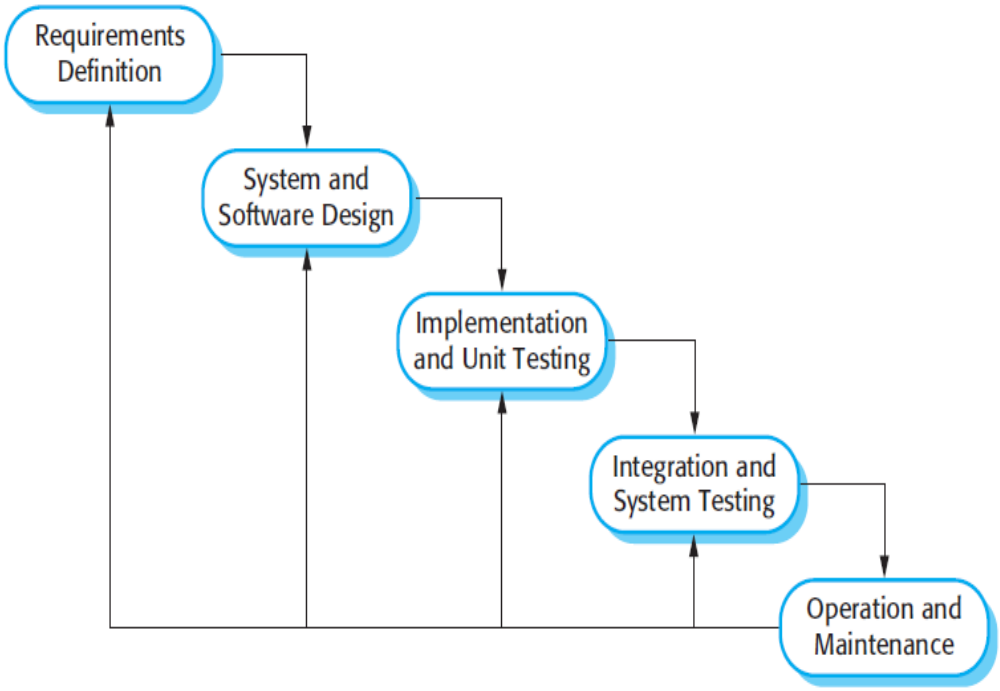
Once the product is tested and ready to be deployed it is released formally in the appropriate market. After the product is released in the market, its maintenance is done for the existing customer base.

1.4.2 Software Process Model or SDLC Model

(a) Waterfall model:

- The waterfall model is a sequential approach, where each fundamental activity of a process represented as a separate phase, arranged in linear order.

- In the waterfall model, you must plan and schedule all of the activities before starting working on them (plan-driven process).
- Plan-driven process is a process where all the activities are planned first, and the progress is measured against the plan.
- The phases of the waterfall model are: Requirements, Design, Implementation, Testing and Maintenance.



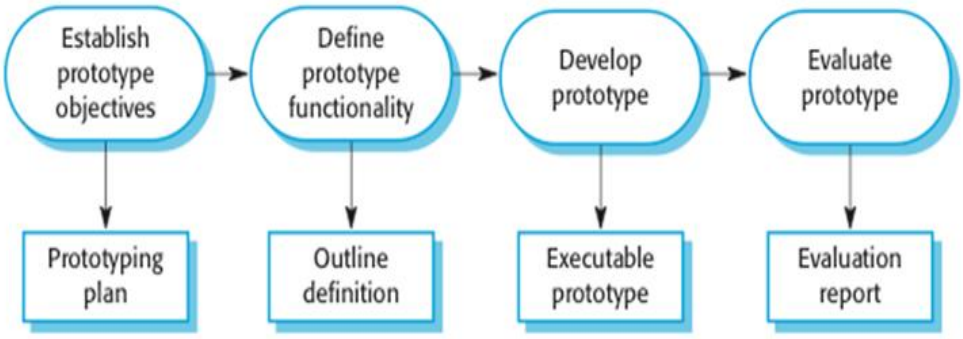
- In principle, the waterfall model should only be applied when requirements are well understood.
- And also unlikely to change radically during development as this model has a relatively rigid structure which makes it relatively hard to accommodate change when the process in underway.
- The software process therefore is not a simple linear but involves feedback from one phase to another.

•

- So, documents produced in each phase may then have to be modified to reflect the changes made.
- In principle, the waterfall model should only be applied when requi rements are well understood.
- And also unlikely to change radically during development as this model has a relatively rigid structure which makes it relatively hard to accommodate change when the process in underway.

(b) Prototyping Model:

- A prototype is a version of a system or part of the system that's developed quickly to check the customer's requirements or feasibility of some design decisions.
- So, a prototype is useful when a customer or developer is not sure of the requirements, or of algorithms, efficiency, business rules, response time, etc.
- In prototyping, the client is involved throughout the development process, which increases the likelihood of client acceptance of the final implementation.
- A software prototype can be used:
- **[1]** In the **requirements engineering**, a prototype can help with the elicitation and validation of system requirements.
- **[2]** In the **system design**, a prototype can help to carry out design experiments to check the feasibility of a proposed design.
- For example, a database design may be prototype-d and tested to check it supports efficient data access for the most common user queries.



- The phases of a prototype are:

(i) Establish objectives: The objectives of the prototype should be made explicit from the start of the process. Is it to validate system requirements, or demonstrate feasibility, etc.

(ii) Define prototype functionality: Decide what are the inputs and the expected output from a prototype. To reduce the prototyping costs and accelerate the delivery schedule, you may ignore some functionality, such as response time and memory utilization unless they are relevant to the objective of the prototype.

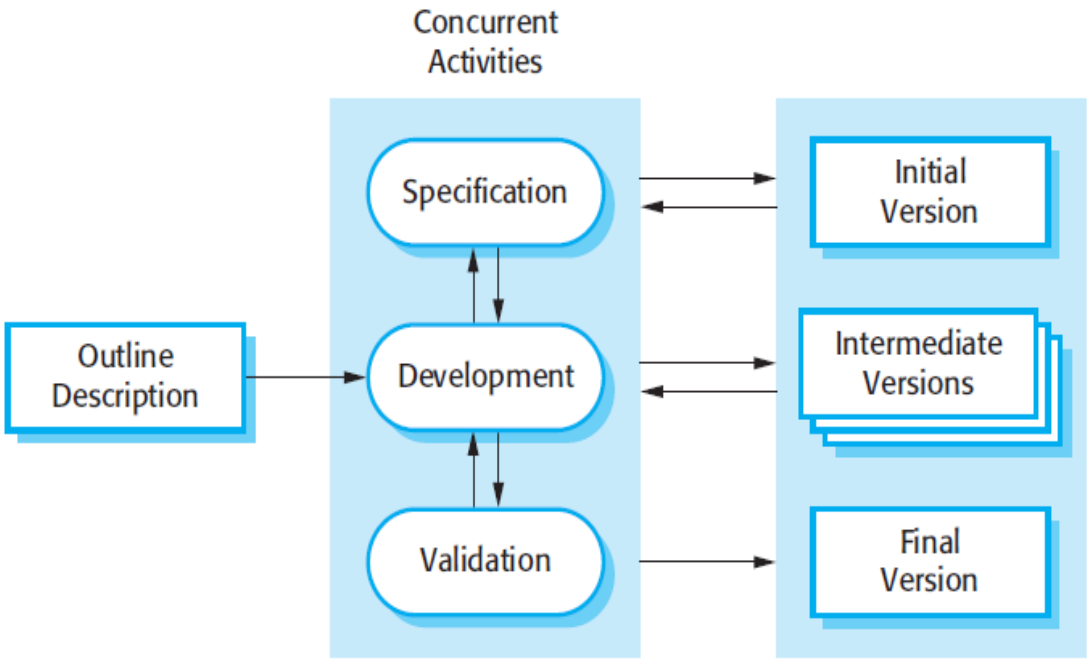
(iii)Develop the prototype: The initial prototype is developed that includes only user interfaces.

(iv) Evaluate the prototype: Once the users are trained to use the prototype, they then discover requirements errors. Using the feedback both the specifications and the prototype can be improved. If changes are introduced, then a repeat of steps 3 and 4 may be needed.

•

(c) Incremental Development Model:

- Incremental development is based on the idea of developing an initial implementation, exposing this to user feedback, and evolving it through several versions until an acceptable system ha s been developed.
- The activities of a process are not separated but interleaved with feedback involved across those activities.

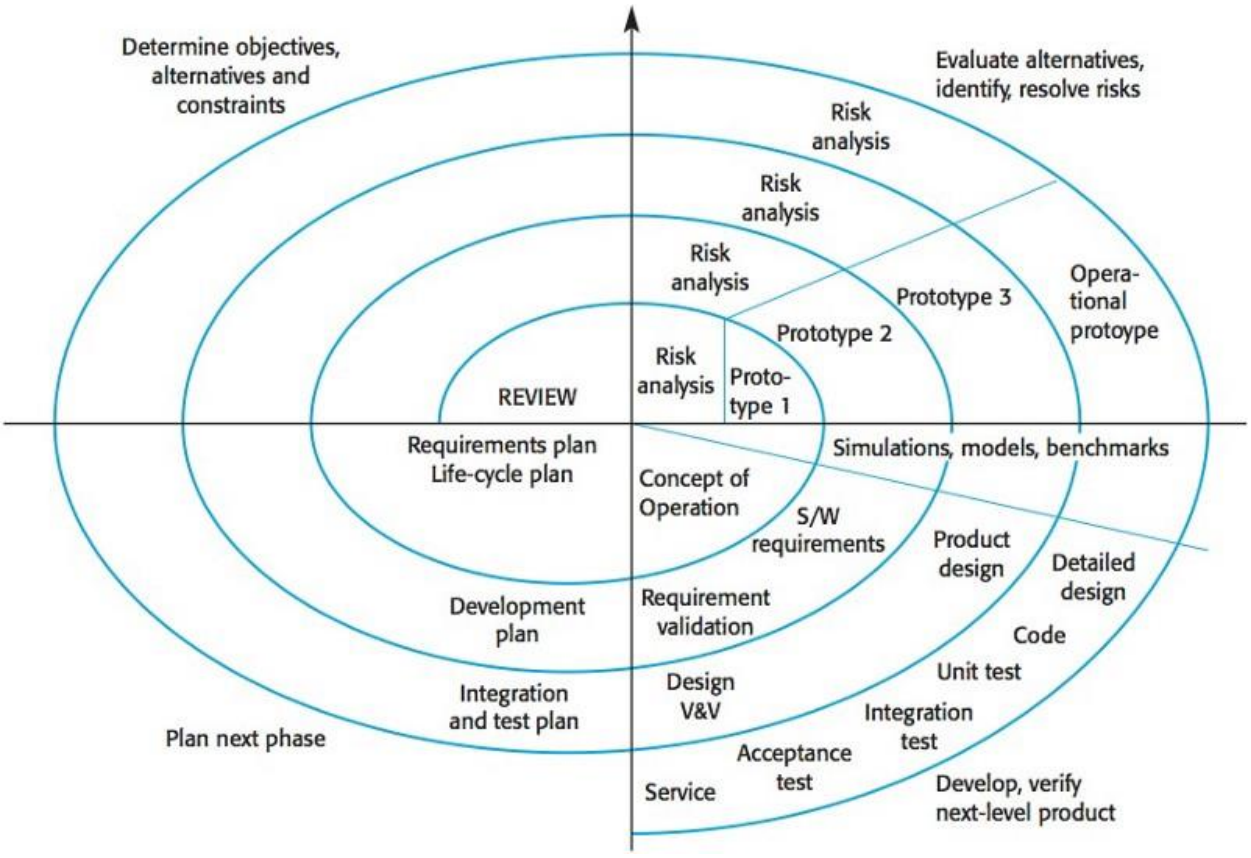


- Each system increment reflects a piece of the functionality that is needed by the customer.
- Generally, the early increments of the system should include the most important or most urgently required functionality.
- This means that the customer can evaluate the system at early stage in the development to see if it delivers what's required.
- If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

•

(d) Spiral Model:

- The spiral model is a risk-driven where the process is represented as spiral rather than a sequence of activities.
- It was designed to include the best features from the waterfall and prototyping models, and introduces a new component; risk-assessment.
- Each loop in the spiral represents a phase.
- Thus the first loop might be concerned with system feasibility, the next loop might be concerned with the requirements definition, the next loop with system design, and so on.



- Each loop in the spiral is split into four sectors:

•

(a) Objective setting: The objectives and risks for that phase of the project are defined.

(b) Risk assessment and reduction: For each of the identified project risks, a detailed analysis is conducted, and steps are taken to reduce the risk. For example, if there's a risk that the requirements are inappropriate, a prototype may be developed.

(c) Development and validation: After risk evaluation, a process model for the system is chosen. So if the risk is expected in the user interface then we must prototype the user interface. If the risk is in the development process itself then use the waterfall model.

(d) Planning: The project is reviewed and a decision is made whether to continue with a further loop or not.

- Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development.
- In practice, however, the model is rarely used.

.

MODULE 2

2.1 PROCESS FRAMEWORK

- Software process models can be prescriptive or agile, complex or simple, all-encompassing or targeted, but in every case, five key activities must occur.

- The framework activities are applicable to all projects and all application domains, and they are a template for every process model.

- Each framework activity is populated by a set of software engineering actions – a collection of related tasks that produces a major software engineering work product.

- Each action is populated with individual work tasks that accomplish some part of the work implied by the action.

- The following *generic process framework* is applicable to the vast majority of software projects.

a) **Communication**: involves heavy communication with the customer (and other stakeholders) and encompasses requirements gathering.

b) **Planning**: Describes the technical tasks to be conducted, the risks that are likely, resources that will be required, the work products to be produced and a work schedule.

c) **Modeling**: encompasses the creation of models that allow the developer and customer to better understand software requirement and the design that will achieve those requirement.

d) **Construction**: combines code generation and the testing required uncovering errors in the code.

e) **Deployment**: deliver the product to the customer who evaluates the delivered product and provides feedback.

- Each software engineering action is represented by a number of different task sets – each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.
- The task set that best accommodates the needs of the project and the characteristics of the team is chosen.

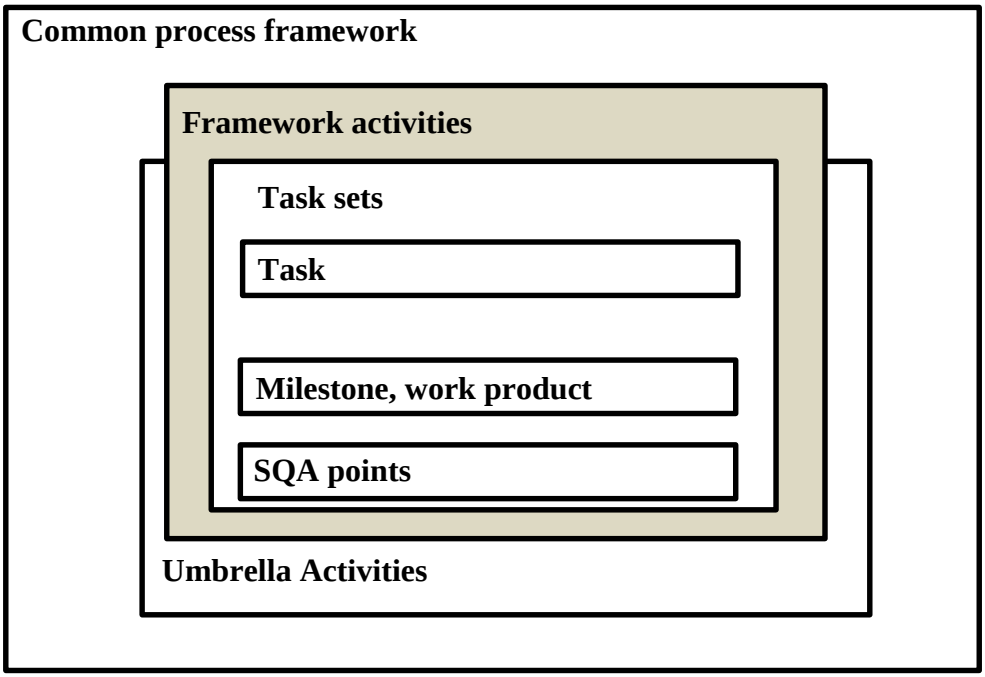


Fig: Process Framework

- The framework described in the generic view of software engineering is complemented by a number of *umbrella activities*. Typical activities include:

•

a) **Software project tracking and control**: allows the team to assess progress against the project plan and take necessary action to maintain schedule.

b) **Risk Management**: Assesses the risks that may affect the outcome of the project or the quality.

c) **Software quality assurance**: defines and conducts the activities required to ensure software quality.

d) **Formal Technical Review**: uncover and remove errors before they propagate to the next action.

e) **Measurement**: defines and collects process, project, and product measures that assist the team in delivering software that meets customers' needs.

f) **Software configuration management**: Manages the effect of change throughout the software process.

g) **Reusability management**: defines criteria for work product reuse.

h) **Work product preparation and production**: encompasses the activities required to create work products such as models, documents, etc.

2.1.1 Capability Maturity Model (CMM):

- SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

- SEI CMM can be used two ways: capability evaluation and software process assessment.

- Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software process capability of an organization.

- The results of capability evaluation indicates the likely contractor performance if the contractor is awarded a work. Therefore, the results of software process capability assessment can be used to select a contractor.

- On the other hand, software process assessment is used by an organization with the objective to improve its process capability. Thus, this type of assessment is for purely internal use.

- SEI CMM classifies software development industries into the following five maturity levels.

- The different levels of SEI CMM have been designed so that it is easy for an organization to slowly build its quality system starting from scratch.

a) **Level 1: Initial**. A software development organization at this level is characterized by ad hoc activities. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic. Therefore, it is also called chaotic level. The success of projects depends on individual efforts and heroics.

b) **Level 2: Repeatable.** At this level, the basic project management practices such as tracking cost and schedule are established. Size and cost estimation techniques like function point analysis, COCOMO, etc. are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications.

•

c) **Level 3: Defined**. At this level the processes for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles, and responsibilities. The processes though defined, the process and product qualities are not measured.

d) **Level 4: Managed.** At this level, the focus is on software metrics. Two types of metrics 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 average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc. Quantitative quality goals are set for the products. The software process and product quality are measured and quantitative quality requirements for the product are met.

e) **Level 5: Optimizing.** At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement.

Applicability of SEI CMM to organizations

- Highly systematic and measured approach to software development suits large organizations dealing with negotiated software, safety-critical software, etc. For those large organizations, SEI CMM model is perfectly applicable.

- But small organizations typically handle applications such as Internet, e-commerce, and are without an established product range, revenue base, and

•

experience on past projects, etc. For such organizations, a CMM-based appraisal is probably excessive.

- These organizations need to operate more efficiently at the lower levels of maturity. For example, they need to practice effective project management, reviews, configuration management, etc.

2.1.2 ISO 9000:

- ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987.

- ISO certification serves as a reference for contract between independent parties. The ISO 9000 standard specifies the guidelines for maintaining a quality system.

- The ISO standard mainly addresses operational aspects and organizational aspects such as responsibilities, reporting, etc.

Types of ISO 9000 quality standards

- ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003.

- The ISO 9000 series of standards is based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically.

•

- The types of industries to which the different ISO standards apply are as follows:

  a) ISO 9001 applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organizations.

  b) ISO 9002 applies to those organizations which do not design products but are only involved in production. Examples of these category industries include steel and car manufacturing industries that buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organizations.

  c) ISO 9003 applies to organizations that are involved only in installation and testing of the products.

Need for obtaining ISO 9000 certification

a) Confidence of customers in an organization increases when organization qualifies for ISO certification.

b) ISO 9000 requires a well-documented software production process to be in place. A well-documented software production process contributes to repeatable and higher quality of the developed software.

c) ISO 9000 makes the development process focused, efficient, and costeffective.

d) ISO 9000 certification points out the weak points of an organization and recommends remedial action.

•

e) ISO 9000 sets the basic framework for the development of an optimal process and Total Quality Management (TQM).

Salient features of ISO 9001 certification

a) All documents concerned with the development of a software product should be properly managed, authorized, and controlled. This requires a configuration management system to be in place.
b) Proper plans should be prepared and then progress against these plans should be monitored.

c) Important documents should be independently checked and reviewed for effectiveness and correctness.

d) The product should be tested against specification.

e) Several organizational aspects should be addressed e.g., management reporting of the quality team.

2.2 PHASES OF SOFTWARE DEVELOPMENT LIFECYCLE

- Software life cycle models describe phases of the software cycle and the order in which those phases are executed.
- Each phase produces deliverables required by the next phase in the life cycle.
- Requirements are translated into design. Code is produced according to the design which is called development phase. After coding and development the

testing verifies the deliverable of the implementation phase against requirements.

- The testing team follows Software Testing Life Cycle (STLC) which is similar to the development cycle followed by the development team.

- There are following six phases in every Software development life cycle model:

1) Requirement gathering and analysis:

- Business requirements are gathered in this phase. This phase is the main focus of the project managers and stake holders.
- Meetings with managers, stake holders and users are held in order to determine the requirements are done at this stage.
- After requirement gathering these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be development is also studied.
- Finally, a Requirement Specification document is created which serves the purpose of guideline for the next phase of the model.
- The testing team follows the Software Testing Life Cycle and starts the Test Planning phase after the requirements analysis is completed.

•

2) Design:

- In this phase the system and software design is prepared from the requirement specifications which were studied in the first phase.
- System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture.
- The system design specifications serve as input for the next phase of the model.
- In this phase the testers comes up with the Test strategy, where they mention what to test, how to test.

3) Implementation / Coding:

- On receiving system design documents, the work is divided in modules/units and actual coding is started.
- Since, in this phase the code is produced so it is the main focus for the developer.
- This is the longest phase of the software development life cycle.

4) Testing:

- After the code is developed it is tested against the requirements to make sure that the product is actually solving

the needs addressed and gathered during the requirements phase.

- During this phase all types of functional testing like unit testing, integration testing, system testing, acceptance testing are done as well as non-functional testing are also done.

5) Deployment:

- After successful testing the product is delivered / deployed to the customer for their use.
- As soon as the product is given to the customers they will first do the beta testing.
- If any changes are required or if any bugs are caught, then they will report it to the engineering team.
- Once those changes are made or the bugs are fixed then the final deployment will happen.

6) Maintenance:

- Once when the customers starts using the developed system then the actual problems comes up and needs to be solved from time to time.

•

- This process where the care is taken for the developed product is known as maintenance.

2.6 REQUIREMENT ANALYSIS: REQUIREMENTS ELICITATION AND ANALYSIS

- After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis.
- In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.
- Requirements elicitation and analysis may involve a variety of different kinds of people in an organization.
- Each organization will have its own version or instantiation of this general model depending on local factors such as the expertise of the staff, the type of system being developed, the standards used, etc.

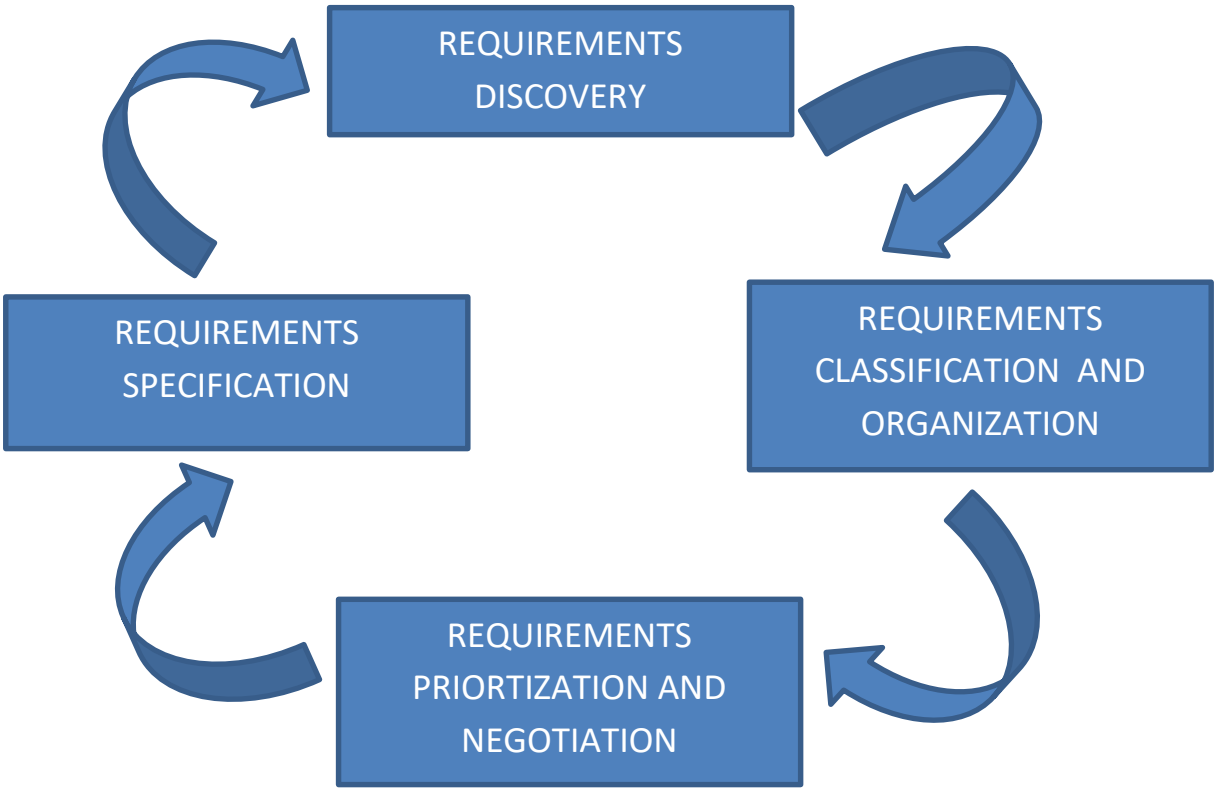


Fig: Requirement elicitation and analysis process

- The process activities are:

1. **Requirements discovery:**

   - This is the process of interacting with stakeholders of the system to discover their requirements.
   - Domain requirements from stakeholders and documentation are also discovered during this activity.

•

2. Requirements classification and organization

- This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters.
- The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system.

3. Requirements prioritization and negotiation

- This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.

4. Requirements specification

- The requirements are documented and input into the next round of the spiral.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don't know what they want from a computer system except in the most general terms.

•

2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.

3. Different stakeholders have different requirements and they may express these in different ways.

4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.

5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process.

2.6.1 Requirements Discovery

- Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information.
- Sources of information during the requirements discovery phase include documentation, system stakeholders, and specifications of similar systems. You interact with stakeholders through interviews and observation and you may use scenarios and prototypes to help stakeholders understand what the system will be like.
- Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system.

.

2.6.2 Interviews

- Formal or informal interviews with system stakeholders are part of most requirements engineering processes.
- In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed.
- Requirements are derived from the answers to these questions and interviews may be of two types:
    - Closed interviews, where the stakeholder answers a pre-defined set of questions.
    - Open interviews, in which there is no pre-defined agenda.
- It can be difficult to elicit domain knowledge through interviews for two reasons:
    - All application specialists use terminology and jargon that are specific to a domain.
    - Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning.
- Effective interviewers have two characteristics:
    - They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders.
    - They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

•

2.6.3 Scenarios

- People usually find it easier to relate to real-life examples rather than abstract descriptions.
- They can understand and criticize a scenario of how they might interact with a software system.
- Requirements engineers can use the information gained from this discussion to formulate the actual system requirements.
- Scenarios can be particularly useful for adding detail to an outline requirements description.
- They are descriptions of example interaction sessions. Each scenario usually covers one or a small number of possible interactions.
- Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system.
- Scenario-based elicitation involves working with stakeholders to identify scenarios and to capture details to be included in these scenarios.
- Scenarios may be written as text, supplemented by diagrams, screen shots, etc.
- Alternatively, a more structured approach such as event scenarios or use cases may be used.

2.6.4 Use cases

- Use cases are a requirements discovery technique that were first introduced in the objectory method and they have now become a fundamental feature of the unified modeling language.

•

- In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction.
- This is then supplemented by additional information describing the interaction with the system.
- The additional information may be a textual description or one or more graphical models such as UML sequence or state charts.
- Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements.
- Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse.
- Lines link the actors with the interaction.
- Use cases identify the individual interactions between the system and its users or other systems.
- Scenarios and use cases are effective techniques for eliciting requirements from stakeholders who interact directly with the system.
- Each type of interaction can be represented as a use case.
- However, because they focus on interactions with the system, they are not as effective for eliciting constraints or high-level business and nonfunctional requirements or for discovering domain requirements.

2.6.5 Ethnography

- Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes.

•

- The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.
- Ethnography is particularly effective for discovering two types of requirements:
    1. Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work.
    2. Requirements that are derived from cooperation and awareness of other people's activities
- Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques.
- However, because of its focus on the end-user, this approach is not always appropriate for discovering organizational or domain requirements.
- They cannot always identify new features that should be added to a system.
- Ethnography is not, therefore, a complete approach to elicitation on its own and it should be used to complement other approaches, such as use case analysis.

2.7 ANALYSIS PRINCIPLES

- Over the past two decades, a large number of analysis modeling methods have been developed.
- Investigators have identified analysis problems and their causes and have developed a variety of notations and corresponding sets of heuristics to overcome them.

•

- Each analysis method has a unique point of view.
    i. The information domain of a problem must be represented and understood.
    ii. The functions that the software is to perform must be defined.
    iii. The behavior of the software must be represented.
    iv. The models that depict information function and behavior must be partitioned in a manner that uncovers details in a layered fashion.
    v. The analysis process should move from essential information toward implementation detail.

- In addition to these operational analysis principles for requirements engineering:
    a) Understand the problem before you begin to create the analysis model.
    b) Develop prototype that enable a user to understand how human/machine interaction will occur.
    c) Record the origin of and the reason for every requirement.
    d) Use multiple views of requirements.
    e) Rank requirements.
    f) Work to eliminate ambiguity

2.8 SOFTWARE PROTOTYPING

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

- Software prototyping is becoming very popular as a software development model, as it enables to understand customer requirements at an early stage of development.
- It helps get valuable feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.
- Prototype is a working model of software with some limited functionality.
- The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.
- Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.
- Following is a stepwise approach explained to design a software prototype.

i. Basic Requirement Identification

- This step involves understanding the very basics product requirements especially in terms of user interface.
- The more intricate details of the internal design and external aspects like performance and security can be ignored at this stage.

ii. Developing the initial Prototype

- The initial Prototype is developed in this stage, where the very basic requirements are showcased and user interfaces are provided.

- These features may not exactly work in the same manner internally in the actual software developed.

- While, the workarounds are used to give the same look and feel to the customer in the prototype developed.

iii. Review of the Prototype

- The prototype developed is then presented to the customer and the other important stakeholders in the project.
- The feedback is collected in an organized manner and used for further enhancements in the product under development.

iv. Revise and Enhance the Prototype

- The feedback and the review comments are discussed during this stage and some negotiations happen with the customer based on factors like – time and budget constraints and technical feasibility of the actual implementation.
- The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until the customer expectations are met.

- Prototypes can have horizontal or vertical dimensions. A Horizontal prototype displays the user interface for the product and gives a broader view of the entire system, without concentrating on internal functions.

•

- A Vertical prototype on the other side is a detailed elaboration of a specific function or a sub system in the product.
- Horizontal prototypes are used to get more information on the user interface level and the business requirements. It can even be presented in the sales demos to get business in the market.
- Vertical prototypes are technical in nature and are used to get details of the exact functioning of the sub systems. For example, database requirements, interaction and data processing loads in a given sub system.

2.8.1 Software Prototyping - Types

- **Throwaway/Rapid Prototyping**
    - Throwaway prototyping is also called as rapid or close ended prototyping.
    - This type of prototyping uses very little efforts with minimum requirement analysis to build a prototype.
    - Once the actual requirements are understood, the prototype is discarded and the actual system is developed with a much clear understanding of user requirements.
- **Evolutionary Prototyping**
    - Evolutionary prototyping also called as breadboard prototyping is based on building actual functional prototypes with minimal functionality in the beginning.

- The prototype developed forms the heart of the future prototypes on top of which the entire system is built.

- By using evolutionary prototyping, the well-understood requirements are included in the prototype and the requirements are added as and when they are understood.

- **Incremental Prototyping**

  - Incremental prototyping refers to building multiple functional prototypes of the various sub-systems and then integrating all the available prototypes to form a complete system.

- **Extreme Prototyping**

  - Extreme prototyping is used in the web development domain. It consists of three sequential phases.

  - First, a basic prototype with all the existing pages is presented in the HTML format.

- Then the data processing is simulated using a prototype services layer.

- Finally, the services are implemented and integrated to the final prototype.

- This process is called Extreme Prototyping used to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the actual services.

- The advantages of the Prototyping Model are as follows :
    a) Increased user involvement in the product even before its implementation.
    b) Since a working model of the system is displayed, the users get a better understanding of the system being developed.
    c) Reduces time and cost as the defects can be detected much earlier.
    d) Quicker user feedback is available leading to better solutions.
    e) Missing functionality can be identified easily.
    f) Confusing or difficult functions can be identified.

- The Disadvantages of the Prototyping Model are as follows:

•

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

.

MODULE 3

3.1 PLANNING PHASE

- The software project management process begins with set of activities called project planning.
- It is the heart of project life cycle, and informs all involved where to go and how to go.
- It helps to manage time, cost, quality, changes, risk and related issues.

3.1.1 Purpose of Project Planning Phase

i. Establish business requirements.
ii. Establish cost, schedule, list of deliverables and delivery dates.
iii. Establish resource plans.
iv. Obtain management approval and proceed to next phase.

3.1.2 Basic process of Project Planning

(a) Software planning: specify in-scope requirement for project to facilitate creating work break down structure.

(b) Preparation of work breakdown structure: Breakdown the project into tasks and sub-tasks.

•

(c) Project schedule development: Listing the entire schedule of activities and sequence of implementation.

(d) Resource planning: It specifies who will do what, at which time and any special skill needed.

(e) Budget planning: It specifies cost to be incurred at completion of the project.

(f) Procurement planning: It focuses on vendors outside your company and subcontracting.

(g) Risk management: It includes possible risks and solutions for them.

(h) Quality planning: It is assessing quality criteria to be used.

(i) Communication planning: It includes designing communication strategy with stakeholders.

3.1.3 Software Scope

- It is the first step in project planning.
- It should have function and performance details which is unambiguous and understandable at management and technical levels.
- It describes data and control to be processed, function, performance, constraints, interfaces and reliability.
- It addresses the following:

•

(i) **Function:** Actions and information transformations performed by the system.

(ii) **Performance:** Processing and response time required.

(iii) **Constraints:** Limits placed over software like memory restriction.

(iv) **Interfaces:** Interaction with user and other system.

(v) **Reliability:** Quantitative requirements for functional performance like mean time between failures, acceptable error rates.

3.1.4 Resources

- The second stage in planning is estimation of resources requires.
- There are three types of resources in project planning and they are as follows:

(i) Human Resources:

- Here both organizational position and specialty are considered.
- This need to identify: (i) skill set; and (ii) development effort.

(ii) Reusable Software Resources:

- Component Based Software Engineering (CBSE) emphasizes reusability- creation and reuse of software building blocks.
- Such building blocks are called components.

•

- It can be categorized as:
  (a) Off-the-shelf components: It aims at the use of existing software from third party.
  (b) Full-experience components: It focus on use of in house software already developed which requires less modification, All the team members will be experienced with the software.
  (c) Partial-experience components: It focuses on use of in house software already developed which requires major modification. All the team members will be experienced with the software.
  (d) New Components: It focuses on building new softwares.

(iii) Environmental Resources:

- Hardware provides the platform to support the softwares required to produce the work.

3.1.5 Empirical Cost Estimation Model: COCOMO Model

- The Constructive Cost Model (COCOMO) is a procedural software cost estimation model developed by Barry W Boehm.
- COCOMO is used to estimate size, effort and duration based on the cost of the software
- COCOMO consists of a hierarchy of three increasingly detailed and accurate forms.

•

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

Basic COCOMO

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

    - **Organic projects** - "small" teams with "good" experience working with "less than rigid" requirements
    - **Semi-detached projects** - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements
    - **Embedded projects** - developed within a set of "tight" constraints. It is also combination of organic and semi-detached projects.(hardware, software, operational, ...)
- The basic COCOMO equations take the form

•

Effort Applied (E) = $a_b(KLOC)^{b_b}$

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

People required (P) = Effort Applied / Development Time

where, KLOC is the estimated number of delivered lines (expressed in thousands) of code for project.

- The coefficients a_b, b_b, c_b and d_b are given in the following table:

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

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

Intermediate COCOMO

- Intermediate COCOMO computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes.

- This extension considers a set of four "cost drivers", each with a number of subsidiary attributes:-

(a) Product attributes

Required software reliability extent

Size of application database

Complexity of the product

(b) Hardware attributes

Run-time performance constraints

Memory constraints

Volatility of the virtual machine environment

Required turnabout time

(c) Personnel attributes

Analyst capability

Software engineering capability

Applications experience

Virtual machine experience

Programming language experience

(d) Project attributes

Use of software tools

Application of software engineering methods

Required development schedule

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

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Size of application database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
Run-time performance constraints			1.00	1.11	1.30	1.66
Memory constraints			1.00	1.06	1.21	1.56
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30	
Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Software engineer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

•

- The Intermediate Cocomo formula now takes the form:

$$E=a_i(KLoC)(b_i)(EAF)$$

where E is the effort applied in person months,

KLoC is the estimated number of thousands of delivered lines of code for the project,

EAF is the factor calculated above.

- The coefficient a_i and the exponent b_i are given in the above table.
- The Development time D calculation uses E in the same way as in the Basic COCOMO.

Detailed COCOMO

- Detailed COCOMO incorporates all characteristics of the intermediate version

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

with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

•

- The detailed model uses different effort multipliers for each cost driver attribute.
- These Phase Sensitive effort multipliers are each to determine the amount of effort required to complete each phase.
- In detailed COCOMO, the whole software is divided into different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort.
- The effort is calculated as a function of program size and a set of cost drivers are given according to each phase of the software life cycle.
- The Six phases of detailed COCOMO are:-
    a) planning and requirements
    b) system design
    c) detailed design
    d) module code and test
    e) integration and test
    f) Cost Constructive model

3.1.6 STAFFING AND PERSONAL PLANNING

- Staffing is the practice of finding, evaluating, and establishing a working relationship with future colleagues on a project and firing them when they are no longer needed.
- Staffing involves finding people, who may be hired or already working for the company (organization) or may be working for competing companies.

- In knowledge economies, where talent becomes the new capital, this discipline takes on added significance to help organizations achieve a competitive advantage in each of their marketplaces.
- "Staffing" can also refer to the industry and/or type of company that provides the functions described in the previous definition for a price.
- A staffing company may offer a variety of services, including temporary help, permanent placement, managed services, training, etc.

Staffing a Software Project

- Staffing must be done in a way that maximizes the creation of some value to a project.
- In this sense, the semantics of value must be addressed regarding project and organizational characteristics.
- Some projects are schedule driven: to create value for such project could mean to act in a way that reduces its schedule or risks associated to it.
- Other projects may be driven by budget, resource allocation, and so on.
- Staff allocation optimizer cannot be fixed by a single utility function, but several such functions should be available for the manager to decide which best fit the project under analysis.
- In our approach, we consider that a characteristic may be a skill, a capability, an experience, some knowledge, a role in the organization or in the project, and others.
- Each characteristic is associated with a rating scale, with the intensity levels the characteristic may assume.
- Staffing is performed according to the following rules:

•

(a) A person can only be allocated to an activity if he or she possesses at least all the characteristics demanded by the activity, in an intensity level greater or equal to the demanded level of the activity.

(b) A person can only be allocated to an activity if he or she is available to perform the activity in the period it needs to be performed.

Personal Planning

- This includes estimation or allocation of right persons or individuals for the right type of tasks of which they are capable.
- The capability of the individuals should always match with the overall objective of the project.
- In software Engineering, personnel planning should be in accordance to the final development of the project.

Staffing Activities for a Software Project

a) Fill organizational positions
   Select, recruit, or promote qualified people for each project position
b) Assimilate newly assigned software personnel
   Orient and familiarize new people with the organization, facilities, and tasks to be done on the project

•

c) Educate or train personnel as necessary
   Make up deficiencies in position qualifications through training and education
d) Provide for general development of the project staff
   Improve knowledge, attitudes, and skills of project personnel
e) Evaluate and appraise project personnel
   Record and analyze the quantity and quality of project work as the basis for personnel evaluations; set performance goals and appraise personnel periodically
f) Compensate the project personnel
   Provide wages, bonuses, benefits, or other financial remuneration commensurate with project responsibilities and performance
g) Terminate project assignments
   Transfer or separate project personnel as necessary
h) Document project staffing decisions
   Record staffing plans, training plans and achievements, appraisal records, and compensations recommendations.

3.2 DESIGN PHASE

- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.
- Initially, the blueprint depicts a holistic view of software.
- That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.

•

- As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

3.2.1 Software Quality Guidelines and Attributes

- There are three characteristics that serve as a guide for the evaluation of a good design:
  a) The design should implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
  b) The design should be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
  c) The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.
- In order to evaluate the quality of a design representation, the software team must establish technical criteria for good design and few of the guidelines are as follows:
  a) A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design, and (3) can be implemented in an evolutionary fashion.
  b) A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

•

c) A design should contain distinct representations of data, architecture, interfaces,and components.
d) A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
e) A design should lead to components that exhibit independent functional characteristics.
f) A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
g) A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
h) A design should be represented using a notation that effectively communicates its meaning.

- The quality attributes represent a target for all software design and few of them are as follows:
  a) **Functionality** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
  b) **Usability** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
  c) **Reliability** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

•

d) **Performance** is measured using processing speed, response time, resource consumption, throughput, and efficiency.
e) **Supportability** combines extensibility, adaptability, and serviceability.

3.2.2 Design Principles

- Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.
- The design model begins by representing the totality of the thing to be built and slowly refines the thing to provide guidance for constructing each detail.
- Basic design principles enable the software engineer to navigate the design process and following are the list of principles:
  (i) The design process should not suffer from "tunnel vision."
  (ii) The design should be traceable to the analysis model.
  (iii) The design should not reinvent the wheel
  (iv) The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.
  (v) The design should exhibit uniformity and integration.
  (vi) The design should be structured to accommodate change.
  (vii) The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
  (viii) Design is not coding, coding is not design.
  (ix) The design should be assessed for quality as it is being created, not after the fact.

•

(x) The design should be reviewed to minimize conceptual (semantic)errors.

3.2.3Design Concepts

- A set of fundamental software design concepts has evolved over the history of software engineering.
- Each provides the software designer with a foundation from which more sophisticated design methods can be applied.
- Each helps you define criteria that can be used to partition software into individual components, separate or data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design.

(a) Abstraction:

- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction,a more detailed description of the solution is provided.
- As different levels of abstraction are developed, you work to create both procedural and data abstractions.
- A *procedural abstraction* refers to a sequence ofinstructions that have a specific and limited function.
- The name of a proceduralabstraction implies these functions, but specific details are suppressed.

- A *data abstraction* is a named collection of data that describes a data object.

(b) Architecture

- Software architecture aims to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.

- Architecture is the structure or organization of program components (modules), the manner in which these components interact, and thestructure of data that are used by the components.

- One goal of software design is to derive an architectural rendering of a system.

- Structural properties define the componentsof a system (e.g., modules, objects, filters) and the manner in which those componentsare packaged and interact with one another.

- Given the specification of these properties, the architectural design can berepresented using one or more of a number of different models.

- *Structuralmodels* represent architecture as an organized collection of programcomponents.

- *Framework models* increase the level of design abstraction byattempting to identify repeatable architectural design frameworks (patterns)that are encountered in similar types of applications.

- *Dynamic models* addressthe behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

- *Processmodels* focus on the design of the business or technical process that the systemmust accommodate.

- Finally, *functional models* can be used to represent the functionalhierarchy of a system.

- A number of different architectural description languages (ADLs) have beendeveloped to represent these models.

- Although many different ADLshave been proposed, the majority provide mechanisms for describing systemcomponents and the manner in which they are connected to one another.

(c) Pattern

- A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns.

- The intent of each design pattern is to provide a description that enables a designer to determine:

  (1) Whether the pattern is applicable to the current work,

  (2) Whether the pattern can be reused, and

  (3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

(d) Separation of concerns

- Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- A concern is a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smallerand therefore more manageable pieces, a problem takes less effort and time to solve.
- This leads to a divide-and-conquer strategy where it is easier to solvea complex problem when you break it into manageable pieces.
- Separation of concerns is manifested in other related design concepts: modularity,aspects, functional independence, and refinement.

(e) Modularity

- Modularity is the most common manifestation of separation of concerns.
- Software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

- Modularity is the single attribute of software that allowsa program to be intellectually manageable.
- You modularize a design (and the resulting program) so that:
    i. Development can be more easily planned;
    ii. Software increments can be defined and delivered;
    iii. Changes can be more easily accommodated;
    iv. Testing and debugging can be conducted more efficiently, and
    v. Long-term maintenance can be conducted without serious side effects.

(f) Information Hiding

- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
- The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance.
- Because most data and procedural detail are hidden from other parts of the software, errors introduced during modification are less likely to propagate to other locations within the software.

(g) Functional Independence

- Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified.

- Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.
- Functional independence is a key to good design, and design is the key to software quality.
- Independence is assessed using two qualitative criteria: **cohesion** and **coupling**.
- *Cohesion* is an indication of the relative functional strength of a module.
- Cohesive module performs a single task, requiring little interactionwith other components in other parts of a program.
- *Coupling* is an indication of the relative interdependence among modules.
- Coupling is an indication of interconnection among modules in a softwarestructure.
- Coupling depends on the interface complexity between modules, thepoint at which entry or reference is made to a module, and what data pass across the interface.

(h) Refinement

- Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth.
- An application is developed by successively refining levels of procedural detail.

•

- Abstraction enables you to specify procedure and data internally but suppress the need for outsiders to have knowledge of low-level details.
- Refinement helps you to reveal low-level details as design progresses.

(i) Refactoring

- Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur.
- As a consequence, refactoring tools are used to analyze changes automatically and to generate a test suite suitable for detecting behavioral changes.

(j) Object Oriented Design Concepts

- The object-oriented (OO) paradigm is widely used in modern software engineering.
- OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others are employed.

3.3 EFFECTIVE MODULAR DESIGN

- Modularity has become an accepted approach in all engineering disciplines.

•

- A modular design reduces complexity, facilitates change, and results in easier implementation by encouraging parallel development of different parts of a system.
- The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding.
- Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified.
- Functional independence is a key to good design, and design is the key to software quality.
- Independence is measured using two qualitative criteria: **cohesion** and **coupling**.
- Cohesion is a measure of the relative functional strength of a module.
- Coupling is a measure of the relative interdependence among modules.

3.3.1 Cohesion

- Cohesion is a natural extension of the information hiding concept.
- A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program.
- Cohesion may be represented as a "spectrum.“ and it can have:
    a. Modules that perform a set of tasks that relate to each other loosely are termed ***coincidentally cohesive***.

•

b. A module that performs tasks that are related logically is ***logically cohesive***.
c. When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits ***temporal cohesion***.

- When processing elements of a module are related and must be executed in a specific order, ***procedural cohesion*** exists.
- When all processing elements concentrate on one area of a data structure, ***communicational cohesion*** is present.

3.3.2 Coupling

- Coupling is a measure of interconnection among modules in a software structure.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates through a system.
- Figure below shows the different types of module coupling:

•

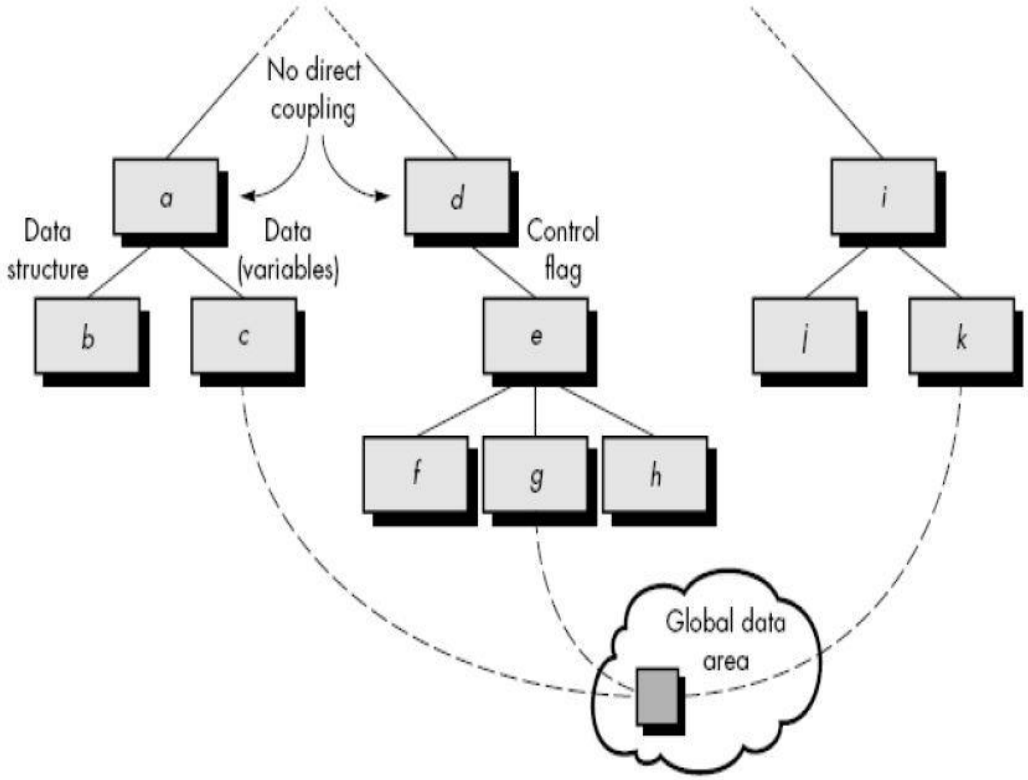


Fig: Types of Coupling

- As long as a simple argument list is present, i.e, simple data are passed low coupling or **data coupling** is exhibited.
- A variation of data coupling, called **stamp coupling** is found when a portion of a data structure is passed via a module interface. This occurs between modules b and a.
- **Control coupling** is very common in most software designs where a "control flag" is passed between modules like d and e.

•

- High coupling occurs when a number of modules reference a global data area and is called **Common coupling** like Modules c, g, and k each access a data item in a global data area.
- The highest degree of coupling, ***content coupling***, occurs when one module makes use of data or control information maintained within the boundary of another module.

3.4 TOP DOWN DESIGN

- Top-down design is a term used to describe how a complex problem is broken down into modules.
- Those modules are then broken down into sub-modules.
- Each sub-module is then broken down further and further, until the sub-modules do just one task and are simple enough to program.
- Instead of writing one big program, the program is split into self-contained 'modules' of code. Modules are sometimes called 'procedures' or 'functions'.

3.4.1 Advantages of Top down Design

(i) It helps get the job done more efficiently because modules of code can be worked on at the same time by different programmers

(ii) It helps a team of programmers work more efficiently because easier modules can be given to less experienced programmers while the harder ones can be given to more experienced ones.

•

(iii) It helps program testing because it is easier to debug lots of smaller self-contained modules than one big program.

(iv) It helps program readability because it is easier to follow what is going on in smaller modules than a big program.

(v) It improves a company's efficiency because self-contained modules can be re-used.

(vi) It improves a Project Manager's ability to monitor the progress of a program.

(vii) It is good for future program maintenance. If a program needs to be changed for any reason, it may be possible simply to remove one module of code and replace it with another.

3.4.2 Disadvantages of Top Down Design

i) The solution provides limited coverage in the first phases.
ii) A minimal percentage of user accounts are managed in the first phases.
iii) You might have to develop custom adapters at an early stage.
iv) The support and overall business will not realize the benefit of the solution as rapidly.
v) The implementation cost is likely to be higher.

3.5 BOTTOM UP DESIGN

- In a bottom-up approach, the individual base elements of the system are first specified in great detail.

•

- These elements are then linked together to form larger subsystems, which then in turn are linked, until a complete top-level system is formed.
- This strategy often resembles a "seed" model, by which the beginnings are small, but eventually grow in complexity and completeness.

3.5.1 Advantages of Bottom Up Design

(i) User and business awareness of the product. Benefits are realized in the early phases.

(ii) You can replace many manual processes with early automation.

(iii) You can implement password management for a large number of users.

(iv) You do not have to develop custom adapters in the early phases.

(v) Your organization broadens identity management skills and understanding during the first phase.

(vi) Tivoli Identity Manager is introduced to your business with less intrusion to your operations.

3.5.2Disadvantages of Bottom Up Design

(i) The organizational structure you establish might have to be changed in a later roll-out phase.

(ii) Because of the immediate changes to repository owners and the user population, the roll-out will have a higher impact earlier and require greater cooperation.

•

(iii) This strategy is driven by the existing infrastructure instead of the business processes.

.

MODULE 4

- The goal of coding is to implement the design in best possible way.
- While coding, program should not be constructed so that it easy to write instead it should be understandable and readable.
- There are many different criteria for judging the program like readability, size of the program, execution time, and required memory.
- Readability and understandability are the main objectives that help in producing software that is more maintainable.

Programming Principles and Guidelines

- The main task before a programmer is to write quality code with few bugs in it.
- Good programming is a practice independent of target programming language.

Common Coding Errors

- Software errors are reality that all programmers have to deal with.
- Though errors can occur in wide variety of ways, some types of errors are found more commonly like:

•

(i) Memory leaks

- It is a situation where the memory is allocated to the program which is not freed subsequently.
- This error is common failures which occur in languages that do not have automatic garbage collection.
- They have little impact on small programs but drastic for long programs.
- A software program with memory leaks keeps consuming memory, till at some point of time the program may come to an exceptional halt because of lack of free memory.

(ii) Freeing an Already Freed Resource

- In programs, resources are first allocated and then freed.
- This error occurs when the programmer tries to free the already freed resource.
- The impact of this error is more severe if we have some malloc statement between the two free statements, there is a chance that the first freed location is now allocated to the new variable and the subsequent free will deallocate it.

(iii) NULL Dereferencing

•

- It occurs when we try to access the contents of location that points to NULL.
- It is a common occurring error which can bring software system down.
- It is also difficult to detect NULL dereferencing as it may occur only in some paths and under certain situations.

(iv) Lack of Unique Addresses

- Aliasing creates problems and among them is violation of unique addresses when we expect different addresses.
- For example, in string concatenation function, we expect source and destination addresses to be different.
- If this is not the case, it can lead to runtime errors.\

(v) Synchronization errors

- These errors are hard to find as they don't occur so often but when occurs it causes serious damage to the system.
- There are different categories of synchronization errors and some of them are as follows:
    1. Deadlocks
    2. Race conditions
    3. Inconsistent synchronization

- Deadlock is a situation in which one or more threads mutually lock each other.
- Race condition occurs when two threads try to access the same resource and result of the execution depends on order of execution of errors.
- Inconsistent synchronization is also common error representing situation where there is a mix of locked and unlocked accesses to some shared variables.

(vi) Array Index Out of Bounds

- Array index goes out of bounds, leading to exceptions.
- Array index values cannot be negative or should not exceed their bounds.

(vii) Arithmetic Exceptions

- These include errors like divide by zero and floating point exceptions.
- The result of these may vary from getting unexpected results to termination of the program.

.

4.1 Some Programming Practices

(i) Control Constructs

It is desirable to use a few standard control constructs rather than wide variety of constructs, just because they are available in language.

(ii) Gotos

Goto should be used sparingly and in disciplined manner. Only when the alternative is using gotos is more complex should the gotos be used.

(iii) Information Hiding

The access functions for the data structures should be made visible while hiding the data structure behind these functions.

(iv) Nesting

If nesting of if-then-else constructs becomes too deep, then the logic become harder to understand. It is often difficult to which a particular else cause is associated.

For example, in the below case:

```
if C1 then S1
        else if C2 then S2
```

•

else if C3 then S3;

If these are disjoint, the structure can be converted as:

if C1 then S1
if C2 then S2
if C3 then S3

This sequence of statements will produce same result but is much easier to understand.

(v) Module size

Programmer should carefully examine any function with too many statements and large modules will not be functionally cohesive. The guideline for modularity should be cohesion and coupling.

(vi) Module Interface

A module with complex interface should be carefully examined. If the interface is complex with more than 5 parameters should be examined and broken into modules with simpler interface.

(vii) Side Effects

When module is invoked, it creates side effects of modifying program state beyond the modification of parameters in the interface.

(viii) Robustness

•

A program might face exceptional conditions like overflow, and in such situations programs should not crash or halt instead should produce some meaningful message and exit successfully.

(ix) Switch case with default

If there is no default case in switch statement, the behavior can be unpredictable at development stage as it can result in bug like NULL dereferencing, memory leak etc.

(x) Empty Catch Block

There are chances that if an exception is caught, there is no action defined and some of the operations may not be performed. It is always good to use catch block even if it is just an error message.

(xi) Trusted Data Sources

Checks should be made before accessing the input data, particularly if it is being provided by the user or is being obtained over the network. Some checks should be done like parity checks, hashes, etc. to ensure the validity of incoming data.

(xii) Give Importance to Exception

Most programmers give less importance to the possible exceptional cases and tend to work with main flow. Though main work is done in main path, it is the exceptional paths that often cause software systems to fail.

.

4.2 Coding Standards

- Programmers spend more time reading the code than writing code.
- Prime importance is to write code in a manner that it is easy to read and understand.
- Coding standards provide rules and regulations for some aspects of programming in order to make code easier to read.
- Most organizations that develop software regularly develop coding standards.
- The major coding standards include the following:

(a) Naming Conventions

- Package names should be in lower case.
- Variable names should be nouns starting with lower case.
- Constant names should all be uppercase.
- Method names should be verbs starting with lowercase.
- Variables with a large scope should have long names and short names with small scope.
- Private class variables should have the _ suffix.

(b) Files

There are conventions on how files should be named, and what files should contain, such that reader can get some idea about file contents.

For Eg: Java source files should have extension .java

Each file should contain class name same as the file name.

(c) Statements

- These guidelines are for the declaration and executable statements in the source code.
- Not everyone organization will agree to this and develop their own guidelines without restricting the flexibility of programmers.
- Some of the common statement guidelines includes the following:

(i) Variables should be initialized where declared.

(ii) Declare related variables together in a common statement.

(iii) Class variables should never be declared public.

(iv) Loop variables should be initialized immediately before the loop.

(v) Avoid use of break and continue in a loop.

(d) Commenting and Layout

- Comments are textual statements that are meant for the program reader to understand the code.
- Comments should explain what the code is doing or why the code is there.
- Providing comments for modules is most useful, as it forms unit of testing, compiling, verification and modification.
- Comments for a module are known as *prologue* which describes the functionality and purpose of the module.
- If the module is modified, then the prologue should also be modified.
- Some guidelines of this are as follows:

(i) Single line comments for a block of code should be aligned with the code.

•

(ii) There should be comments for all major variables

TESTING

- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- The main testing objectives includes:
    a) Testing is a process of executing a program with intent of finding an error.
    b) A good test case is one that has a high probability of ending an undiscovered error.
    c) A successful test is one that uncovers all errors.
- The testing principle includes the following:
    a) All tests should be traceable to customer requirement.
    b) Test should be planned long before testing begins.
    c) Exhaustive testing is not possible
    d) To be most effective, testing should be conducted by an independent third party.

4.3 Black Box Testing

- It is also known as Behavioral Testing
- It is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester.
- These tests can be functional or non-functional, though usually functional.

•

- This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see.
- This method attempts to find errors in the following categories:

  a) Incorrect or missing functions
  b) Interface errors
  c) Errors in data structures or external database access
  d) Behavior or performance errors
  e) Initialization and termination errors

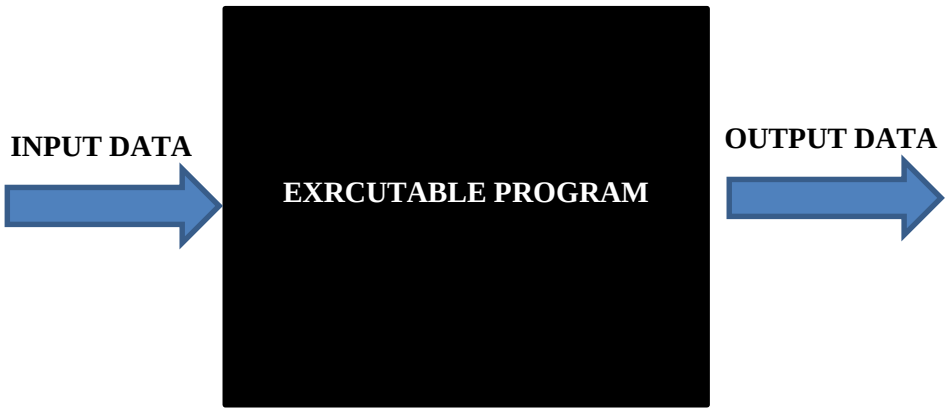


Fig: Black Box Testing

4.3.1 Techniques

- There are different techniques involved in black-box testing and some are as follows:

(a) Equivalence class portioning

- The natural approach is to divide the input domain into a set of equivalence classes, so that if the program works correctly for a value, then it will work correctly for all the other values in that class.
- The equivalence class partitioning method tries to approximate this ideal. An equivalence class is formed of the inputs for which the behavior of the system is specified or expected to be similar.
- Each group of inputs for which the behavior is expected to be different from others is considered a separate equivalence class.
- The rationale of forming equivalence classes like this is the assumption that if the specifications require the same behavior for each element in a class of values, then the program is likely to be constructed so that it either succeeds or fails for each of the values in that class.
- One common approach for determining equivalence classes is that if there is reason to believe that the entire range of an input will not be treated in the same manner, then the range should be split into two or more equivalence classes, each consisting of values for which the behavior is expected to be similar.
- Another approach for forming equivalence classes is to consider any special value for which the behavior could be different as an equivalence class.
- Once equivalence classes are selected for each of the inputs, then the issue is to select test cases suitably.
- One strategy is to select each test case covering as many valid equivalence classes as it can, and one separate test case for each invalid equivalence class.

- A somewhat better strategy which requires more test cases is to have a test case cover at most one valid equivalence class for each input, and have one separate test case for each invalid equivalence class.
- In the latter case, the number of test cases for valid equivalence classes is equal to the largest number of equivalence classes for any input, plus the total number of invalid equivalence classes.

(b) Boundary Value Analysis

- It has been observed that programs that work correctly for a set of values in an equivalence class fail on some special values.
- These values often lie on the boundary of the equivalence class. Test cases that have values on the boundaries of equivalence classes are therefore likely to be "high-yield" test cases, and selecting such test cases is the aim of boundary value analysis.
- In boundary value analysis, we choose an input for a test case from an equivalence class, such that the input lies at the edge of the equivalence classes.
- Boundary values for each equivalence class, including the equivalence classes of the output, should be covered.
- Boundary value test cases are also called "extreme cases."
- In case of ranges, for boundary value analysis it is useful to select the boundary elements of the range and an invalid value just beyond the two ends (for the two invalid equivalence classes).
- So, if the range is $0.0 < x < 1.0$, then the test cases are 0.0, 1.0 (valid inputs), and −0.1, and 1.1 (for invalid inputs).

•

(c) Pair ways testing

- Many of the defects in software generally involve one condition, that is, some special value of one of the parameters. Such a defect is called a single-mode fault.
- Single-mode faults can be detected by testing for different values of different parameters.
- However, all faults are not single-mode and there are combinations of inputs that reveal the presence of faults.
- These multimode faults can be revealed during testing by trying different combinations of the parameter values—an approach called *combinatorial testing*.
- Some research has suggested that most software faults are revealed on some special single values or by interaction of a pair of values.
- Most faults tend to be either single-mode or double-mode.
- For testing for double-mode faults, we need not test the system with all the combinations of parameter values, but need to test such that all combinations of values for each pair of parameters are exercised. This is called *pairwise testing*.

(d) State based Testing

- There are some systems that are essentially stateless in that for the same inputs they always give the same outputs or exhibit the same behavior.

- There are, however, many systems whose behavior is state-based in that for identical inputs they behave differently at different times and may produce different outputs.
- The reason for different behavior is that the state of the system may be different, so the behavior and outputs of the system depend not only on the inputs provided, but also on the state of the system.
- The state of the system depends on the past inputs the system has received, so the state represents the cumulative impact of all the past inputs on the system.
- If the set of states of a system is manageable, a state model of the system can be built.
- The state model shows what state transitions occur and what actions are performed in a system in response to events.
- When a state model is built from the requirements of a system, we can only include the states, transitions, and actions that are stated in the requirements or can be inferred from them.
- If more information is available from the design specifications, then a richer state model can be built.
- A state model for a system has four components:
    a) **States**: Represent the impact of the past inputs to the system.
    b) **Transitions**: Represent how the state of the system changes from one state to another in response to some events.
    c) **Events:** Inputs to the system.
    d) **Actions:** The outputs for the events.

.

4.3.2 Advantages

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- Tester need not know programming languages or how the software has been implemented.
- Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
- Test cases can be designed as soon as the specifications are complete.

4.3.3 Disadvantages

- Only a small number of possible inputs can be tested and many program paths will be left untested.
- Without clear specifications, which are the situation in many projects, test cases will be difficult to design.
- Tests can be redundant if the software designer/developer has already run a test case.

4.4 White Box Testing

- White box testing is concerned with testing the implementation of the program.
- The intent of this testing is not to exercise all the different input or output conditions but to exercise the different programming structures and data structures used in the program.
- White-box testing is also called structural testing.

•

- To test the structure of a program, structural testing aims to achieve test cases that will force the desired coverage of different structures.
- One approach to structural testing: control flow-based testing, which is most commonly used in practice.

4.4.1 Control Flow based Testing

- Most common structure-based criteria are based on the control flow of the program.
- In these criteria, the control flow graph of a program is considered and coverage of various aspects of the graph is specified as criteria.
- Let the control flow graph (or simply flow graph) of a program P be G. A node in this graph represents a block of statements that is always executed together, i.e., whenever the first statement is executed, all other statements are also executed.
- An edge (i, j) (from node i to node j) represents a possible transfer of control after executing the last statement of the block represented by node i to the first statement of the block represented by node j.
- A node corresponding to a block whose first statement is the start statement of P is called the start node of G, and a node corresponding to a block whose last statement is an exit statement is called an exit node.
- A path is a finite sequence of nodes (n1, n2, ..., nk), k > 1, such that there is an edge (ni, ni+1) for all nodes ni in the sequence (except the last node nk).
- A complete path is a path whose first node is the start node and the last node is an exit node.

- The simplest coverage criterion is statement coverage, which requires that each statement of the program be executed at least once during testing.
- In other words, it requires that the paths executed during testing include all the nodes in the graph. This is also called the all-nodes criterion.
- This coverage criterion is not very strong, and can leave errors undetected.
- A more general coverage criterion is branch coverage, which requires that each edge in the control flow graph be traversed at least once during testing.
- In other words, branch coverage requires that each decision in the program be evaluated to true and false values at least once during testing.
- Testing based on branch coverage is often called branch testing.
- The trouble with branch coverage comes if a decision has many conditions in it.
- A more general coverage criterion is one that requires all possible paths in the control flow graph be executed during testing.
- This is called the path coverage criterion or the all-paths criterion, and the testing based on this criterion is often called path testing.
- The difficulty with this criterion is that programs that contain loops can have an infinite number of possible paths.

4.5 Testing Strategic Issue

- Even the best strategy will fail if a series of overriding issues are not addressed.
- A software testing strategy will succeed only when software testers:

•

(1) Specify product requirements in a quantifiable manner long before testing commences

(2) State testing objectives explicitly

(3) Understand the users of the software and develop a profile for each user category

(4) Develop a testing plan that emphasizes “rapid cycle testing,”

(5) Build “robust” software that is designed to test itself

(6) Use effective technical reviews as a filter prior to testing

(7) Conduct technical reviews to assess the test strategy and test cases themselves

(8) Develop a continuous improvement approach for the testing process.

4.6 Unit Testing

- Once a programmer has written the code for a module, it has to be verified before it is used by others.
- Testing remains the most common method of this verification and at the programmer level the testing done for checking the code the programmer has developed is called unit testing.
- Unit testing is like regular testing where programs are executed with some test cases except that the focus is on testing smaller programs or modules which are typically assigned to one programmer (or a pair) for coding.
- A unit may be a function or a small collection of functions for procedural languages, or a class or a small collection of classes for object-oriented languages.

•

- It suffices that during unit testing the tester, who is generally the programmer, will execute the unit with a variety of test cases and study the actual behavior of the units being tested for these test cases.
- Based on the behavior, the tester decides whether the unit is working correctly or not.
- If the behavior is not as expected for some test case, then the programmer finds the defect in the program (an activity called debugging), and fixes it.
- After removing the defect, the programmer will generally execute the test case that caused the unit to fail again to ensure that the fixing has indeed made the unit behave correctly.
- An issue with unit testing is that as the unit being tested is not a complete system but just a part, it is not executable by itself.
- Furthermore, in its execution it may use other modules that have not been developed yet.
- Due to this, unit testing often requires drivers or stubs to be written. Drivers play the role of the "calling" module and are often responsible for getting the test data, executing the unit with the test data, and then reporting the result.
- Stubs are essentially "dummy" modules that are used in place of the actual module to facilitate unit testing.

4.7 Integration Testing

- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

- The objective is to take unit-tested components and build a program structure that has been dictated by design.
- There is often a tendency to attempt non-incremental integration where all components are combined in advance and the entire program is tested as a whole.
- In Incremental integration the program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.
- A number of different incremental integration strategies are developed like:

(a) Top-Down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.
- Depth-first integration integrates all components on a major control path of the program structure.
- Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.

(b) Bottom-Up Integration

- Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

•

- Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.

(c) Regression Testing

- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

(d) Smoke Testing

- Smoke testing is an integration testing approach that is commonly used when product software is developed.
- It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

4.8 Validation Testing

- Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected.

- Testing focuses on user-visible actions and user-recognizable output from the system.

4.8.1 Validation Test Criteria

- Software validation is achieved through a series of tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that:
  (a) Functional requirements are satisfied
  (b) All behavioral characteristics are achieved,
  (c) All content is accurate and properly presented,
  (d) All performance requirements are attained,
  (e) Documentation is correct,
  (f) Usability and other requirements are met

4.8.2 Configuration Review

- An important element of the validation process is a configuration review.
- The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to support activities

4.8.3 Alpha and Beta Testing

- It is virtually impossible for a software developer to foresee how the customer will really use a program.

•

- Instructions for use may be misinterpreted; strange combinations of data may be used; output that seemed clear to the tester may be unintelligible to a user in the field.
- Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.
- The alpha test is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.
- The beta test is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present.
- Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer.
- The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.
- A variation on beta testing, called customer acceptance testing, is sometimes performed when custom software is delivered to a customer under contract.
- The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer.

4.9 System Testing

- Software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted.

•

- These tests fall outside the scope of the software process and are not conducted solely by software engineers.
- However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

4.9.1 Recovery Testing

- Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic (performed by the system itself), re-initialization, check-pointing mechanisms, data recovery, and restart are evaluated for correctness.
- If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

4.9.2 Security Testing

- Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.
- Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.

4.9.3 Stress Testing

- Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

•

- A variation of stress testing is a technique called sensitivity testing.
- In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.

4.9.4 Performance Testing

- Performance testing occurs throughout all steps in the testing process.
- Even at the unit level, the performance of an individual module may be assessed as tests are conducted.
- However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

4.9.5 Deployment Testing

- Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate.
- In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

•

MODULE 5

5.1 MAINTENANCE

- Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updates done after the delivery of software product.
- There are number of reasons, why modifications are required, some of them are briefly mentioned below:

a) **Market Conditions** - Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.

b) **Client Requirements** - Over the time, customer may ask for new features or functions in the software.

c) **Host Modifications** - If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.

d) **Organization Changes** - If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

•

5.1.2 Types of Maintenance

- In a software lifetime, type of maintenance may vary based on its nature.
- It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature.
- Following are some types of maintenance based on their characteristics:
- **Corrective Maintenance** - This includes modifications and updates 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 updates 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 updates 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.

5.1.3 Maintenance Activities

- IEEE provides a framework for sequential maintenance process activities.
- It can be used in iterative manner and can be extended so that customized items and processes can be included.

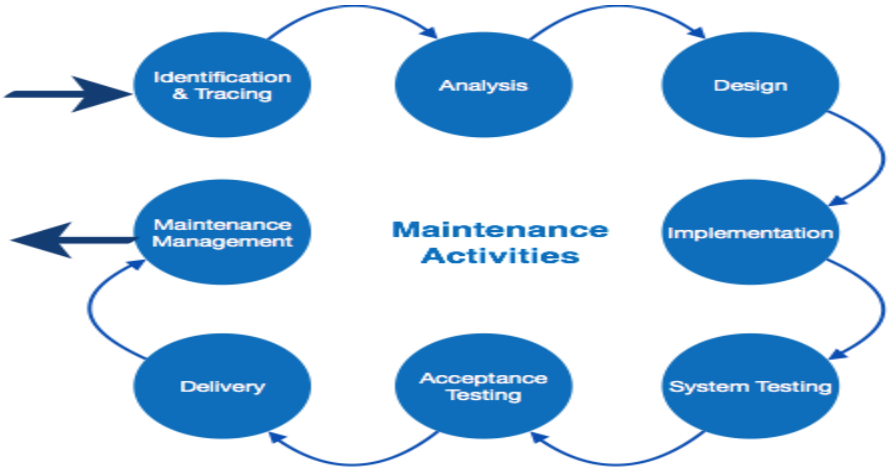


Fig: Maintenance Activities

- These activities go hand-in-hand with each of the following phase:
  a) **Identification & Tracing** - It involves activities pertaining to identification of requirement of modification or maintenance. It is generated by user or system may itself report via logs or error messages.Here, the maintenance type is classified also.

  b) **Analysis** - The modification is analyzed for its impact on the system including safety and security implications. If probable impact is severe, alternative solution is looked for. A set of required modifications is then materialized into requirement specifications. The cost of modification/maintenance is analyzed and estimation is concluded.

  c) **Design** - New modules, which need to be replaced or modified, are designed against requirement specifications set in the previous stage. Test cases are created for validation and verification.

•

d) **Implementation** - The new modules are coded with the help of structured design created in the design step.Every programmer is expected to do unit testing in parallel.

e) **System Testing** - Integration testing is done among newly created modules. Integration testing is also carried out between new modules and the system. Finally the system is tested as a whole, following regressive testing procedures.

f) **Acceptance Testing** - After testing the system internally, it is tested for acceptance with the help of users. If at this state, user complaints some issues they are addressed or noted to address in next iteration.

g) **Delivery** - After acceptance test, the system is deployed all over the organization either by small update package or fresh installation of the system. The final testing takes place at client end after the software is delivered.Training facility is provided if required, in addition to the hard copy of user manual.

h) **Maintenance management** - Configuration management is an essential part of system maintenance. It is aided with version control tools to control versions, semi-version or patch management.

5.2 RISK MANAGEMENT

- Risk analysis and management are a series of steps that help a software team understand and manage uncertainty. Many problems can plague a software project.

•

- A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

- Everyone involved in the software process—managers, software engineers, and other stakeholders—participate in risk analysis and management.
- Recognizing what can go wrong is the first step, called "risk identification.
- Next, each risk is analyzed to determine the likelihood that it will occur and the damage that it will do if it does occur.
- Once this information is established, risks are ranked, by probability and impact.
- Finally, a plan is developed to manage those risks that have high probability and high impact.

5.2.1 Reactive versus Proactive Risk Strategies

- At best, a reactive strategy monitors the project for likely risks.
- Resources are set aside to deal with them, should they become actual problems.
- More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly.
- This is often called a fire-fighting mode. When this fails, "crisis management" takes over and the project is in real jeopardy.

•

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

5.2.2 Software Risk

- Risk always involves two characteristics: *uncertainty* —the risk may or may not happen; and *loss* —if the risk becomes a reality, unwanted consequences or losses will occur.
- The different categories of risks are
  - (i) *Project risks* threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase.
  - (ii) *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.
  - (iii) *Business risks* threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five

•

business risks are: (1) building an excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn't understand how to sell (sales risk), (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks).

- Another general categorization of risks includes:
  (i) *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment.

  (ii) *Predictable risks* are extrapolated from past project experience.

  (iii) *Unpredictable risks* are they can and do occur, but they are extremely difficult to identify in advance.

5.2.3 Risk Identification

- *Risk identification* is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.).
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.
- There are two distinct types of risks: generic risks and product-specific risks.

•

- *Generic risks* are a potential threat to every software project.
- *Product-specific risks* can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand.

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

i. ***Product size***—risks associated with the overall size of the software to be built or modified.

ii. ***Business impact***—risks associated with constraints imposed by management or the marketplace.

iii. ***Customer characteristics***—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

iv. ***Process definition—***risks associated with the degree to which the software process has been defined and is followed by the development organization.

v. ***Development environment***—risks associated with the availability and quality of the tools to be used to build the product.

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

•

- The risk item checklist can be organized in different ways.
  - Questions relevant to each of the topics can be answered for each software project.
  - The answers to these questions allow the planner to estimate the impact of risk.
  - A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of “risk components and drivers" are listed along with their probability of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions.

5.2.3.1 Assessing Overall Project Risk

- The following questions have derived from risk data obtained by surveying experienced software project managers in different part of the world.
- The questions are ordered by their relative importance to the success of a project.

  1. Have top software and customer managers formally committed to support the project?

  2. Are end-users enthusiastically committed to the project and the system/product to be built?

  3. Are requirements fully understood by the software engineering team and their customers?

  4. Have customers been involved fully in the definition of requirements?

  5. Do end-users have realistic expectations?

  6. Is project scope stable?

  7. Does the software engineering team have the right mix of skills?

•

8. Are project requirements stable?

9. Does the project team have experience with the technology to be implemented?

10. Is the number of people on the project team adequate to do the job?

11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

- If any one of these questions is answered negatively, mitigation, monitoring, and management steps should be instituted without fail.
- The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

5.2.3.2 Risk Components and Drivers

- The project manager identifies the risk drivers that affect software risk components—performance, cost, support, and schedule.
- The risk components are defined in the following manner:
    a. ***Performance risk***—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
    b. ***Cost risk***—the degree of uncertainty that the project budget will be maintained.
    c. ***Support risk***—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
    d. ***Schedule risk***—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

•

- The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic.

5.2.4Risk Mitigation, Monitoring, and Management

- All of the risk analysis activities presented to this point have a single goal: to assist the project team in developing a strategy for dealing with risk.
- An effective strategy must consider three issues:
    - Risk avoidance
    - Risk monitoring
    - Risk management and contingency planning
- If a software team adopts a proactive approach to risk, avoidance is always the best strategy.
- This is achieved by developing a plan for *risk mitigation.*
- For example, assume that high staff turnover is noted as a project risk, *r1*. Based on past history and management intuition, the likelihood, *l1*, of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact, *x1*, is projected at level 2. That is, high turnover will have a critical impact on project cost and schedule.
- **To mitigate this risk,** project management must develop a strategy for reducing turnover. Among the possible steps to be taken are:
    - Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, and competitive job market).
    - Mitigate those causes that are under our control before the project starts.
    - Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.

•

  - Organize project teams so that information about each development activity is widely dispersed.
  - Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
  - Conduct peer reviews of all work (so that more than one person is "up to speed").
  - Assign a backup staff member for every critical technologist.
  - As the project proceeds, risk monitoring activities commence.
- The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely.
- In the case of high staff turnover, the following factors can be monitored:
  - General attitude of team members based on project pressures.
  - The degree to which the team has jelled.
  - Interpersonal relationships among team members.
  - Potential problems with compensation and benefits.
  - The availability of jobs within the company and outside it.

- In addition to monitoring these factors, the project manager should monitor the effectiveness of risk mitigation steps.
- For example, a risk mitigation step noted here called for the definition of documentation standards and mechanisms to be sure that documents are developed in a timely manner.
- This is one mechanism for ensuring continuity, should a critical individual leave the project.

•

- The project manager should monitor documents carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

5.3 PROJECT MANAGEMENT CONCEPTS

- Project management involves the planning, monitoring, and control of people, process, and events that occur during software development.
- Everyone manages, but the scope of each person's management activities varies according his or her role in the project.
- Software needs to be managed because it is a complex, long duration undertaking.
- Managers must focus on the fours P's to be successful (people, product, process, and project).
- A project plan is a document that defines the four P's in such a way as to ensure a cost effective, high quality software product.
- The only way to be sure that a project plan worked correctly is by observing that a high quality product was delivered on time and under budget.
- **People** (recruiting, selection, performance management, training, compensation, career development, organization, work design, team/culture development)
- **Product** (product objectives, scope, alternative solutions, constraint tradeoffs)

•

- **Process** (framework activities populated with tasks, milestones, work products, and QA points)
- **Project (**planning, monitoring, controlling)

5.3.1The People

- The "people factor" is so important that the Software Engineering Institute has developed a *people management capabilitymaturity model* (PM-CMM), "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability"
- The people management maturity model defines the following key practice areas for software people: **recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development.**
- Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

5.3.1.1 The Players

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

  1.Senior managers who define the business issues that often have significant influence on the project.

•

2. Project (technical) managers who must plan, motivate, organize, and control the practitioners who do software work.

3. Practitioners who deliver the technical skills that are necessary to engineer a product or application.

4. Customers who specify the requirements for the software to be engineered and other *stakeholders* who have a peripheral interest in the outcome.

5. End-users who interact with the software once it is released for production use.

- Every software project is populated by people who fall within this taxonomy. To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader.

5.3.1.2 Team Leaders

- Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. MOI model of leadership:
    - **Motivation.** The ability to encourage (by "push or pull") technical people to produce to their best ability.
    - **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
    - **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

•

- Another view of the characteristics that define an effective project manager emphasizes four key traits:
  - **Problem solving.** An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.
  - **Managerial identity.** A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.
  - **Achievement.** To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his own actions that controlled risk taking will not be punished.
  - **Influence and team building.** An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

5.3.1.3 The Software Team

- The following options are available for applying human resources to a project that will require *n* people working for *k* years:

  1. *n* individuals are assigned to *m* different functional tasks, relatively little combined work occurs; coordination is the responsibility of a software manager who may have six other projects to be concerned with.

•

2. *n* individuals are assigned to *m* different functional tasks ($m < n$) so that informal "teams" are established; an ad hoc team leader may be appointed; coordination among teams is the responsibility of a software manager.

3. *n* individuals are organized into *t* teams; each team is assigned one or more functional tasks; each team has a specific structure that is defined for all teams working on a project; coordination is controlled by both the team and a software project manager.

- The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty.
- Mantei suggests three generic team:
  - **Democratic decentralized (DD).** This software engineering team has no permanent leader. Rather, "task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks." Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.
  - **Controlled decentralized (CD).** This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

•

- **Controlled Centralized (CC).** Top-level problem solving and internal team coordination are managed by a team leader. Communication between the leader and team members is vertical.

5.3.2 The Product

- Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified.
- Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.
- A software project manager is confronted with a dilemma at the very beginning of a software engineering project.
- Quantitative estimates and an organized plan are required, but solid information is unavailable.
- A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed "now!"
- Therefore, we must examine the product and the problem it is intended to solve at the very beginning of the project.
- At a minimum, the scope of the product must be established and bounded.

5.3.2.1 Software Scope

•

- The first software project management activity is the determination of *software scope.* Scope is defined by answering the following questions:
  - **Context.** How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?
  - **Information objectives.** What customer-visible data objects are produced as output from the software? What data objects are required for input?
  - **Function and performance.** What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

5.3.2.2 Problem Decomposition

- Problem decomposition, sometimes called *partitioning* or *problem elaboration,* is an activity that sits at the core of software requirements analysis.
- During the scoping activity no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas:

  (1) the functionality that must be delivered and

  (2) the process that will be used to deliver it.
- Human beings tend to apply a divide and conquer strategy when they are confronted with complex problems.
- Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins.

.

5.3.3 The Process

- A software process provides the framework from which a comprehensive plan for software development can be established.
- A small number of framework activities are applicable to all software projects, regardless of their size or complexity.
- A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.
- Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model.
- Umbrella activities are independent of any one framework activity and occur throughout the process.
- The generic phases that characterize the software process—definition, development, and support—are applicable to all software
- . The problem is to select the process model that is appropriate for the software to be engineered by a project team.

5.3.3.1 Melding the Product and the Process

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

- Assume that the organization has adopted the following set of framework activities :
    a. *Customer communication*—tasks required to establish effective requirements elicitation between developer and customer.
    b. *Planning*—tasks required to define resources, timelines, and other project related information.
    c. *Risk analysis*—tasks required to assess both technical and management risks.
    d. *Engineering*—tasks required to build one or more representations of the application.
    e. *Construction and release*—tasks required to construct, test, install, and provide user support (e.g., documentation and training).
    f. *Customer evaluation*—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering activity and implemented during the construction activity.
- The team members who work on a product function will apply each of the framework activities to it.
- In essence, a matrix similar to the one shown in figure below.
- Each major product function (the figure notes functions for the word-processing software) is listed in the left-hand column.
- Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.5
- The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the tasks

associated with each cell, and work products to be produced as a consequence of each task.

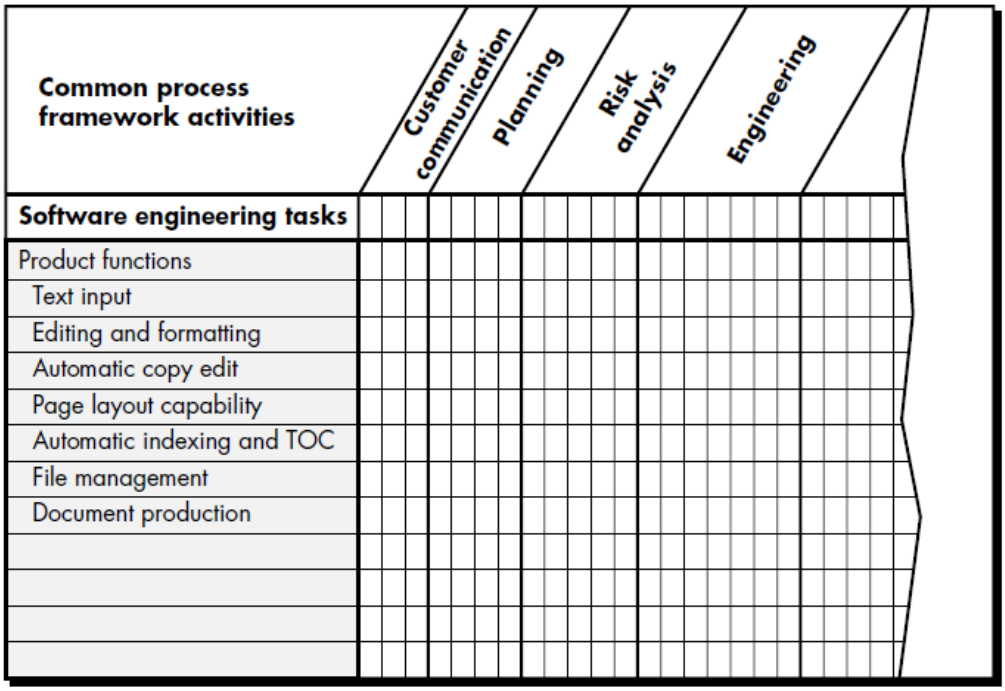


Fig: Melding the Product and the Process

5.3.3.2 Process Decomposition

- A software team should have a significant degree of flexibility in choosing the software engineering paradigm that is best for the project and the software engineering tasks that populate the process model once it is chosen.
- A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach.
- If very tight time constraints are imposed and the problem can be heavily compartmentalized, the RAD model is probably the right option. If the

deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best.

- Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, significant reuse potential) will lead to the selection of other process models.
- Process decomposition commences when the project manager asks, relatively simple project might require the following work tasks for the *customer communication* activity:

    1. Develop list of clarification issues.

    2. Meet with customer to address clarification issues.

    3. Jointly develop a statement of scope.

    4. Review the statement of scope with all concerned.

    5. Modify the statement of scope as required.

- These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

5.3.4 The Project

- We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity.

- In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring and controlling the project.
- In order to manage a successful software project, we must understand what can go wrong (so that problems can be avoided) and how to do it right.
- In an excellent paper on software projects, John Reel defines ten signs that indicate that an information systems project is in jeopardy:

  1. Software people don't understand their customer's needs.

  2. The product scope is poorly defined.

  3. Changes are managed poorly

  4. The chosen technology changes.

  5. Business needs change [or are ill-defined].

  6. Deadlines are unrealistic.

  7. Users are resistant.

  8. Sponsorship is lost [or was never properly obtained].

  9. The project team lacks people with appropriate skills.

  10. Managers [and practitioners] avoid best practices and lessons learned.

- Reel suggests a five-part commonsense approach to software projects:

•

1. Start on the right foot. This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objects and expectations for everyone who will be involved in the project

2. Maintain momentum. Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.

3. Track progress. For a software project, progress is tracked as work products (e.g., specifications, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity. In addition, software process and project measures can be collected and used to assess progress against averages developed for the software development organization.

4. Make smart decisions. In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).

5. Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

.

MODULE 6

6.1 PROJECT SCHEDULING AND TRACKING

- Eventhough technology has risen up so advanced, stillthe software development faces late delivery.

- Although there are many reasons why software is delivered late, most can betraced to one or more of the following root causes:

  (i) An unrealistic deadline established by someone outside the software teamand forced on managers and practitioners on the group.
  (ii) Changing customer requirements that are not reflected in schedulechanges.
  (iii) An honest underestimate of the amount of effort and/or the number of resourcesthat will be required to do the job.
  (iv) Predictable and/or unpredictable risks that were not considered whenthe project commenced.
  (v) Technical difficulties that could not have been foreseen in advance.
  (vi) Human difficulties that could not have been foreseen in advance.
  (vii) Miscommunication among project staff that results in delays.
  (viii) A failure by project management to recognize that the project is fallingbehind schedule and a lack of action to correct the problem.

6.1.1 Basic Principles

A number of basic principles guide software project scheduling:

•

(i) **Compartmentalization.** The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.

(ii) **Interdependency.** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Other activities can occur independently.

(iii) **Time allocation.** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

(iv) **Effort validation.** Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people has been scheduled at any given time.

(v) **Defined responsibilities.** Every task that is scheduled should be assigned to a specific team member.

(vi) **Defined outcomes.** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a module) or a part of a work product. Work products are often combined in deliverables.

(vii) **Defined milestones.** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

.

6.1.2 Relationship between People and Effort

- There is a common myth that is still believed by many managers who are responsible for software development work: "If we fall behind schedule, we can always add more programmers and catch up later in the project."
- Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further.
- The people who are added must learn the system, and the people who teach them are the same people who were doing the work.
- While teaching, no work is done, and the project falls further behind.
- Over the years, empirical data and theoretical analysis have demonstrated that project schedules are elastic. That is, it is possible to compress a desired project completion date to some extent and it is also possible to extend a completion date.
- The Putnam-Norden-Rayleigh (PNR) Curve 5 provides an indication of the relationship between effort applied and delivery time for a software project.
- A version of the curve, representing project effort as a function of delivery time, is shown in following figure.
- The curve indicates a minimum value t_0 that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended).
- As we move left of t_0(i.e., as we try to accelerate delivery), the curve rises nonlinearly.

•

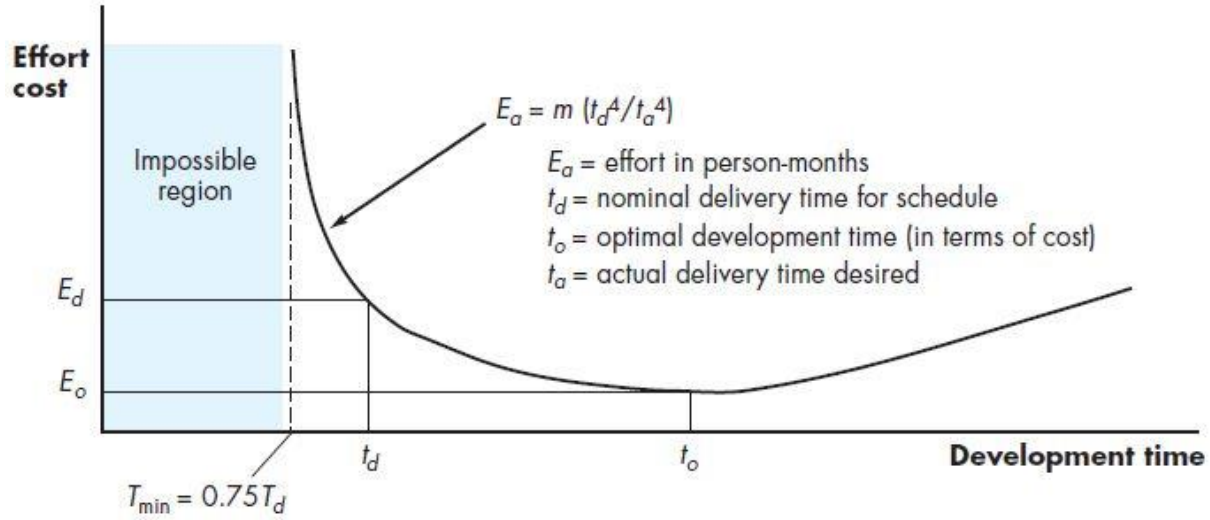


Fig: Relationship between People and Effort(PNR Graph)

- The number of delivered lines of code (source statements), L, is related to effort and development time by the equation:

  $L = P^3\ E^{1/3} t^{4/3}$

  where E is development effort in person-months, P is a productivity parameter that reflects a variety of factors that leads to high-quality software engineering work and t is the project duration in calendar months.
- Rearranging this software equation, we can arrive at an expression for development effort E:

  $E = L^3/P^3 t^4$

6.1.3 Defining a Task Set for the Software Project

•

- Regardless of the process model that is chosen, the work that a software team performs is achieved through a set of tasks that enable you to define, develop, and ultimately support computer software.
- No single task set is appropriate for all projects. The set of tasks that would be appropriate for a large, complex system would likely be perceived as overkill for a small, relatively simple software product.
- Therefore, an effective software process should define a collection of task sets, each designed to meet the needs of different types of projects.
- The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work.
- Although it is difficult to develop a comprehensive taxonomy of software project types, most software organizations encounter the following projects:
  1. **Concept development projects** that are initiated to explore some new business concept or application of some new technology.
  2. **New application development projects** that are undertaken as a consequence of a specific customer request.
  3. **Application enhancement projects** that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.
  4. **Application maintenance projects** that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.
  5. **Reengineering projects** that are undertaken with the intent of rebuildingan existing (legacy) system in whole or in part.

6.1.3.1A Task Set Example

•

- Concept development projects are initiated when the potential for some new technology must be explored.
- There is no certainty that the technology will be applicable,but a customer (e.g., marketing) believes that potential benefit exists.
- Concept development projects are approached by applying the following major tasks:

(a) **Concept scoping** determines the overall scope of the project.

(b) **Preliminary concept planning** establishes the organization's ability to undertake the work implied by the project scope.

(c) **Technology risk assessment** evaluates the risk associated with the technology to be implemented as part of the project scope.

(d) **Proof of concept** demonstrates the viability of a new technology in the software context.

(e) **Concept implementation** implements the concept representation in a manner that can be reviewed by a customer and is used for"marketing" purposes when a concept must be sold to other customers or management.

(f) **Customer reaction** to the concept solicits feedback on a new technology concept and targets specific customer applications.

6.1.3.2 Refinement of Major Tasks

- The major tasks (i.e., software engineering actions) described in the preceding section may be used to define a macroscopic schedule for a project.
- However,the macroscopic schedule must be refined to create a detailed project schedule.

•

- Refinement begins by taking each major task and decomposing it into a set ofsubtasks (with related work products and milestones).

6.2 SOFTWARE CONFIGURATION MANAGEMENT (SCM)

- Change is inevitable when computer software is built and can lead to confusion when you and other members of a software team are working on a project.
- Configuration management is the art of identifying,organizing, and controlling modifications to the software being built by a programmingteam.
- The goal is to maximize productivity by minimizing mistakes.
- SCM activities are developed to (1) identify change, (2) control change,(3) ensure that change is being properly implemented, and (4) report changes toothers who may have an interest.
- There are four fundamental sources of change:
  (i) New business or market conditions dictate changes in product requirements or business rules.
  (ii) New stakeholder needs demand modification of data produced by informationsystems, functionality delivered by products, or services deliveredby a computer-based system.
  (iii) Reorganization or business growth/downsizing causes changes in projectpriorities or software engineering team structure.

•

(iv) Budgetary or scheduling constraints cause a redefinition of the system orproduct.

6.2.1 Elements of SCM

Four important elements that should exist when a configuration management system is developed:

(a) Component elements —A set of tools coupled within a file managementsystem (e.g., a database) that enables access to and management of each software configuration item.

(b) Process elements —A collection of procedures and tasks that define aneffective approach to change management (and related activities) for allconstituencies involved in the management, engineering, and use of computersoftware.

(c) Construction elements —A set of tools that automate the construction ofsoftware by ensuring that the proper set of validated components (i.e., thecorrect version) have been assembled.

(d) Human elements —A set of tools and process features (encompassing otherCM elements) used by the software team to implement effective SCM.

6.2.2 Baseline

- A baseline is a software configuration management concept that helps you to control change without seriously impeding justifiable change.
- A specification or product that has been formally reviewed and agreed upon, thatthereafter serves as the basis for further development, and that can be changed onlythrough formal change control procedures.

•

- Before a software configuration item becomes a baseline, change may be madequickly and informally.
- However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

6.3 USER INTERFACE DESIGN

- User interface design creates an effective communication medium between a human and a computer.
- Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.
- A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

6.3.1 Rules of User Interface Design

There are three basic goals in User Interface Design commonly known as Golden Rules:

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

•

6.3.1.1 Place the User in Control

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface.

(i) **Define interaction modes in a way that does not force a user into unnecessary or undesired actions**. An interaction mode is the current state of theinterface. For example, if *spellcheck* is selected in a word-processor menu, thesoftware moves to a spell checking mode. The user should be able to enter and exit the mode with little orno effort.

(ii) **Provide for flexible interaction**. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, or voice recognition commands. But every action is not amenable to every interaction mechanism.

(iii) **Allow user interaction to be interruptible and undoable**. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

(iv) **Streamline interaction as skill levels advance and allow the interaction to be customized**. Users often find that they perform the same sequence of interactionsrepeatedly. It is worthwhile to design a "macro" mechanism that enables anadvanced user to customize the interface to facilitate interaction.

•

(v) **Hide technical internals from the casual user**. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

6.3.1.2 Reduce the User's Memory Load

- The more a user has to remember, the more error-prone will be the interaction with the system.
- It defines design principles that enable an interface to reduce the user's memory load:

(i) **Reduce demand on short-term memory**. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

(ii) **Establish meaningful defaults.**The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

(iii) **Define shortcuts that are intuitive**When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

•

(iv) **The visual layout of the interface should be based on a real world metaphor.**For example, a bill payment system should use a check book and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual clues.

(v) **Disclose information in a progressive fashion**. The interface should be organized hierarchically. An example, common to many word-processing applications, is the underlining function. The function itself is one of a number of functions under a text style menu. However, every underlining capability is not listed. The user must pick underlining, then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

6.3.1.3 Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to a design standard that is maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that are used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented. It defines a set of design principles that help make the interface consistent:

(i) **Allow the user to put the current task into a meaningful context.**Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of

•

the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

(ii) **Maintain consistency across a family of applications.**A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

(iii) **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.** Once a particularinteractive sequence has become a de facto standard (e.g., the use of alt-S tosave a file), the user expects this in every application he encounters. A change(e.g., using alt-S to invoke scaling) will cause confusion.

6.4 CASE Tools

- **Computer-aided software engineering** (CASE) tools assist software engineering managers and practitioners in every activity associated with the software process.
- They automate project management activities, manage all work products produced throughout the process, and assist engineers in their analysis, design, coding and test work.
- CASE tools can be integrated within a sophisticated environment.
- Software engineers now recognize that they need more and varied tools along with an organized and efficient workshop in which to place the tools.

•

- The workshop for software engineering has been called an *integrated project support environment* and the tools that fill the workshopare collectively called *computer-aided software engineering*.
- CASE provides the software engineer with the ability to automate manual activities and to improve engineering insight.

6.4.1 Building Blocks of CASE

- Computer aided software engineering can be as simple as a single tool that supports a specific software engineering activity or as complex as a complete "environment" that encompasses tools, a database, people, hardware, a network, operating systems, standards, and many other components.

- The building blocks for CASE are illustrated in the following figure:

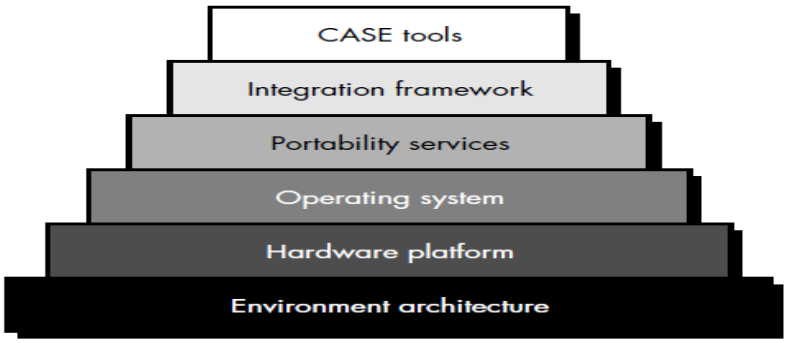


Fig: Building blocks of CASE

- Rather, successful environments for software engineering are built on an environment architecture that encompasses appropriate hardware and systems software.

•

- In addition, the environment architecture must consider the human work patterns that are applied during the software engineering process.
- A set of *portability services* provides a bridge between CASE tools and their integration framework and the environment architecture.
- The *integration framework* is a collection of specialized programs that enables individual CASE tools to communicate with one another, to create a project database, and to exhibit the same look and feel to the end-user (the software engineer).
- Portability services allow CASE tools and their integration framework to migrate across different hardware platforms and operating systems without significant adaptive maintenance.

6.4.2 Taxonomy of CASE Tools

- CASE tools can be classified by function, by their role as instruments for managers or technical people, by their use in the various steps of the software engineering process, by the environment architecture (hardwareand software) that supports them, or even by their origin or cost.
- The taxonomy presented here uses function as a primary criterion.

(i) **Business process engineering tools.** The primary objective for tools in this category is to represent business data objects, their relationships, and how these data objects flow between different business areas within a company.

(ii) **Process modeling and management tools.** Process modeling tools (also called *process technology* tools) are used to represent the key elements of

•

a process so that it can be better understood. Such tools can also provide links to process descriptions that help those involved in the process to understand the work tasks that are required to perform it. Process management tools provide links to other tools that provide support to defined process activities.

(iii) **Project planning tools.** Tools in this category focus on two primary areas: **software project effort and cost estimation and project scheduling**. Estimation tools compute estimated effort, project duration, and recommended number of people for a project.

(iv) **Risk analysis tools.** Identifying potential risks and developing a plan to mitigate, monitor, and manage them is of paramount importance in large projects. Risk analysis tools enable a project manager to build a risk table by providing detailed guidance in the identification and analysis of risks.

(v) **Project management tools.** The project schedule and project plan must be tracked and monitored on a continuing basis. In addition, a manager should use tools to collect metrics that will ultimately provide an indication of software product quality.

(vi) **Requirements tracing tools.** When large systems are developed, things "fall into the cracks." That is, the delivered system does not fully meet customer specified requirements. The objective of requirements tracing tools is to provide a systematic approach to the isolation of requirements, beginning with the customer request for proposal or specification.

(vii) **Metrics and management tools.** Software metrics improve a manager's ability to control and coordinate the software engineering process and a practitioner's ability to improve the quality of the software that is produced.

•

Management-oriented tools capture project specific metrics (e.g., LOC/person-month, defects per function point) that provide an overall indication of productivity or quality.

(viii) Documentation tools. Document production and desktop publishing tools support nearly every aspect of software engineering and represent a substantial "leverage" opportunity for all software developers. Most software development organizations spend a substantial amount of time developing documents, and in many cases the documentation process itself is quite inefficient. Documentation tools provide an important opportunity to improve productivity.

(ix) System software tools. CASE is a workstation technology. Therefore, the CASE environment must accommodate high-quality network system software, object management services, distributed component support, electronic mail, bulletin boards, and other communication capabilities.

(x) Quality assurance tools. The majority of CASE tools that claim to focus on quality assurance are actually metrics tools that audit source code to determine compliance with language standards. Other tools extract technical metrics in an effort to project the quality of the software that is being built.

(xi) Database management tools. Database management software serves as a foundation for the establishment of a CASE database (repository) that we have called the *project database. D*atabase management tools for CASE are evolving from relational database management systems to object oriented database management systems.

•

(xii) Software configuration management tools. Software configuration management lies at the kernel of every CASE environment. Tools can assist in all five major SCM tasks—**identification, version control, change control, auditing, and status accounting**. The CASE database provides a mechanism for identifying each configuration item and relating it to other items; the change control process can be implemented with the aid of specialized tools; easy access to individual configuration items facilitates the auditing process; and CASE communication tools can greatly improve status accounting (reporting information about changes to all who need to know).

(xiii) Analysis and design tools. Analysis and design tools enable a software engineer to create models of the system to be built. The models contain a representation of data, function, and behavior (at the analysis level) and characterizations of data, architectural, component-level, and interface design .

(xiv) PRO/SIM tools. PRO/SIM (prototyping and simulation) tools provide the software engineer with the ability to predict the behavior of a real-time system prior to the time that it is built.

(xv) Interface design and development tools. Interface design and development tools are actually a tool kit of software components (classes) such as menus, buttons, window structures, icons, scrolling mechanisms, device drivers, and so forth.

(xvi) Prototyping tools. A variety of different prototyping tools can be used. *Screen painters* enable a software engineer to define screen layout rapidly for interactiveapplications. More sophisticated CASE prototyping tools

•

enable the creation of a datadesign, coupled with both screen and report layouts.

(xvii) Programming tools. The programming tools category encompasses the compilers, editors, and debuggers that are available to support most conventional programming languages. In addition, object-oriented programming environments, fourth generation languages, graphical programming environments, application generators, and database query languages also reside within this category.

(xviii) Web development tools. The activities associated with Web engineering are supported by a variety of tools for WebApp development. These include tools that assist in the generation of text, graphics, forms, scripts, applets, and other elements of a Web page.

(xix) Integration and testing tools. In their directory of software testing tools, Software Quality Engineering defines the following testing tools categories:

- *Data acquisition*—tools that acquire data to be used during testing.
- *Static measurement*—tools that analyze source code without executing test cases.
- *Dynamic measurement*—tools that analyze source code during execution.
- *Simulation*—tools that simulate function of hardware or other externals.
- *Test management*—tools that assist in the planning, development, and control of testing.

•

- *Cross-functional tools*—tools that cross the bounds of the preceding categories.

(xx) Static analysis tools. Static testing tools assist the software engineer in deriving test cases. Three different types of static testing tools are used in the industry: **code based testing tools, specialized testing languages, and requirements-based testing tools**. ***Code-based testing tools***accept source code (or PDL) as input and perform a number of analyses that result in the generation of test cases.

(xxi) Dynamic analysis tools. Dynamic testing tools interact with an executing program, checking path coverage, testing assertions about the value of specific variables, and otherwise instrumenting the execution flow of the program.

(xxii) Test management tools. Test management tools are used to control and coordinate software testing for each of the major testing steps. Tools in this category manage and coordinate regression testing, perform comparisons that ascertain differences between actual and expected output.

(xxiii) Client/server testing tools. The c/s environment demands specialized testing tools that exercise the graphical user interface and the network communications requirements for client and server.

(xxiv) Reengineering tools. Tools for legacy software address a set of maintenance activities that currently absorb a significant percentage of all software-related effort. The reengineering tools category can be subdivided into the following functions:

- *Reverse engineering to specification tools* take source code as input and generate graphical structured analysis and design models, where-used lists, and other design information.
- *Code restructuring and analysis tools* analyze program syntax, generate a control flow graph, and automatically generate a structured program.
- *On-line system reengineering tools* are used to modify on-line database systems

•

CONTENT BEYOND SYLLABUS

Top 10 Testing Automation Tools for Software Testing

1. Selenium

Selenium is a testing framework to perform web application testing across various browsers and platforms like Windows, Mac, and Linux. Selenium helps the testers to write tests in various programming languages like Java, PHP, C#, Python, Groovy, Ruby, and Perl. It offers record and playback features to write tests without learning Selenium IDE.

Selenium proudly supports some of the largest, yet well-known browser vendors who make sure they have Selenium as a native part of their browser. Selenium is undoubtedly the base for most of the other software testing tools in general.

2. TestingWhiz

TestingWhiz is a test automation tool with the code-less scripting by Cygnet Infotech, a CMMi Level 3 IT solutions provider. TestingWhiz tool's Enterprise edition offers a complete package of various automated testing solutions like web testing, software testing, database testing, API testing, mobile app testing, regression test suite maintenance, optimization, and automation, and cross-browser testing.

TestingWhiz offers various important features like:

- Keyword-driven, data-driven testing, and distributed testing
- Browser Extension Testing
- Object Eye Internal Recorder
- SMTP Integration
- Integration with bug tracking tools like Jira, Mantis, TFS and FogBugz
- Integration with test management tools like HP Quality Center, Zephyr, TestRail, and Microsoft VSTS
- Centralized Object Repository
- Version Control System Integration
- Customized Recording Rule

•

3. HPE Unified Functional Testing (HP – UFT formerly QTP)

HP QuickTest Professional was renamed to HPE Unified Functional Testing. HPE UFT offers testing automation for functional and regression testing for software applications.

Visual Basic Scripting Edition scripting language is used by this tool to register the test processes and operates the various objects and controls in testing the applications.

QTP offers various features like:

- Integration with Mercury Business Process Testing and Mercury Quality Center
- Unique Smart Object Recognition
- Error handling mechanism
- Creation of parameters for objects, checkpoints, and data-driven tables
- Automated documentation

4. TestComplete

TestComplete is a functional testing platform that offers various solutions to automate testing for desktop, web, and mobile applications by SmartBear Software.

TestComplete offers the following features:

- GUI testing
- Scripting Language Support – JavaScript, Python, VBScript, JScript, DelphiScript, C++Script & C#Script
- Test visualizer
- Scripted testing
- Test recording and playback

5. Ranorex

Ranorex Studio offers various testing automation tools that cover testing all desktop, web, and mobile applications.

Ranorex offers the following features:

- GUI recognition

•

- Reusable test codes
- Bug detection
- Integration with various tools
- Record and playback

6. Sahi

Sahi is a testing automation tool to automate web applications testing. The open-source Sahi is written in Java and JavaScript programming languages.

Sahi provides the following features:

- Performs multi-browser testing
- Supports ExtJS, ZK, Dojo, YUI, etc. frameworks
- Record and playback on the browser testing

7. Watir

Watir is an open-source testing tool made up of Ruby libraries to automate web application testing. It is pronounced as “water.”

Watir offers the following features:

- Tests any language-based web application
- Cross-browser testing
- Compatible with business-driven development tools like RSpec, Cucumber, and Test/Unit
- Tests web page’s buttons, forms, links, and their responses

8. Tosca Testsuite

Tosca Testsuite by Tricentis uses model-based test automation to automate software testing.

Tosca Testsuite comes with the following capabilities:

- Plan and design test case
- Test data provisioning
- Service virtualization network
- Tests mobile apps
- Integration management

•

- Risk coverage

9. Telerik TestStudio

Telerik TestStudio offers one solution to automate desktop, web, and mobile application testing including UI, load, and performance testing.

Telerik TestStudio offers various compatibilities like:

- Support of programming languages like HTML, AJAX, ASP.NET, JavaScript, Silverlight, WPF, and MVC
- Integration with Visual Basic Studio 2010 and 2012
- Record and playback
- Cross-browser testing
- Manual testing
- Integration with bug tracking tools

10. Katalon Studio

Katalon Studio is a free automation testing solution developed by Katalon LLC. The software is built on top of the open-source automation frameworks Selenium, Appium with a specialized IDE interface for API, web and mobile testing. This tool includes a full package of powerful features that help overcome common challenges in web UI test automation.

Katalon Studio consists of the following features:

- Built-in object repository, XPath, object re-identification
- Supports Java/Groovy scripting languages
- Built-in support for Image-based testing
- Support Continuous Integration tools like Jenkins & TeamCity
- Supports Duel-editor Interface
- Customizable execution workflow