Paper: Software Engineering **Paper Code: BCA 208**

Aim

To understand the importance, limitations and challenges of processes involved in Software development.

Objectives

To gain knowledge of various software models.
To gain knowledge of various software design activities.
To learn cost estimation, software testing, Maintenance and debugging.

INSTRUCTIONS TO PAPER SETTERS: Maximum Marks: 75

1. Question No. 1 should be compulsory and cover the entire syllabus. This question should have objective or short answer type questions. It should be of 25 marks.
2. Apart from Question No. 1, rest of the paper shall consist of four units as per the syllabus. Every unit should have two questions. However, student may be asked to attempt only 1 question from each unit. Each question should be 12.5 marks.

UNIT – I

Introduction: Software Crisis, Software Processes & Characteristics, Software life cycle models, Waterfall, Prototype, Evolutionary and Spiral Models
Software Requirements analysis & specifications: Requirement engineering, requirement elicitation techniques like FAST, QFD, Requirements analysis using DFD(with case studies), Data dictionaries & ER Diagrams, Requirements documentation, Nature of SRS, Characteristics & organization of SRS.[T1][T2] [T3] **[No. of Hrs.: 12]**

UNIT – II

Software Project Management Concepts: The Management spectrum, The People, The Problem, The Process, The Project.
Software Project Planning: Size Estimation like lines of Code & Function Count, Cost Estimation Models, COCOMO, Risk Management.[T1][T2][T3] **[No. of Hrs.: 10]**

UNIT - III

Software Design: Cohesion & Coupling, Classification of Cohesiveness & Coupling, Layered arrangement of modules, Function Oriented Design, Object Oriented Design[T1][T2]
Software Metrics: Software measurements: What & Why, Token Count, Halstead Software Science Measures, Design Metrics, Data Structure Metrics.[T1][T2] **[No. of Hrs.: 10]**

UNIT - IV

Software Testing: Code Review, Testing Process, Types of Testing, Functional Testing, Structural Testing, Test Activities, Unit Testing, Integration Testing and System Testing(Performance Testing and Error Seeding), Debugging Activities. [T1][T2][R1]
Software Maintenance: Management of Maintenance, Maintenance Process, Reverse Engineering, Software Re-engineering, Configuration Management, Documentation.[T1][T3]
[No. of Hrs.: 12]

Note : A Minimum of 40 Lectures is mandatory for each course.

Syllabus of Bachelor of Computer Applications (BCA), approved by BCA Coordination Committee on 26th July 2011 & Sub-
Committee Academic Council held 28th July 2011. W.e.f. academic session 2011-12

TEXT Books:

[T1] K. K. Aggarwal & Yogesh Singh, “Software Engineering”, 2nd Ed., New Age International, 2005.
[T2] Rajib Mall, “Fundamental of Software Engineering”, 3rd Edition, PHI Learning Private Limited
[T3] I. Sommerville, “Software Engineering”, 9th Edition, Pearson Edu.

REFERENCE:

[R1] Jibitesh Mishra and Ashok Mohanty, “Software Engineering”, Pearson
[R2] R. S. Pressman, “Software Engineering – A practitioner’s approach”, 5th Ed., McGraw Hill Int. Ed., 2001.
[R3] James Peter, W. Pedrycz, “Software Engineering: An Engineering Approach”, John Wiley & Sons.

Unit 1
Software Engineering

Software Processes & Characteristics

- The roadmap to building high quality software products is software process.
- Software processes are adapted to meet the needs of software engineers and managers as they undertake the development of a software product.
- A software process provides a framework for managing activities that can very easily get out of control.
- Different projects require different software processes.
- The software engineer's work products (programs, documentation, data) are produced as consequences of the activities defined by the software process.
- The best indicators of how well a software process has worked are the quality, timeliness, and long-term viability of the resulting software product.

Software Engineering

- Software engineering encompasses a process, management techniques, technical methods, and the use of tools.

Generic Software Engineering Phases

- Definition phase - focuses on what (information engineering, software project planning, requirements analysis).
- Development phase - focuses on how (software design, code generation, software testing).
- Support phase - focuses on change (corrective maintenance, adaptive maintenance, perfective maintenance, preventative maintenance).

Requirements of Software Engineering

- Requirements elicitation (find out from customers what the product objectives are, what is to be done, how the product fits into business needs, and how the product is used on a day to day basis)
- Requirements analysis and negotiation (requirements are categorized and organized into subsets, relations among requirements identified, requirements reviewed for correctness, requirements prioritized based on customer needs)
- Requirements specification (work product produced describing the function, performance, and development constraints for a computer-based system)
- System modeling (system representation that shows relationships among the system components)
- Requirements validation (examines the specification to ensure requirement quality and that work products conform to agreed upon standards)

- Requirements management (set of activities that help project team to identify, control, and track requirements and changes as project proceeds)

Software Crisis

- Software failures receive a lot more publicity than software engineering success stories.
- The software crisis predicted thirty years ago has never materialized and software engineering successes outnumber the failures.
- The problems that afflict software development are associated more with how to develop and support software properly, than with simply building software that functions correctly.

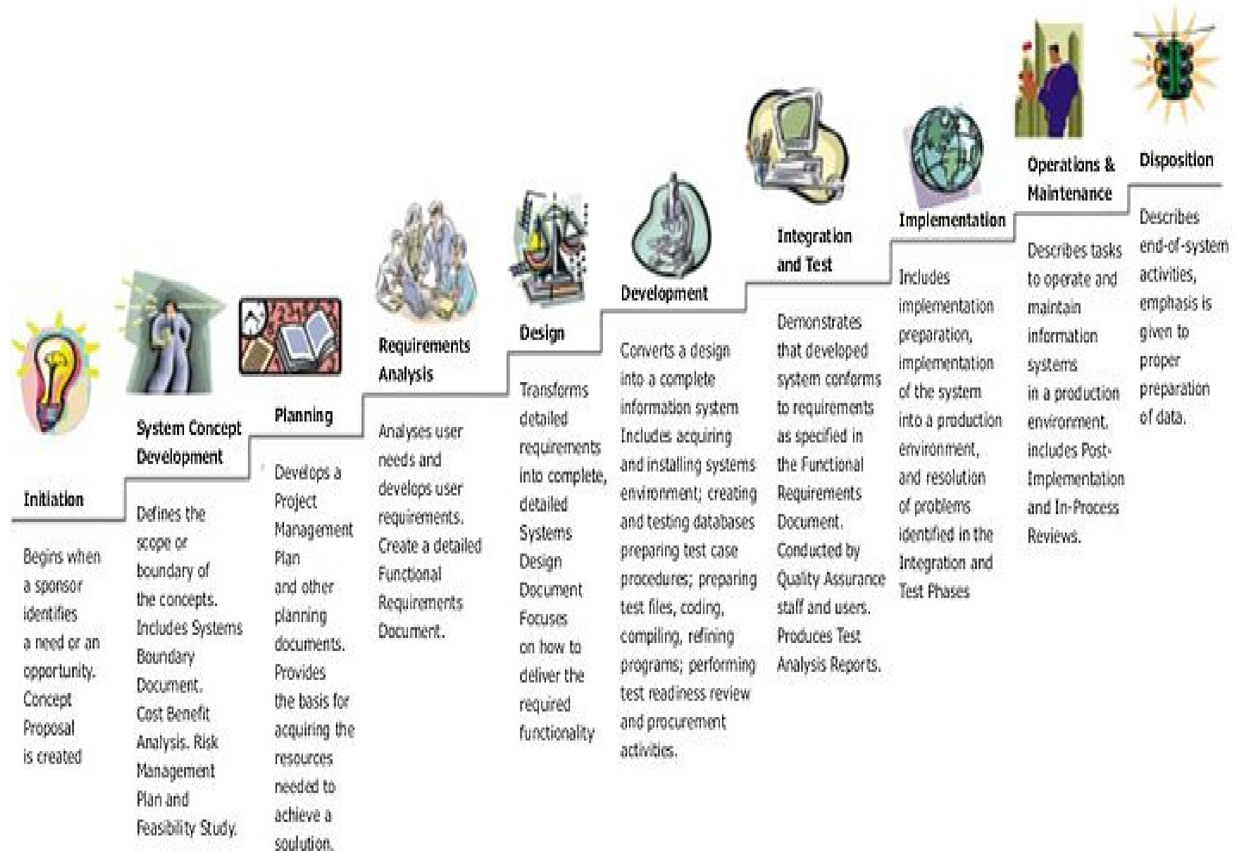
The System Development Life Cycle

- **Systems analysis, requirements definition**: Defines project goals into defined functions and operation of the intended application. Analyzes end-user information needs.
- **Systems design**: Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudo code and other documentation.
- **Implementation**: The real code is written here.
- **Integration and testing**: Brings all the pieces together into a special testing environment, then checks for errors, bugs and interoperability.
- **Acceptance, installation, deployment**: The final stage of initial development, where the software is put into production and runs actual business.
- **Maintenance**: What happens during the rest of the software's life: changes, correction, additions, moves to a different computing platform and more. This is often the longest of the stages.

In the following example (see picture) these stage of the systems development life cycle are divided in ten steps from definition to creation and modification of IT work products:

Systems Development Life Cycle (SDLC)

Life-Cycle Phases



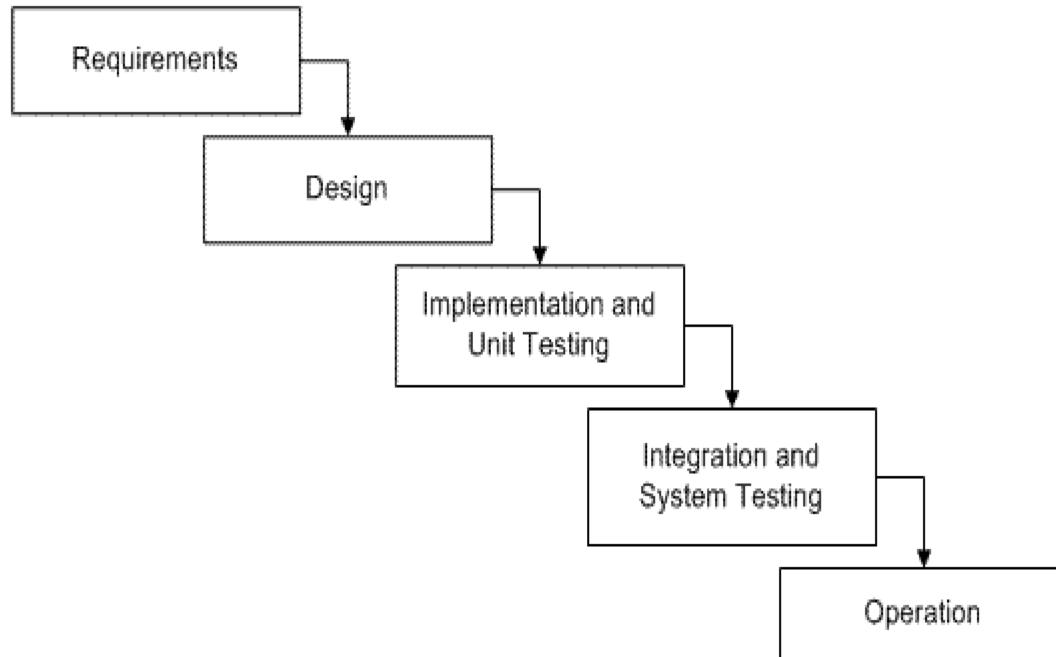
Strength and Weaknesses of SDLC	
Strengths	**Weaknesses**
Control.	Increased development time.
Monitor large projects.	Increased development cost.
Detailed steps.	Systems must be

	defined up front.
Evaluate costs and completion targets.	Rigidity.
Documentation.	Hard to estimate costs, project overruns.
Well defined user input.	User input is sometimes limited.
Ease of maintenance.	
Development and design standards.	
Tolerates changes in MIS staffing.	

Software Life Cycle Model

1. **Waterfall Model**

This is the most common and classic of life cycle models, also referred to as a linear-sequential life cycle model. It is very simple to understand and use. In a waterfall model, each phase must be completed in its entirety before the next phase can begin. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. Unlike what I mentioned in the general model, phases do not overlap in a waterfall model.



Waterfall Life Cycle Model

Advantages

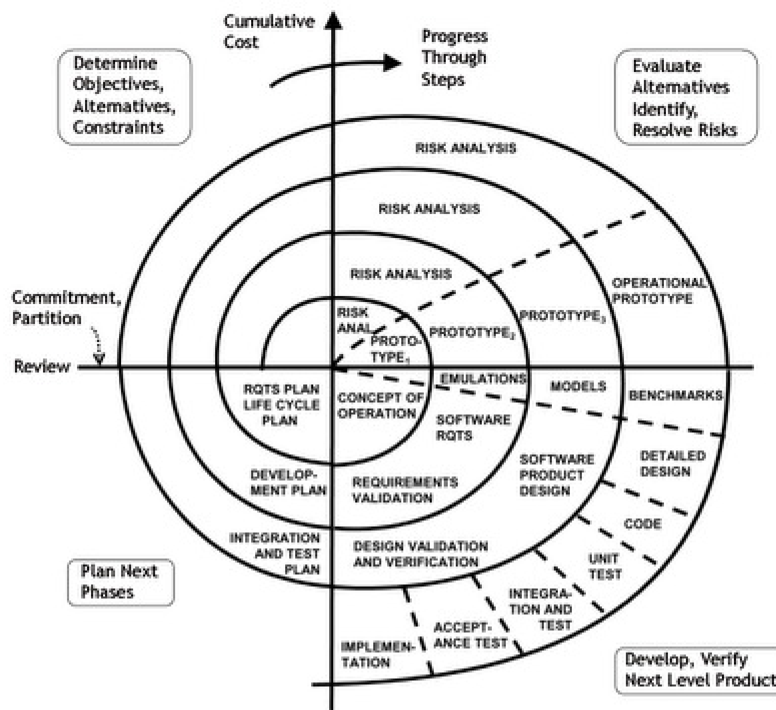
- Simple and easy to use.
- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Disadvantages

- Adjusting scope during the life cycle can kill a project
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing.

Spiral Model

The spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed. Each subsequent spirals builds on the baseline spiral. Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase. Software is produced in the engineering phase, along with testing at the end of the phase. The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral. In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.



Spiral Life Cycle Model

Advantages

- High amount of risk analysis
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.

Disadvantages

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

Software Quality Assurance

Meaning of Software Quality

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

- Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product;

- Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

Structural quality is evaluated through the analysis of the software inner structure, its source code, in effect how its architecture adheres to sound principles of software architecture. In contrast, functional quality is typically enforced and measured through software testing.

According to the IEEE Software quality is:
(1) The degree to which a system, component, or process meets specified requirements.
(2) The degree to which a system, component, or process meets customer or user needs or expectations.

Factor of Quality Assurance (McCall's Quality Factors)

- Product Operation
  - Correctness
  - Efficiency
  - Integrity
  - Reliability
  - Usability

- Product Revision

- Flexibility
- Maintainability
- Testability

- Product Transition
  - Interoperability
  - Portability
  - Reusability

REQUIREMENT ENGINEERING

Requirements engineering (RE) refers to the process of formulating, documenting and maintaining software requirements and to the subfield of Software Engineering concerned with this process.

The first use of the term 'requirements engineering' was probably in 1979 in a TRW technical report but did not come into general use until the 1990s with the publication of an IEEE Computer Society tutorial and the establishment of a conference series on requirements engineering.

In the waterfall model requirements engineering is presented as the first phase of the development process. Later software development methods, including the Rational Unified Process, Extreme Programming and Scrum assume that requirements engineering continues through the lifetime of a system.

The activities involved in requirements engineering vary widely, depending on the type of system being developed and the specific practices of the organization(s) involved. These may include:

1. Requirements inception -
2. Requirements identification - identifying new requirements
3. Requirements analysis and negotiation - checking requirements and resolving stakeholder conflicts
4. Requirements specification (Software Requirements Specification)- documenting the requirements in a requirements document
5. System modeling - deriving models of the system, often using a notation such as the Unified Modeling Language
6. Requirements validation - checking that the documented requirements and models are consistent and meet stakeholder needs
7. Requirements management - managing changes to the requirements as the system is developed and put into use

REQUIREMENT ELICITATION TECHNIQUES LIKE FAST, QFD,

Elicitation of requirements is the first step to requirement engineering and forms a very crucial part of the entire process. The information collected during requirement elicitation has to go through the usual steps of interpretation, modeling, analysis and validation before the engineer is confident that the collected data for requirement is enough to work with. Thus, one can say that, requirement elicitation is very closely connected with the process of requirement engineering. One of the primary goals of requirement elicitation is to find out what problems exist and need to be resolved. It is thus, engaged in identifying system boundaries. These boundaries on the other hand, define where the final delivered system will fit in with respect to the current operational environment. Identifying boundaries is very important because the boundaries of a system can affect all subsequent efforts of elicitation. Right or, wrong selection of boundaries can affect identification of stake holders, identification of user classes, identification of goals, tasks, scenario etc. It is highly important to elicit goals in the early development process. While it is crucial to elicit the high level goals in the earliest, elicitation of the lower level goals continue throughout as the development takes place. The business goals can be considered the high level goals, while the lower level goals consist of technical goals. Elicitation of requirements sometimes, becomes very difficult because of the inability of the users to articulate their requirements. Requirement elicitation techniques are therefore used to make the process better.

Quality function deployment (QFD), also commonly known as a way to represent the "voice of the customer," is a process for capturing customer requirements and translating them into requirements that can be used by designers, producers, and suppliers. As with many other topics mentioned in this chapter, there are numerous books, articles, consulting services, software, and Web sites devoted to QFD; it will be discussed further in this book in Chapter 30, "Software Quality Assurance." It deserves mention in the context of requirements elicitation because the very first step in QFD is to "identify the customer's vital requirements for the product and translate them into design requirements." At this point in our software life cycle, we are not ready to translate customer requirements into design requirements—that comes later, after the SRS is written and analysis modeling occurs—but we are very interested in all methods of stakeholder requirements elicitation and capture.

The process of QFD can be found within methods of brainstorming, FAST, and JAD. With QFD, sharing of information is achieved through the efforts of a cross-functional team from various stakeholder groups such as marketing, sales, service, distribution, product engineering, process engineering, procurement, production, and of course, the end-user of the software system. A second QFD characteristic found throughout the requirements elicitation process has to do with capturing the requirements information in one place, in a compact form. Lastly, with QFD and other requirements elicitation methods, there is support for consensus and decision making, especially when complex relationships and trade-offs are involved to achieve the best overall solution. Such support is imperative, as we almost always deal with conflicting requirements when all stakeholders are represented.

DATA DICTIONARIES & ER DIAGRAMS

Dictionaries
Data Dictionaries are simply repositories to store information about all data items defined in DFD.
Includes :
Name of data item
Aliases (other names for items)

Description/Purpose
Related data items
Range of values
Data flows

Data structure definition

DATA FLOW DIAGRAMS

DFD show the flow of data through the system.
--All names should be unique
-- It is not a flow chart
-- Suppress logical decisions
-- Defer error conditions & handling until the end of
the analysis
Symbol Name Function
Requirements Analysis
Data Flow Connect process
Process Perform some transformation of its

input data to yield output data

ENTITY-RELATIONSHIP DIAGRAMS

It is a detailed logical representation of data for an organization and uses three main constructs.
Entities Relationships Attributes
Entities
Fundamental thing about which data may be maintained. Each entity has its own identity. Entity Type is the description of all entities to which a common definition and common relationships and attributes

REQUIREMENTS DOCUMENTATION

This is the way of representing requirements in a consistent format
SRS serves many purpose depending upon who is writing it.
-- written by customer
-- written by developer
Serves as contract between customer & developer.

NATURE OF SRS, CHARACTERISTICS & ORGANIZATION OF SRS

Nature of SRS
Basic Issues
• Functionality

- External Interfaces
- Performance
- Attributes
- Design constraints imposed on an Implementation.

SRS Should
-- Correctly define all requirements
-- not describe any design details
-- not impose any additional constraints
Characteristics of a good SRS
An SRS Should be
_ Correct
_ Unambiguous
_ Complete
_ Consistent

REQUIREMENTS DOCUMENTATION

Correct
An SRS is correct if and only if every requirement stated therein is one that the software shall meet.
Unambiguous
An SRS is unambiguous if and only if, every requirement stated therein has only one interpretation.
Complete
An SRS is complete if and only if, it includes the following elements
(i) All significant requirements, whether related to functionality, performance, design constraints, attributes or external interfaces.
(ii) Responses to both valid & invalid inputs.
(iii) Full Label and references to all figures, tables and diagrams in the SRS and definition of all terms and units of measure.
Consistent
An SRS is consistent if and only if, no subset of individual requirements described in it conflict.

ORGANIZATION OF THE SRS

IEEE has published guidelines and standards to organize an
SRS.
First two sections are same. The specific tailoring occurs in section-3.
1. Introduction
1.1 Purpose
1.2 Scope
1.3 Definition, Acronyms and abbreviations
1.4 References
1.5 Overview
2. The Overall Description

2.1 Product Perspective
2.1.1 System Interfaces
2.1.2 Interfaces
2.1.3 Hardware Interfaces
2.1.4 Software Interfaces
2.1.5 Communication Interfaces
2.1.6 Memory Constraints
2.1.7 Operations
2.1.8 Site Adaptation Requirements

2.2 Product Functions
2.3 User Characteristics
2.4 Constraints
2.5 Assumptions for dependencies
2.6 Apportioning of requirements
3. Specific Requirements
3.1 External Interfaces
3.2 Functions
3.3 Performance requirements
3.4 Logical database requirements
3.5 Design Constraints
3.6 Software System attributes
3.7 Organization of specific requirements
3.8 Additional Comments.

Software prototyping refers to the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed. It is an activity that can occur in software development and is comparable to prototyping as known from other fields, such as mechanical engineering or manufacturing.

A prototype typically simulates only a few aspects of, and may be completely different from, the final product.

Prototyping has several benefits: The software designer and implementer can get valuable feedback from the users early in the project. The client and the contractor can compare if the software made matches the software specification, according to which the software program is built. It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met. The degree of completeness and the techniques used in the prototyping have been in development and debate since its proposal in the early 1970s.

The Evolution/ process of prototyping involves the following steps

1. Identify basic requirements

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

2. Develop Initial Prototype

   The initial prototype is developed that includes only user interfaces. (See Horizontal Prototype, below)

3. Review

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

4. Revise and Enhance the Prototype

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

Advantages of prototyping

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

Reduced time and costs: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of what the user really wants can result in faster and less expensive software.

Improved and increased user involvement: Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications.The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality. The final product is more likely to satisfy the users desire for look, feel and performance.

Disadvantages of prototyping

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

Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.

User confusion of prototype and finished system: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error-checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to conflict.

Developer misunderstanding of user objectives: Developers may assume that users share their objectives (e.g. to deliver core functionality on time and within budget), without understanding wider commercial issues. For example, user representatives attending Enterprise software (e.g. PeopleSoft) events may have seen demonstrations of "transaction auditing" (where changes are logged and displayed in a difference grid view) without being told that this feature demands additional coding and often requires more hardware to handle extra database accesses. Users might believe they can demand auditing on every field, whereas developers might think this is feature creep because they have made assumptions about the extent of user requirements. If the developer has committed delivery before the user requirements were reviewed, developers are between a rock and a hard place, particularly if user management derives some advantage from their failure to implement requirements.

Developer attachment to prototype: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. (This may suggest that throwaway prototyping, rather than evolutionary prototyping, should be used.)

Excessive development time of the prototype: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.

Expense of implementing prototyping: the start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling, or both. Many companies tend to just jump into the prototyping

without bothering to retrain their workers as much as they should.

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

UNIT-II

SOFTWARE PROJECT MANAGEMENT CONCEPTS

MANAGEMENT SPECTRUM:

People: who deals with the software
Product: which were developed
Process: the way through which the product is developed.
Project: combination of people, process and project to develop the software.

Requirements Analysis

- Software engineering task that bridges the gap between system level requirements engineering and software design.
- Provides software designer with a representation of system information, function, and behavior that can be translated to data, architectural, and component-level designs.
- Expect to do a little bit of design during analysis and a little bit of analysis during design.

Communication Techniques

- Interview

An interview is a conversation between two people (the interviewer and the interviewee) where questions are asked by the interviewer to obtain information from the interviewee The qualitative research interview seeks to describe and the meanings of central themes in the life world of the subjects. The main task in interviewing is to understand the meaning of what the interviewees say

- Brainstorming

Brainstorming is a group creativity technique by which a group tries to find a solution for a specific problem by gathering a list of ideas spontaneously contributed by its members. The term was popularized by Alex Faickney Osborn in the 1953 book Applied Imagination. Osborn claimed that brainstorming was more effective than individuals working alone in generating ideas, although more recent research has questioned this conclusion.

Analysis Principles

- The information domain of the problem must be represented and understood.
- The functions that the software is to perform must be defined.
- Software behavior must be represented.
- Models depicting information, function, and behavior must be partitioned in a hierarchical manner that uncovers detail.
- The analysis process should move from the essential information toward implementation detail.

Software Prototyping

- Throwaway prototyping (prototype only used as a demonstration of product requirements, finished software is engineered using another paradigm)
- Evolutionary prototyping (prototype is refined to build the finished system)
- Customer resources must be committed to evaluation and refinement of the prototype.
- Customer must be capable of making requirements decisions in a timely manner.

Software Requirements Specification ('SRS)

- It is a requirements specification for a software system – is a complete description of the behavior of a system to be developed. It includes a set of use cases that describe all the interactions the users will have with the software.
- In addition to use cases, the SRS also contains non-functional requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance engineering requirements, quality standards, or design constraints).
- Software requirements is a sub-field of software engineering that deals with the elicitation, analysis, specification, and validation of requirements for software.[1]
- The software requirement specification (SRS) document enlists all necessary requirements for project development. To derive the requirements we need to have clear and thorough understanding of the products to be developed. This is prepared after detailed communications with project team and the customer. A general organization of an SRS is as follows.
- Introduction
  - Purpose
  - Scope
  - Definitions
  - System Overview
  - References
- Overall Description
  - Product Perspective
  - Product Functions
  - User Characteristics
  - Constraints, Assumptions and Dependencies
- Specific Requirements
  - External interface requirements

- Functional requirements
- Performance requirements
- Design constraints
- Logical database requirement
- Software System attributes
- other requirements

Analysis modeling

Analysis Model Elements

- Data dictionary - contains the descriptions of all data objects consumed or produced by the software
- Entity relationship diagram (ERD) - depicts relationships between data objects
- Data flow diagram (DFD) - provides an indication of how data are transformed as they move through the system; also depicts functions that transform the data flow (a function is represented in a DFD using a process specification or PSPEC)
- State transition diagram (STD) - indicates how the system behaves as a consequence of external events, states are used to represent behavior modes. Arcs are labeled with the events triggering the transitions from one state to another (control information is contained in control specification or CSPEC)

Data Modeling Elements (ERD)

- Data object - any person, organization, device, or software product that produces or consumes information
- Attributes - name a data object instance, describe its characteristics, or make reference to another data object
- Relationships - indicate the manner in which data objects are connected to one another

Functional Modeling and Information Flow (DFD)

- Shows the relationships of external entities, process or transforms, data items, and data stores
- DFD's cannot show procedural detail (e.g. conditionals or loops) only the flow of data through the software
- Refinement from one DFD level to the next should follow approximately a 1:5 ratio (this ratio will reduce as the refinement proceeds)
- To model real-time systems, structured analysis notation must be available for time continuous data and event processing (e.g. Ward and Mellor or Hately and Pirbhai)

Data Dictionary Contents

- Name - primary name for each data or control item, data store, or external entity
- Alias - alternate names for each data object
- Where-used/how-used - a listing of processes that use the data or control item and how it is used (e.g. input to process, output from process, as a store, as an external entity)
- Content description - notation for representing content
- Supplementary information - other data type information, preset values, restrictions, limitations, etc.

SOFTWARE PROJECT PLANNING

In order to conduct a successful software project, we
must understand:
_ Scope of work to be done Software Project Planning
_ The risk to be incurred
_ The resources required
_ The task to be accomplished
_ The cost to be expended

_ The schedule to be followed

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired. Software
Project Planning
Size estimation
Cost estimation Development time
Resources
requirements
Project
Scheduling
If LOC is simply a count of the number of lines then When comments and blank lines are ignored, the program

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declaration, and executable and non-executable statements".

FUNCTION COUNT

Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem.

RISK MANAGEMENT

Risk analysis involves examining how project outcomes might change with modification of risk input variables. Risk prioritization focus for severe risks.

Risk exposure: It is the product of the probability of incurring a loss due to the risk and the potential magnitude of that loss. Risk Management Planning produces a plan for dealing with each significant risks.

Risk Control

_ Record decision in the plan.

Risk resolution is the execution of the plans of dealing with each risk.

COST ESTIMATION MODELS:

CONSTRUCTIVE COST MODEL (COCOMO)

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

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

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)

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) = ab(KLOC)bb

Development Time (D) = cb(Effort Applied)db [months]

People required (P) = Effort Applied / Development Time [count]

where, KLOC is the estimated number of delivered lines (expressed in thousands) of code for project.
The coefficients ab, bb, cb and db are given in the following table:

Software project	ab	bb	cb	Db
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

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

Intermediate COCOMOs

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:-

- Product attributes
    - Required software reliability
    - Size of application database
    - Complexity of the product
- Hardware attributes
    - Run-time performance constraints
    - Memory constraints
    - Volatility of the virtual machine environment
    - Required turnabout time
- Personnel attributes
    - Analyst capability
    - Software engineering capability
    - Applications experience
    - Virtual machine experience

- Programming language experience
- Project attributes
    - Use of software tools
    - Application of software engineering methods
    - Required development schedule

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

Detailed COCOMO

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

The detailed model uses different effort multipliers for each cost driver attribute. These Phase Sensitive effort multipliers are each to determine the amount of effort required to complete each phase.

In detailed COCOMO, the effort is calculated as function of program size and a set of cost drivers given according to each phase of software life cycle.

A Detailed project schedule is never static.

The five phases of detailed COCOMO are:-

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

UNIT-III

DESIGN CONCEPTS AND PRINCIPLES & DESIGN METHODS:

A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria. During the design process the software requirements model is transformed into design models that describe the details of the data structures, system architecture, interface, and components. Each design product is reviewed for quality before moving to the next phase of software

development.Design Principles

The design process : A design should

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

The design principles

- process should not suffer from tunnel vision
- should be traceable to the analysis model
- should not reinvent the wheel
- should minimize intellectual distance between the software and the problem as it exists in the real world
- should exhibit uniformity and integration
- should be structured to accommodate change
- should be structured to degrade gently, even with bad data, events, or operating conditions are encountered
- should be assessed for quality as it is being created
- should be reviewed to minimize conceptual (semantic) errors

The Design Concepts

- Abstraction - allows designers to focus on solving a problem without being concerned about irrelevant lower level details (procedural abstraction - named sequence of events, data abstraction - named collection of data objects)
- Refinement - process of elaboration where the designer provides successively more detail for each design component
- Modularity - the degree to which software can be understood by examining its components independently of one another
- Software architecture - overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
- Control hierarchy or program structure - represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e.g. event sequences)
- Structural partitioning - horizontal partitioning defines three partitions (input, data transformations, and output); vertical partitioning (factoring) distributes control in a top-down

manner (control decisions in top level modules and processing work in the lower level modules)
- Data structure - representation of the logical relationship among individual data elements (requires at least as much attention as algorithm design)
- Software procedure - precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure)
- Information hiding - information (data and procedure) contained within a module is inaccessible to modules that have no need for such information

Effective Modular Design Method Evaluation Criteria

- Modular decomposability - provides systematic means for breaking problem into sub problems
- Modular compos ability - supports reuse of existing modules in new systems
- Modular understandability - module can be understood as a stand-alone unit
- Modular continuity - side-effects due to module changes minimized
- Modular protection - side-effects due to processing errors minimized

Design Heuristics for Effective Modularity

- Evaluate the first iteration of the program structure to reduce coupling and improve cohesion.
- Attempt to minimize structures with high fan-out; strive for fan-in as structure depth increases.
- Keep the scope of effect of a module within the scope of control for that module.
- Evaluate module interfaces to reduce complexity, reduce redundancy, and improve consistency.
- Define modules whose function is predictable and not overly restrictive (e.g. a module that only implements a single sub function).
- Strive for controlled entry modules, avoid pathological connection (e.g. branches into the middle of another module)

The design model

- The design model builds on the analysis model by describing, in greater detail, the structure of the system and how the system will be implemented. Classes that were identified in the analysis model are refined to include the implementation constructs.

- The design model is based on the analysis and architectural requirements of the system. It represents the application components and determines their appropriate placement and use within the overall architecture.
- In the design model, packages contain the design elements of the system, such as design classes, interfaces, and design subsystems, that evolve from the analysis classes. Each package can contain any number of sub packages that further partition the contained design elements. These architectural layers form the basis for a second-level organization of the elements that describe the specifications and implementation details of the system.

Design Documentation

- Design document is a way for you to communicate to others what your design decisions are and why

your decisions are good decisions. Don't worry if your design is not UML compliant and don't worry if you didn't use a special modeling tool to create it. The biggest factor that determines if your design document is good is whether or not it clearly explains your intentions.

- This presents a problem, however. In order to convey design decisions, you have to consider the audience that you are writing for. A peer developer will understand why a well-crafted class abstraction is a good design, however your manager will probably not. Because your peer developers and your manager have different concepts of what makes a design good, there is a need for two design documents; one for peer developers and one for managers. Each document serves a different and equally valuable purpose as you begin your project development.

COHESION & COUPLING AND THEIR CLASSIFICATION

- Cohesion and Coupling deal with the quality of an OO design. Generally, good OO design calls for loose coupling and high cohesion. The goals of OO designs are to make the application
  - Easy to Create
  - Easy to Maintain
  - Easy to Enhance

Coupling:

- Coupling is the degree to which one class knows about another class. Let us consider two classes class A and class B. If class A knows class B through its interface only i.e it interacts with class B through its API then class A and class B are said to be loosely coupled.
- If on the other hand class A apart from interacting class B by means of its interface also interacts through the non-interface stuff of class B then they are said to be tightly coupled. Suppose the developer changes the class B's non-interface part i.e non API stuff then in case of loose coupling class A does not breakdown but tight coupling causes the class A to break.
- So its always a good OO design principle to use loose coupling between the classes i.e all interactions between the objects in OO system should use the APIs. An aspect of good class and API design is that classes should be well encapsulated.

Cohesion:

- Cohesion is used to indicate the degree to which a class has a single, well-focused purpose. Coupling is all about how classes interact with each other, on the other hand cohesion focuses on

how single class is designed. Higher the cohesiveness of the class, better is the OO design.

Benefits of Higher Cohesion:

- Highly cohesive classes are much easier to maintain and less frequently changed.
- Such classes are more usable than others as they are designed with a well-focused purpose.

FUNCTION ORIENTED DESIGN

Function Oriented design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function. Thus, system is designed from a functional viewpoint.

OBJECT ORIENTED DESIGN

Definitions

i. Design entity. An element (Component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.
ii. Design View. A subset of design entity attribute information that is specifically suited to the needs of a software project activity.
iii. Entity attributes. A named property or characteristics of a design entity. It provides a statement of fact about the entity.
iv. Software design description (SDD). A representation of a software system created to facilitate analysis, planning, implementation and decision making

Software measurements: What & Why

"When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginnings of knowledge but you have scarcely in your thoughts advanced to the stage of Science."

measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined unambiguous rules

Types of Metric

direct measurement

eg number of lines of code

indirect/ derived measurement

defect density = no. of defects in a software

product / total size of product

prediction

eg predict effort required to develop software from

measure of the functionality Ð function point count

Token Count

The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program is defined as:

$$\eta = \eta_1 + \eta_2$$

where

η : vocabulary of a program

η_1 : number of unique operators

η_2 : number of unique operands

Design Metrics and Data Structure Metrics

Data Structure Metrics

Program	*Data Input*	*Internal Data*	*Data Output*
Payroll	Name/ Social Security No./ Pay Rate/ Number of hours worked	Withholding rates Overtime factors Insurance premium Rates	Gross pay withholding Net pay Pay ledgers
Spreadsheet	Item Names/ Item amounts/ Relationships among items	Cell computations Sub-totals	Spreadsheet of items and totals
Software Planner	Program size/ No. of software developers on team	Model parameters Constants Coefficients	Est. project effort Est. project duration

LAYERED ARRANGMENT IN MODULES:

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

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

A **bottom-up** approach (also known as inductive reasoning, and in many cases used as a synonym of *synthesis*) is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of information processing

based on incoming data from the environment to form a perception. Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output). In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

UNIT-IV

Software Testing & Software Maintenance:

Software Testing

Code Review

The importance of software testing to software quality cannot be over emphasized. Once source code has been generated, software must be tested to allow errors to be identified and removed before delivery to the customer. While it is not possible to remove every error in a large software package, the software engineer's goal is to remove as many as possible early in the software development cycle. It is important to remember that testing can only find errors, it cannot prove that a program is bug free. Two basic test techniques involve testing module input/output (black-box) and exercising internal logic of software components (white-box). Formal technical reviews by themselves cannot find allow software defects, test data must also be used. For large software projects, separate test teams may be used to develop and execute the set of test cases used in testing. Testing must be planned and designed. The SEPA web site contains the template for a generic test plan.

Software Testing Objectives

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

Software Testing Principles/Process:

- All tests should be traceable to customer requirements.
- Tests should be planned long before testing begins.
- The Pareto principle (80% of all errors will likely be found in 20% of the code) applies to software testing.
- Testing should begin in the small and progress to the large.
- Exhaustive testing is not possible.

- To be most effective, testing should be conducted by an independent third party.

Strategic Approach to Software Testing

- Testing begins at the component level and works outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- The developer of the software conducts testing and may be assisted by independent test groups for large projects.
- The role of the independent tester is to remove the conflict of interest inherent when the builder is testing his or her own product.
- Testing and debugging are different activities.
- Debugging must be accommodated in any testing strategy.
- Make a distinction between verification (are we building the product right?) and validation (are we building the right product?)

TESTING ACTIVITIES

Functionality testing

- It is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. Functionality testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.
- It is a type of black box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (Not like in white-box testing)
- It differs from system testing in that functional testing "verif[ies] a program by checking it against ... design document(s) or specification(s)", while system testing "validate[s] a program by checking it against the published user or system requirements"

Functional testing typically involves five steps:

1. The identification of functions that the software is expected to perform
2. The creation of input data based on the function's specifications
3. The determination of output based on the function's specifications
4. The execution of the test case
5. The comparison of actual and expected outputs

Structural testing

Structural testing (also known as clear box testing, glass box testing, transparent box testing, White-box testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to

exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT).

While white-box testing can be applied at the unit, integration and system levels of the software testing process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

White-box test design techniques include:

- Control flow testing
- Data flow testing
- Branch testing
- Path testing

Unit Testing, Integration Testing and System

Software design methodologists distinguish between several types of automated tests. First of all, unit tests test only a single "unit" of the code (say a module or a class), to see if it behaves as expected. They generally make sure that the behaviour of the module is sane and desirable, while not trying to see if it works as part of the larger scheme.

On the other hand, system tests test the entire system. For example, if we're writing code to generate a web-site, we could test that the various pages of the resultant site contain some of the qualities that we expect. System tests tests the system as a whole, to see if there's a bug somewhere.

Between unit tests and system tests there could be several intermediate layers of tests, normally called integration tests .

Debugging Activities

- Debugging (removal of a defect) occurs as a consequence of successful testing.
- Some people are better at debugging than others.
- Common approaches:

1. Brute force (memory dumps and run-time traces are examined for clues to error causes)
2. Backtracking (source code is examined by looking backwards from symptom to potential causes of errors)
3. Cause elimination (uses binary partitioning to reduce the number of locations potential where errors can exist)

Software Maintenance

Software maintenance is done to correct faults, improve performance, or adapt a software system to a new environment. Software must be maintained when:

1. The reality the software models changes,
2. New functionality is added,
3. It is easier to change software than hardware,
4. Software must be updated to run on improved hardware or with improved software.

They categorized maintenance activities into four classes:

Maintenance Type	Description
Corrective	Fixes a fault in the software without changing or adding to the software's functionality.
Adaptive	Modifies software to preserve functionality in a changed environment.
Perfective	Improves software performance, maintainability, etc., and can extend the functionality of the application.
Preventive	Changes are made to the system in order to prevent further faults and to improve the structure and maintainability of the system.

The survey showed that around 75% of the maintenance effort was on the first two types, and error correction consumed about 21%. Many subsequent studies suggest a similar magnitude of the problem. Studies show that contribution of end user is crucial during the new requirement data gathering and analysis. And this is the main cause of any problem during software evolution and maintenance. So software maintenance is important because it consumes a large part of the overall lifecycle costs and also the inability to change software quickly and reliably means that business opportunities are lost.

Software maintenance processes

This section describes the six software maintenance processes as:

1. The implementation process contains software preparation and transition activities, such as the conception and creation of the maintenance plan; the preparation for handling problems identified during development; and the follow-up on product configuration management.
2. The problem and modification analysis process, which is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyze each request, confirm it (by reproducing the situation) and check its validity, investigate it and propose a solution, document the request and the solution proposal, and finally, obtain all the required authorizations to apply the modifications.
3. The process considering the implementation of the modification itself.
4. The process acceptance of the modification, by confirming the modified work with the individual who submitted the request in order to make sure the modification provided a solution.
5. The migration process (platform migration, for example) is exceptional, and is not part of daily maintenance tasks. If the software must be ported to another platform without any change in

functionality, this process will be used and a maintenance project team is likely to be assigned to this task.

6. Finally, the last maintenance process, also an event which does not occur on a daily basis, is the retirement of a piece of software.

Management of maintenance:

1. The reality the software models changes,
2. New functionality is added,
3. It is easier to change software than hardware,

Software must be updated to run on improved hardware or with improved software

Maintenance process models

Traditional models fail to capture the evolutionary nature of software. Therefore different models are required that recognizes the requirement to build maintainability into the system.

The five models that are used most in the industry are:

- The quick fix model
- Boehm's model
- Osborne's model
- The iterative enhancement model
- The reuse oriented model.

The quick fix model is an ad-hoc approach. Its goal is to identify the problem and then fix it as quickly as possible. Due to time constraints, the model does not pay attention to the long-term affects of the fixes. The advantage of this model is that it gets work done quickly with lower cost.

For example, if a system is developed and maintained by only one person, then that person will know the system well enough to make changes in a short time without the need to manage detailed documentation. Reverse engineering

A second model is Boehm's model. The foundation of Boehm's model is based on economic models and principles. The use of economic models helps us to better understand the problem and improve productivity in maintenance. Osborne's model is concerned with the reality of the maintenance environment. In Osborne's point of view, technical problems that arise during maintenance are due to poor communication and control between management. Osborne recommends four strategies to address these issues.

1) Maintenance requirements need to be included in the change specification.

2) A quality assurance program is required to establish quality assurance requirements.

3) A metrics needs to be developed in order to verify that the maintenance goals have been met.

4) Managers need to be provided with feedback through performance reviews

The iterative enhancement model considers that changes made to the system during the software lifetime make up an iterative process. This model was adapted from development to maintenance.

The model has three stages.

1. First, the system has to be analyzed.
2. Next, proposed modifications are classified.
3. Finally the changes are implemented.

This model is not effective when the documentation of the system is not complete, as the model assumes that a full documentation of the system exists. The reuse oriented model assumes that existing program components could be reused The steps for the reuse model are identifying the parts of the old system which have the potential for reuse, fully understanding the system parts, modifying the old system parts according to the new requirements, and integrating the modified parts into the new system.

All of these models have their strengths and weaknesses. Therefore, usually more than one model is necessary for all maintenance activities. The best approach is to combine the models when require

Reverse engineering

Reverse engineering is the process of discovering the technological principles of a device, object, or system through analysis of its structure, function, and operation. It often involves taking something (e.g., a mechanical device, electronic component, software program, or biological, chemical, or organic matter) apart and analyzing its workings in detail to be used in maintenance, or to try to make a new device or program that does the same thing without using or simply duplicating (without understanding) the original.

Reverse engineering has its origins in the analysis of hardware for commercial or military advantage. The purpose is to deduce design decisions from end products with little or no additional knowledge about the procedures involved in the original production. The same techniques are subsequently being researched for application to legacy software systems, not for industrial or defence ends, but rather to replace incorrect, incomplete, or otherwise unavailable documentation.

Software reengineering

The reengineering of software was described by Chikofsky and Cross in their 1990 paper, as "The examination and alteration of a system to reconstitute it in a new form". Less formally, reengineering is

the modification of a software system that takes place after it has been reverse engineered, generally to add new functionality, or to correct errors.

This entire process is often erroneously referred to as reverse engineering; however, it is more accurate to say that reverse engineering is the initial examination of the system, and reengineering is the subsequent modification.

Re-engineering is mostly used in the context where a legacy system is involved. Software systems are evolving on high rate because there more research to make the better so therefore software system in most cases, legacy software needs to operate on a new computing platform. 'Re-engineering' is a set of activities that are carried out to re-structure a legacy system to a new system with better functionalities and conform to the hardware and software quality constraint.

Documentation

Documentation is a term used in several different ways. Generally, documentation (to document) refers to the process of providing evidence. Modules of Documentation are Helpful. The most common meanings are:

- the process of documenting knowledge (or rather knowledge claims) (as done in, for example, scientific journals).
- the writing of software documentation (or other kinds of product documentation)
- a synonym for the term document
- a synonym for the term bibliography
- a field of study and a profession founded by Paul Otlet (1868-1944) and Henri La Fontaine (1854-1943), which is also termed documentation science. Professionals educated in this field are termed document lists. This field changed its name to information science in 1968, but some uses of the term documentation still exists and there have been efforts to reintroduce the term documentation as a field of study.

Documentation may include

- written information for any read, projection or technical performing,
- Data media of any format and for any reproduction, other content.

Common types of documentation include user guides, white papers, on-line help, quick-reference guides. It is less common to see hard-copy (paper) documentation. Documentation is distributed via websites, software products, and other on-line applications.

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.

As a rule, system testing takes, as its input, all of the "integrated" software components that have passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are

integrated together (called *assemblages*) or between any of the *assemblages* and the hardware. System testing is a more limited type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.

System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS). System testing tests not only the design, but also the behaviour and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification(s).