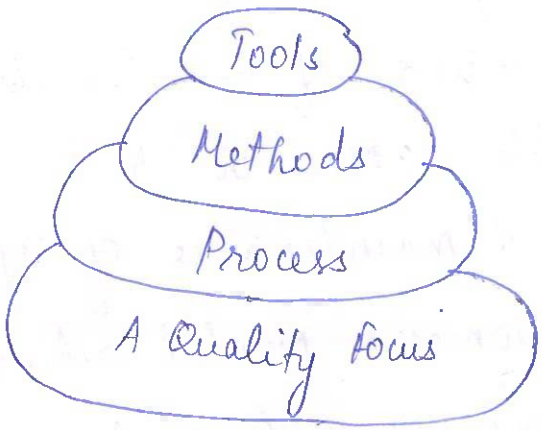
UNIT- I

* Introduction to Software Engineering
* Software Project Management - Life cycle Activities
* Traditional - Waterfall, V Model
* Prototype, Spiral, RAD
* Conventional - Agile
* XP, SCRUM
* Introduction to Requirement Engineering
* Requirements Elicitation
* Software Project Effort & Cost Estimation
* Cocomo 1 & 2
* Risk Management
* Configuration Management
* Project Planning - WBC, Planning
* Scope, Risk.

UNIT-I

INTRODUCTION TO SOFTWARE ENGINEERING

* Software are instructions that when executed provide desired features, function & performance.
* Data structures that enable the programs to adequately manipulate information & descriptive information in both hard copy & virtual forms that describes the operation & use of program



* Software Engineering methods provide the technical how-to's for building software. Methods encompass a broad array of task that include communication requirement analysis, design modeling, program construction, testing & support.
* Software Engineering tools provide automated or semiautomated support for the process & the methods
* When tools are integrated so that information created by one tool can be used by another

* A system for the support of software development called computer aided software engineering is established.

* Software Engineering is a layered technology. Any Engineering approach must rest on organizational commitment to quality.
  * Total Quality Management, Six Sigma foster a continuous process improvement culture
  * The application of a systematic, disciplined, quantifiable approach to the development, operation & maintenance of software.
  * Design becomes a pivotal activity & software should exhibit high quality.
  * The problem should be best understood before a software solution is developed.

SOFTWARE PROJECT MANAGEMENT - LIFE CYCLE ACTIVITIES

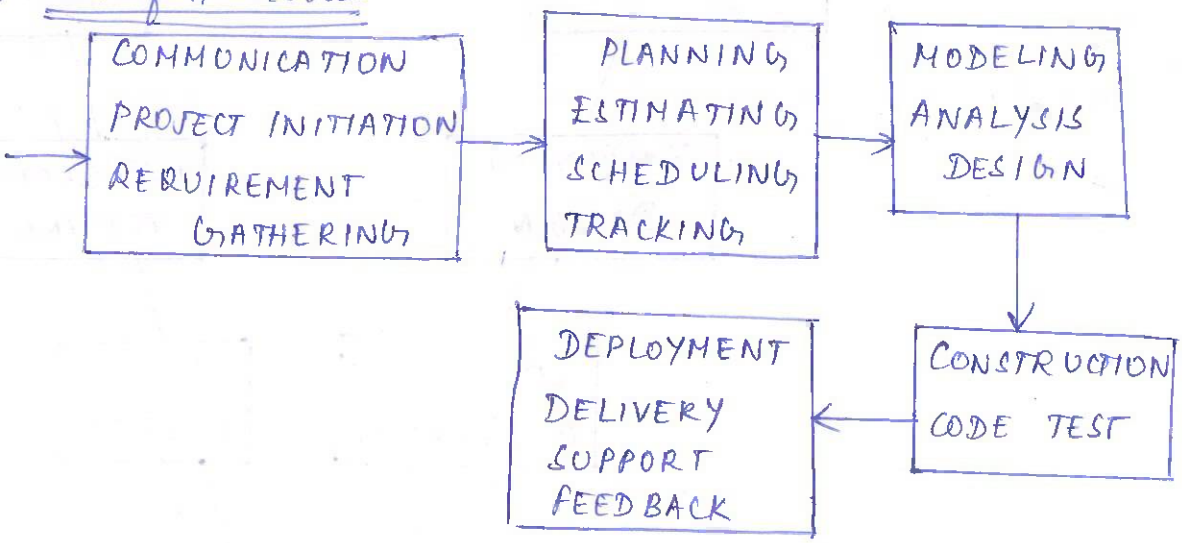
* SPM activities include
  (i) Requirement Analysis
  (ii) Product Design
  (iii) Programming
  (iv) Test Planning
  (v) Verification & Validation

(vi) Project Office Functions

(vii) Configuration Management & Quality Assurance.

TRADITIONAL – WATERFALL, VMODEL.

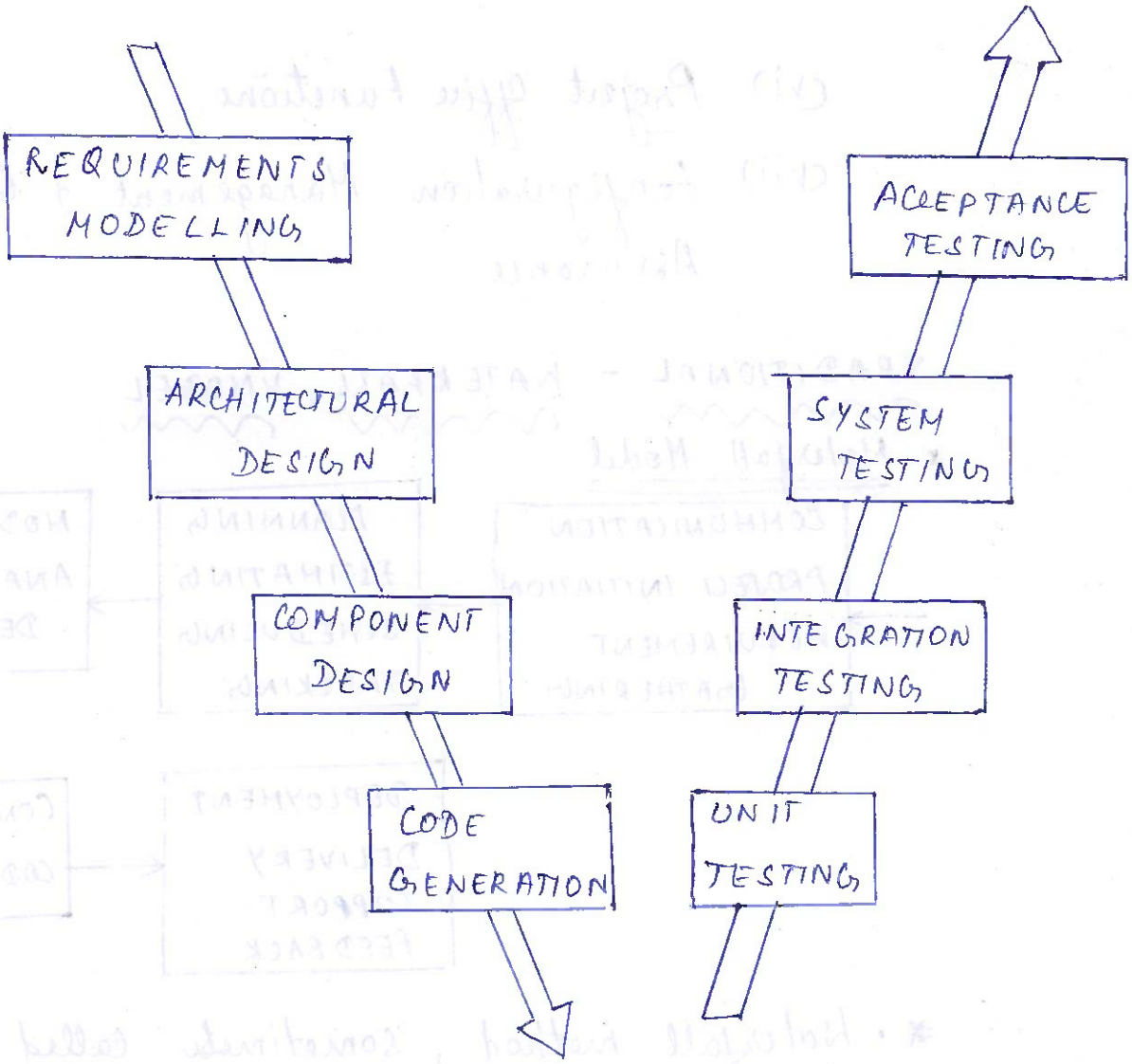
* Waterfall Model



* • Waterfall method, sometimes called the classic life cycle, suggests a systematic sequential approach to software development that begins with customer specification of requirements & progresses through Planning, Modelling, Construction + Deployment, culminating ongoing support to the completed software
  • A variation in the representation of Waterfall Model is called V-Model

* V-MODEL :

  • Real project rarely follows the sequential flow that the waterfall model proposes. Although the linear model can accomodate iteration, it does so indirectly.

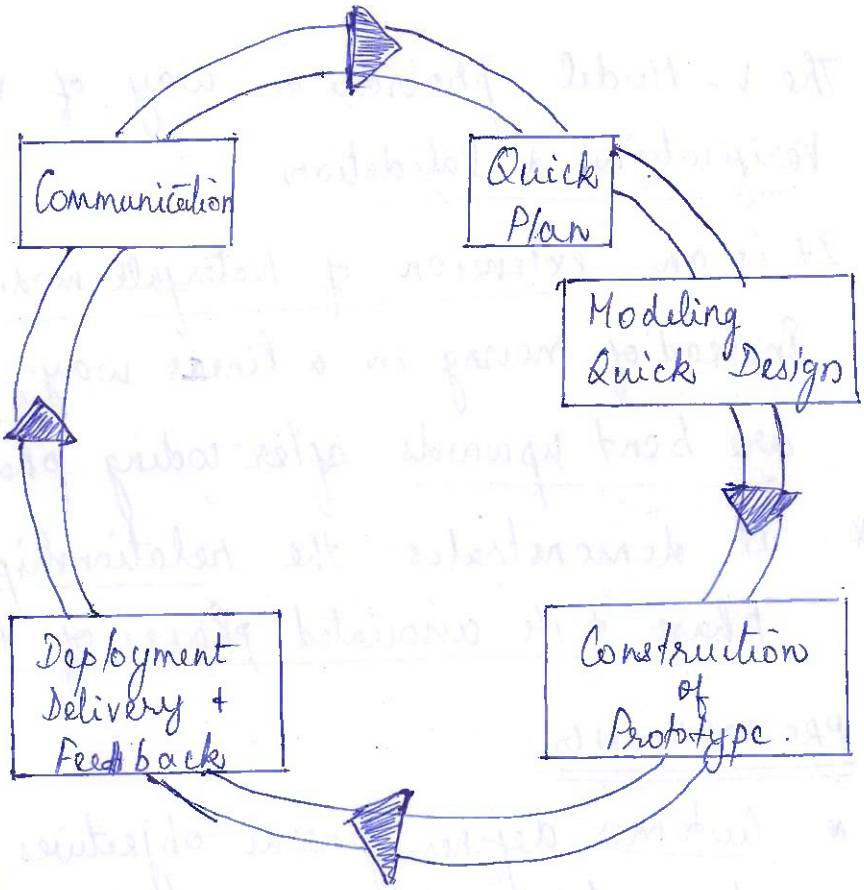


* It is often difficult for the customer to state all requirements explicitly. The waterfall model requires these requirements & forces difficulty in accomodating the natural uncertainity.
* The customer must have patience. A working version of the programs will not be available until late in the project life span. I N-f Model.
* As the software team moves down the left side of the "V" basic problem requirements are refined into progressively more detailed and technical representation of the problem and its solution. V-Model.

* The V-Model provides a way of visualizing, Verification & Validation
* It is an extension of Waterfall model
* Instead of moving in a linear way, process steps are bent upwards after coding phase.
* It demonstrates the relationship between each phase & its associated phases of testing.

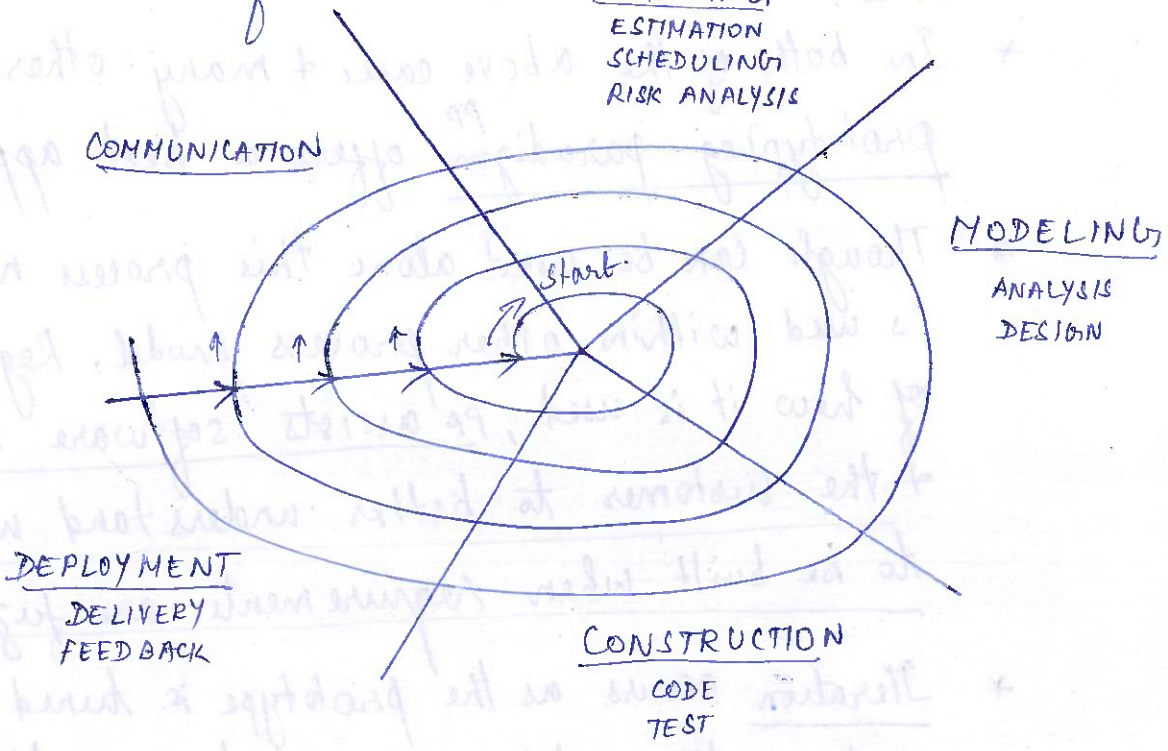
PROTOTYPING

* Customer defines general objectives for software, but does not identify detailed input, processing or output requirements.
* Developer may be unsure of the efficiency of an algorithm, adaptability of an operating system or the form human-machine interaction should take.
* In both of the above cases & many other cases, prototyping paradigm (PP) offers a best approach
* Though can be used alone this process model is used within other process model. Regardless of how it is used, PP assists software engineer & the customer to better understand what is to be built when requirements are fuzzy.
* Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while developer better understands what needs to be done.



SPIRAL MODEL

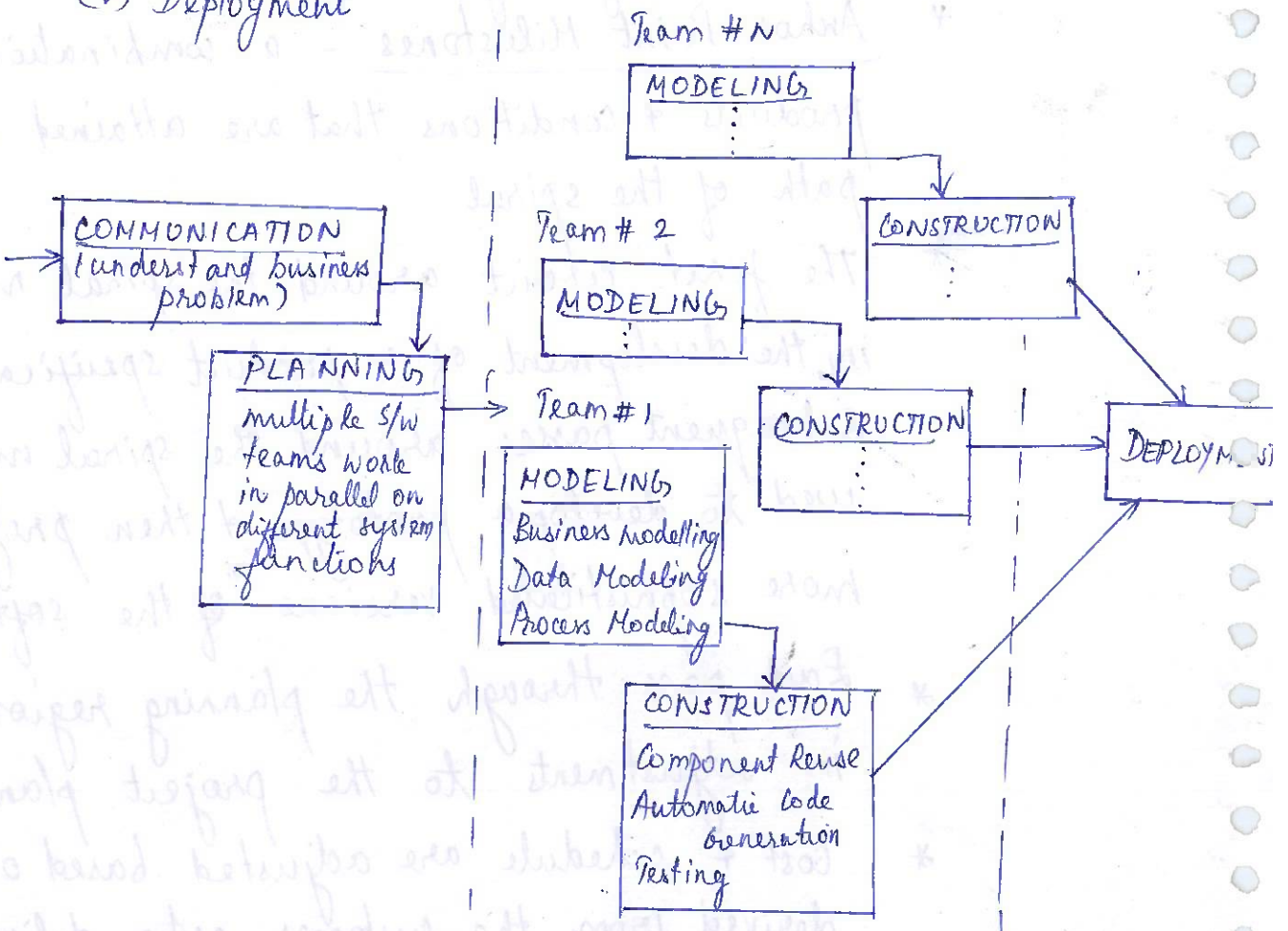
* It is an evolutionary software process model that couples the iterative nature of prototyping with the controlled & systematic aspects of the waterfall model



* Using spiral model, software is developed in a series of evolutionary releases.
* During early iterations, the release might be a paper model or prototype.
* During later iterations, the release might be more complete versions of the engineered system.
* Spiral model is divided into a set of framework activities defined by the software engineering team
* As the evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction beginning at the centre
* Anchor Point Milestones - a combination of work products & conditions that are attained along the path of the spiral.
* The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype & then progressively more sophisticated versions of the software.
* Each pass through the planning region results in adjustments to the project plan.
* Cost & schedule are adjusted based on feedback derived from the customer after delivery.
* This can be applied throughout the life of the computer software.

RAD MODEL

* Rapid Application Development is an incremental software process model that emphasizes on short development cycle.
* High speed adaptation of waterfall model by using a component-based construction approach.
* If requirements are well understood & project scope is constrained RAD enables a fully functional system within a very short time period.
* It also uses generic framework activities.
  (i) Communication (ii) Planning (iii) Modeling (iv) Construction
  (v) Deployment



60-90 Days.

CONVENTIONAL MODELS:

* Conventional - conforming or adhering to accepted standards (widely accepted)

* AGILE MODELS:

  - Agility - is the ability to respond to changes.
  - Change is what software development is very much about - changes in s/w being built
    changes in team members
    changes in technology
    changes in project that creates product.
  - Agile Alliance defines 12 principles for those who want to achieve agility:

    (i) Satisfy the customer through early & continuous delivery of valuable s/w.

    (ii) Welcome changing requirements even late in development

    (iii) Deliver working software frequently, from a couple of weeks to a couple of months, with preference to the shorter timescale.

    (iv) Business people & developers must work together daily throughout the project.

    (v) Build projects around motivated individuals. Give them the environment & support they need, & trust them to get the job done.

    (vi) The most efficient & effective method of conveying information to & within a development team is face-to-face conversation

(vii) Working software is the primary measure of progress

(viii) Promote sustainable development

(ix) Agile processes require continuous attention to technical excellence & good design enhances agility

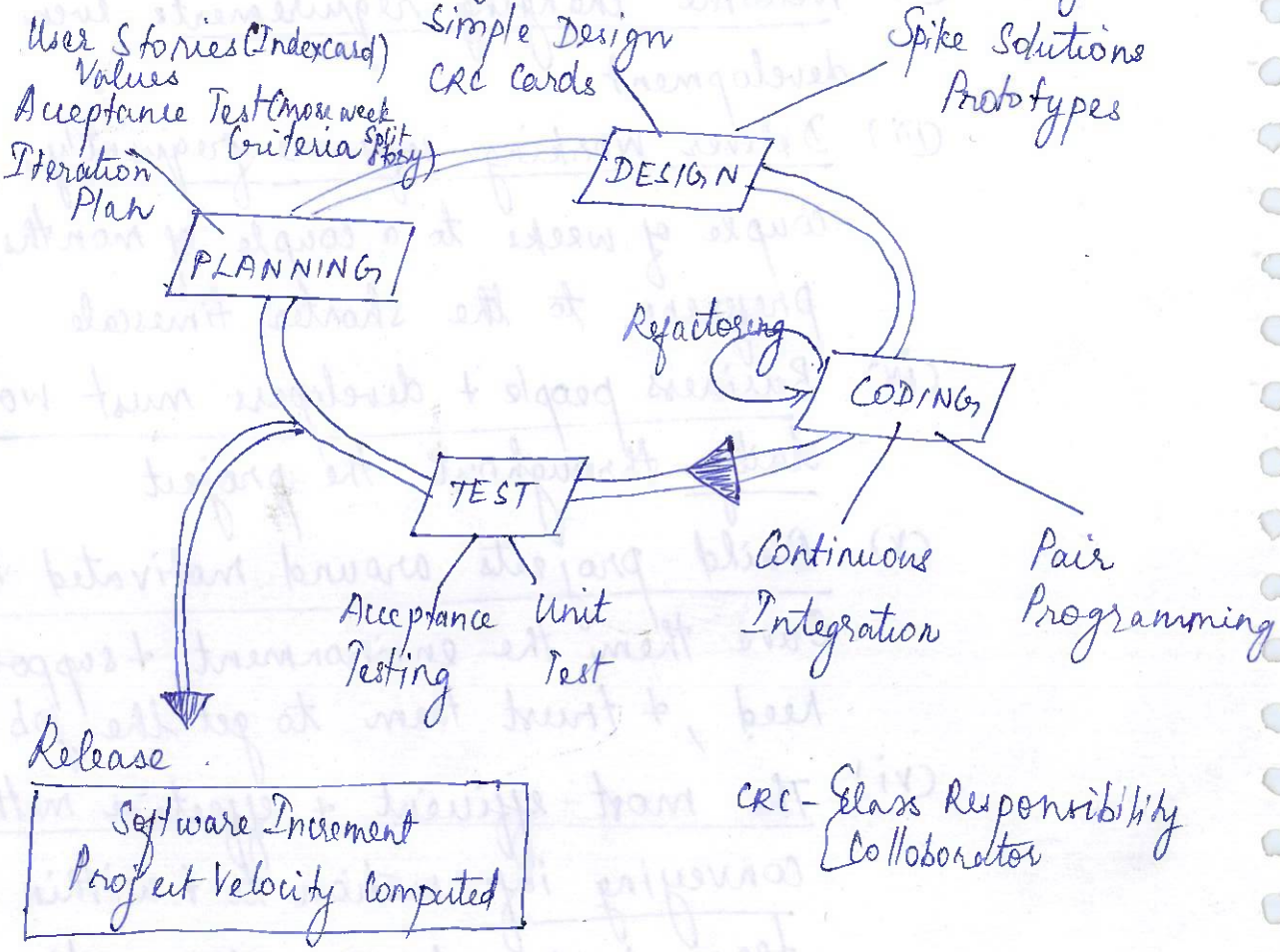
(x) Simplicity

(xi) Self organizing teams

(xii) At regular interval, the team reflects on how to become more effective, then tunes & adjusts its behavior accordingly.

* EXTREME PROGRAMMING (XP)

- XP uses an object-oriented approach
- It encompasses a set of rules & practices that occur within the context of 4 framework activities (i) Planning (ii) Design (iii) Coding & (iv) Testing



* SCRUM

- The name is derived from an activity that occurs during a rugby match)
- Scrum principles are consistent with agile principles
  (i) Small working teams are organized to maximize communication, minimize overhead & maximize sharing of informal knowledge
  (ii) Process must be adaptable to both technical & business changes
  (iii) Process yields frequent software increments that can be inspected, adjusted tested, documented and built on.
  (iv) Development work & the people who perform it are partitioned into clean, low coupling partitions are packets.
  (v) Constant testing + documentation is performed as the product is built.
  (vi) Scrum process provides the ability to declare a product done whenever required
- These principles are used to guide development activities within a process using framework activities: (i) Requirement Analysis (ii) Design (iii) Evolution & (iv) Delivery
- With each framework activity work tasks occur within a process pattern called Sprint
- The work conducted within a sprint (no. of sprints) will vary with product complexity & size) is

INTRODUCTION TO REQUIREMENTS ENGINEERING (RE)

* Understanding the requirements of a problem
* RE provides the appropriate mechanism for understanding what the customer wants (1), analyzing need (2), assessing feasibility (3), negotiating a reasonable solution (4), specifying the solution (5), validating the specification (6) and managing the requirements (7) as they are transformed into an operational system.
* The RE process is accomplished through the execution of 7 distinct functions
  (i) Inception (ii) Elicitation (iii) Elaboration
  (iv) Negotiation (v) Specification (vi) Validation
  (vii) Management

*ELICITATION

- Ask the customer, the users & others what the objectives for the system / product are, what is to be accomplished, how the system / product fits into the needs of the business & finally how the system / product is to be used on a day-to-day basis
- Problems Encounters by Elicitation
  (i) Problems of Scope : Boundary of the system is ill-defined or customers / users specify unnecessary technical detail leading to confusion rather than

(ii) Problems of understanding : The customers/users are not completely sure of what is needed, have poor understanding of the capabilities & limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information essential, specify requirements that conflict needs of customers/users, specify requirements that are ambiguous/untestable.

(iii) Problems of volatility : Requirements change over time.

- Inception uses Q+A session (context free questions) but this might not be overwhelmingly successful for a detailed Elicitation Requirements.
- Collaborative Requirements Gathering (CRG) : To address the problem a team oriented approach is preferred. The stakeholders & developers work together to identify the problem, propose elements of solution, negotiate approaches & specify preliminary set of solution requirements.
- ▲ Basic Guidelines for CRG
  - (i) Meetings conducted & attended by both S/W Engineers + Customers
  - (ii) Rules for preparation + participation are established
  - (iii) An agenda is suggested to cover all important points + also free flow of ideas.

(iv) A facilitator (customer/developer/outsider) controls the meeting

(v) A definition mechanism (chat/worksheet/virtual forum) is used

(vi) Goal is to identify problem, specify elements of s/n, negotiate approaches, specify preliminary set of s/n requirements.

- Quality Function Deployment (QFD) : This technique translates the needs of the customer into technical requirements for software. It aims at maximizing customer satisfaction from the software engineering process. It identifies 3 types of requirements

(i) Normal Requirements : Customer is satisfied with this as these requirements reflect objectives & goals stated for a product/system during meetings with the customer Eg: graphical displays.

(ii) Expected Requirements : These are implicit to the product/system & are so fundamental that the customer does not explicitly state them. Absence might lead to dissatisfaction. Eg: human/machine interaction.

(iii) Exciting Requirements : These go beyond customer's expectations & prove to be very satisfying when present. Eg: No. of page layout.

- User Scenarios : Developer & users should create a set of scenarios to understand how the functions & features will be used by different classes of end-users (Use cases) — How system will be used

- Elicitation Work Products (WP) : WP produced as a consequence of RE include
  (i) Statement of need & feasibility
  (ii) Bounded statement of scope for the system/product
  (iii) List of customers, users & other stake holders who participated in requirements elicitation
  (iv) A description of the systems technical environment
  (v) A list of requirements & the domain constraints that apply.
  (vi) A set of usecase scenarios that provide insight into the use of the system / product under different operating conditions
  (vii) Prototypes developed to better define requirements

  All these WP's are reviewed by all people who participated in RE.

SOFTWARE PROJECT ESTIMATION (EFFORT & COST)

* It is the most expensive element of virtually all computer - based systems
* A large cost estimation error can make the difference between profit & loss.
* Cost overrun can be disastrous for the developer
* To achieve reliable cost & effort estimates, a no. of options arise:
  (i) Delay estimation until late in the project
  (ii) Base estimates on similar projects that have already been completed.

(iii) Relatively simple decomposition techniques to generate project cost & effort estimates

(iv) Use one/more empirical models for software cost & effort estimation.

* option(i) not practical & must be provided "up front".
* option(i) can work reasonably well, if current project is quite similar to pass efforts & other project influences. But unfortunately, past experience are not always a good indicator of future results.

* DECOMPOSITION TECHNIQUES

(*) Software Sizing: The accuracy of a software project estimate depends on no. of things:

a) Degree to which planner has properly estimated the size of the product to be built
b) Ability to translate the size estimate into human effort, calendar time & dollars.
c) The degree to which project plan reflects the abilities of the software team
d) stability of product requirements & the environment that support the SE effort

- In project planning, size refers to the quantifiable outcome of the software project.
- If a direct approach is taken, size can be measured in Lines of code (LOC)
- If an indirect approach is taken, size is represented as function points (FP)

(*) Problem-Based Estimation

- LOC & FP data are used in 2 ways during project estimation
  (i) As an estimation variable to size each element of the software
  (ii) As baseline metrics collected from past projects & use in conjunction with estimation variables to develop cost & effort projections.
- Project planner begins with a bounded statement of software scope & from this statement attempts to decompose software into problem functions that can be estimated individually.
- LOC / FP is then estimated for each function.
- Planner on the other hand, may choose another component for sizing such as classes / objects, changes or business processes affected.
- Baseline productivity metrics (LOC/pm or FP/pm^5) are then applied to the appropriate estimation variable & cost / effort for the function is derived.
- Function estimates are combined to produce an overall estimate for the entire project.
- When LOC is used as the estimation variable, decomposition is absolutely essential & is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed

- When FP estimates are used, decomposition works differently. Rather than focusing on function, each of the 5 information domain characteristics as well as 14 complexity adjustment values are estimated. The resultant estimates can then be used to derive a FP value that can be tied to past data & used to generate an estimate.
- Regardless of the estimation variable used, the project planner begins by estimating a range of values for each function / information domain value. Using historical data the planner estimates an optimistic size value for each function / count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified.
- A 3-point / expected-value can be computed.
- The Expected-value for the estimation variable (size), S, can be computed as a weighted average of the optimistic (S_{opt}), most likely (S_m) & pessimistic (S_{pess}) estimates.

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

- Eg: LOC based

Optimistic – 4600 LOC
most likely – 6900 LOC
pessimistic – 8600 LOC
Expected value = 6800 LOC

Estimated LOC
33200

Historical Data : 620 LOC/pm ; Labor rate : $8000/pm
Cost/LOC : $13 ($\frac{8000}{620}$) Total Estimated Cost = $431,000 (33200 × 13)

- Eg : FP-Based

Info. Domain Value	Opt	Likely	Pess.	Est Count	Weight	FP Count
No. of external I/p's	20	24	30	24	4	97
No. of external O/p's	12	15	22	16	5	78
No. of external inquiries	16	22	28	22	5	88
No. of internal logical files	4	4	5	4	10	42
No. of external interface files	2	2	3	2	7	15
						320

Historical Data : 6.5 FP/pm ; Labour Rate : $8000/m

Cost / FP : $1230. [8000 / 6.5] Total Estimated Cost = $461,000 [375 × 1230 = 461 250]

Total Estimated Effort = 58 person/m [461000 / 8000]

Estimated no. of FP = FP estimated = count-total × (0.65 + 0.01 × ΣFi)]

ΣFi :- {total of value adjustment factors (Pg. 646) = 52} = 320 (0.65 + 0.01 × 52)

count-total :- Σ FP count = 320 = 375

EMPIRICAL ESTIMATION MODELS

* This uses empirically derived formulas to predict effort as a function of LOC/FP.

* STRUCTURE OF ESTIMATION MODELS

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

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

A, B & C — empirically derived constants
E — effort in person-months
e_v — estimation variable (either LOC/FP)

COCOMO MODEL

* It stands for Constructive Cost Model (COCOMO)
* Most widely used software cost estimation models in the industry.
* It has evolved into a more comprehensive estimation model called COCOMO II
* Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the foll. areas:
  - Application composition model : Used during early stages of software engineering, when prototyping of user interfaces, consideration of software & system interaction, assessment of performance, & evaluation of technology maturity are paramount.
  - Early Design Stage Model : Used once requirements have been stabilized and basic software has been established.
  - Post-architecture Stage Model : Used during the construction of the software.
* COCOMO II models require sizing information. 3 different sizing options are (i) Object Points (ii) FP (iii) LOC
* Application Composition Model uses OPs — an indirect software measure that is computed using counts of the no. of (i) screens (ii) reports (iii) components required to build the application.
* Each object instance is classified into one of 3 complexity levels (i) Simple (ii) Medium (iii) Difficult

* Complexity is a function of the no. of source of the client & server data tables that are required to generate the screen / report & the no of views / sections presented as part of the screen / report.

Object Type	Complexity Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Component			10

* The OP count is then determined by multiplying the original no. of object instances by the weighting factor & summing to obtain a total OP count.

* New Object Points (NOP)

$$NOP = (\text{object points}) \times [(100 - \%\ \text{reuse})/100]$$

* To derive an estimate of effort based on the computed NOP value, a productivity rate, PROD must be derived

$$PROD = NOP / \text{person-month}$$

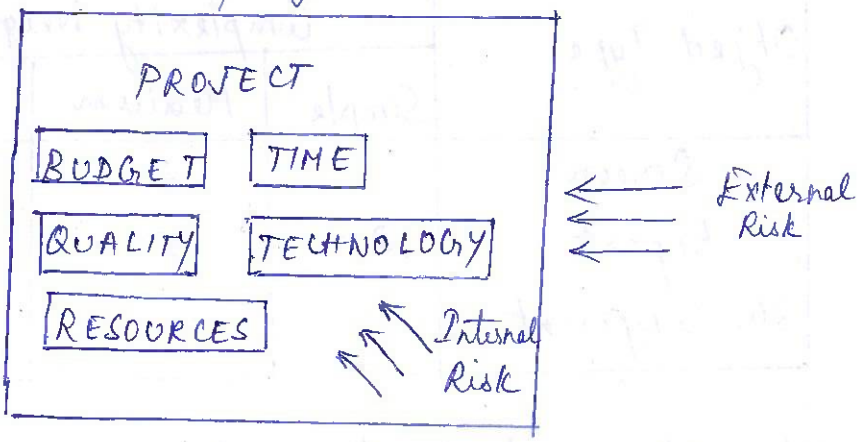
* for different levels of developer experience & development environment maturity

* Once PROD has been determined, an estimate of project effort can be derived as

$$\text{Estimated Effort} = NOP / PROD$$

RISK MANAGEMENT

* Risk can be categorized as internal & external
* Internal Risk - Risk arises due to an aspect being dealt with project team



* External Risk - All other risks are categorized into it.

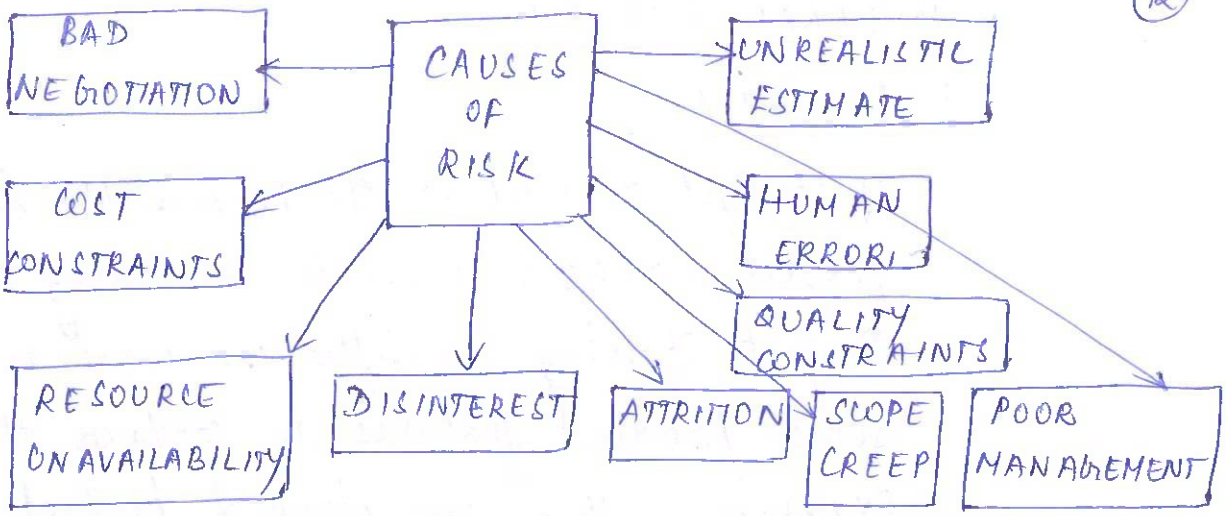
CAUSES OF RISK

(i) QUALITY CONSTRAINTS :
A seperate software process group is formed that overseas the quality of projects.

(ii) RESOURCE UNAVAILABILITY :
Finding & Procuring a good software professional is a major task. Retaining him within the organization is yet another challenge

(iii) DISINTEREST :
Software professional might become disinterested and opt to move out of the project.

(iv) ATTRITION :
Due to high demand for software professionals most have job offers in hand at any time. They quit an organization if the find any good offer.



(v) SCOPE CREEP:

Requirements keep changing & new requirements keep piling up even after the project has completed the testing phase

(vi) COST CONSTRAINTS:

The project cannot proceed as sources of funds have dried up & project expenses cannot be met.

(vii) BAD NEGOTIATION:

Lack of foresight on the part of the customer due to bad negotiation happens & adequate support is also not provided.

(viii) UNREALISTIC ESTIMATE:

It is always better to keep a buffer when an estimate is made, to take care of uncertainities.

(ix) HUMAN ERROR:

It is caused due to distractions of brain & might end up in requirements / design / construction may get injected with defect. Proper Review should be done.

(x) POOR MANAGEMENT:

Lack of experience in managing a project may bring liabilities

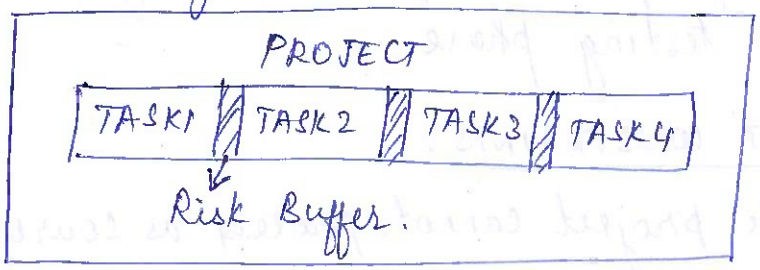
* INTERNAL RISK TYPES

(i) **BUDGET RISK :**

If budget goes above the permissible limit, then project manager must do something to control it. Project Steering Committee can work to cut short some product features to contain the project expenses within the planned budget.

(ii) **TIME RISK :**

The project should never be allowed to cross the targeted release dates

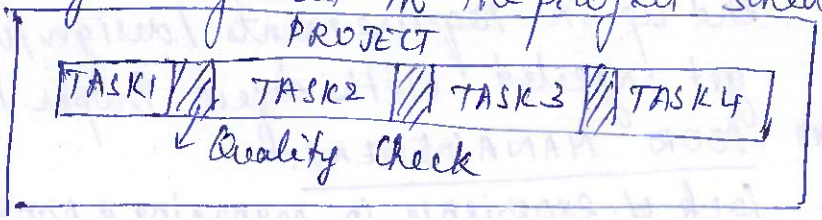


(iii) **RESOURCE RISK :**

Project team members leaving in the middle of a project is a big loss, so Knowledge Management System implementation is needed.

(iv) **QUALITY RISK :**

To deal with this, the best policy is to check for quality integrated in the project schedule itself.



(v) TECHNOLOGY RISK :

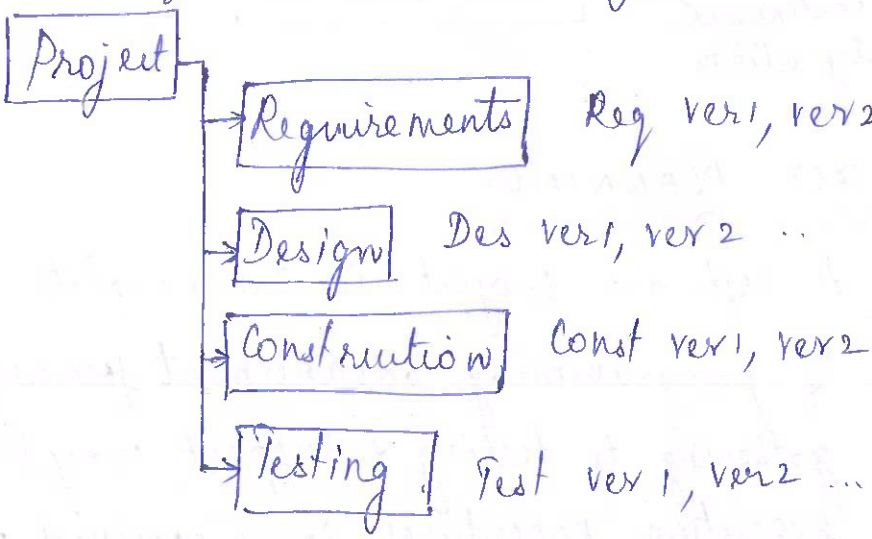
An appropriate selection of hardware, programming language, the access method is very ideal.

CONFIGURATION MANAGEMENT (CM)

* A CM system is used by the entire project team including own team as well as contractors + service providers.

* It includes following fields.

(i) Project Name (ii) Year + Time Stamp (iii) Document Name (iv) Document No. (v) Author (vi) Activity Identifier (vii) Document Type (viii) Version Numbers



* CONFIGURATION MANAGEMENT TECHNIQUES

(i) Centralized CM system

(ii) Secured access mechanism with role based access control

(iii) Continuous integration of software build with Smoke test facility.

(IV) Easy branching mechanisms to branch out on entity software version

(V) Audit facility

* It would be difficult to control versions of documents & would make it harder for teams to manage their work. CMS provides a solution for this problem
* However synchronizing different versions of documents over different CM system is a tedious & error prone task

PROJECT PLANNING

* A software project is the complete methodology of programming advancement from requirement gathering to testing & support, completed by the execution procedures, in a specified period to achieve a software product

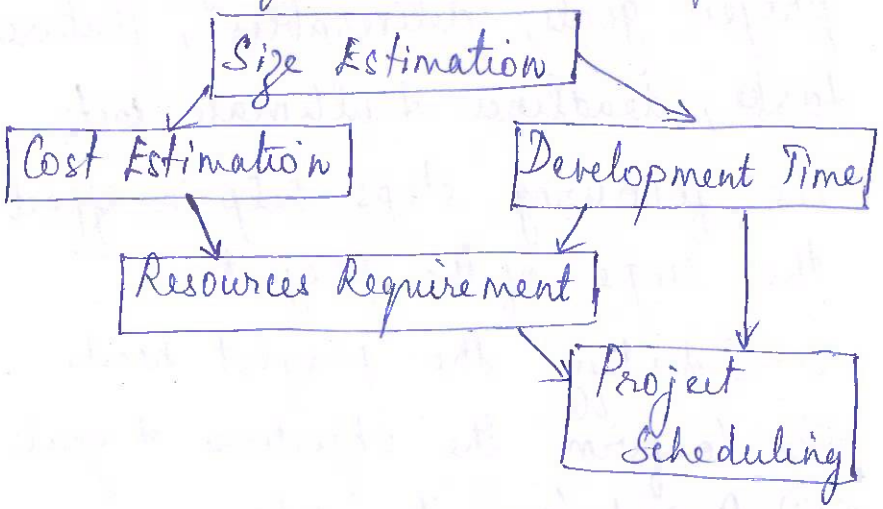
PLANNING

* Objective of S/W project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost and schedule

* Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of planning tasks.
* Therefore, the plan must be adapted & updated as the project proceeds.

* Project Planning Activities

(i) Establish project scope
(ii) Determine feasibility
(iii) Analyze Risks
(iv) Defines Required Resources
  (a) Human Resources
  (b) Reusable software Resources
  (c) Environmental Resources
(v) Estimate Cost + Effort
  (a) Decompose problem
  (b) Develop 2/more estimates
  (c) Reconcile estimates
(vi) Develop a Project Schedule
  (a) Establish a meaningful task set
  (b) Define a task network
  (c) Use scheduling tools to develop a timeline chart
  (d) Define schedule tracking mechanisms.



WORK BREAKDOWN STRUCTURE (WBS)

* WBS divides complex projects to simpler & manageable tasks.
* Project managers use this method for simplifying the project execution
* In WBS, much larger tasks are broken down to manageble chunks of work.
* Following are a few reasons for creating a WBS in a project
  (i) Accurate & readable project organization
  (ii) Accurate assignment of responsibilities to the project team
  (iii) Indicates the project milestones & control points
  (iv) Helps to estimate the cost, time & Risk.
  (v) Illustrate the project scope, so that the stake holders can have a better understanding of the same

PROJECT SCOPE

* It is part of Project Planning that involves determining & documenting a list of specific project goals, deliverables, features, functions, tasks, deadlines & ultimate costs.
* The following steps help in effectively defining the scope of the project:
  (i) Identify the project needs
  (ii) Confirm the objectives & goals of the project
  (iii) Project Scope Description

(iv) Expectation & Acceptance

(v) Identify Constraints

(vi) Identify Necessary changes.

PROJECT RISKS

* Possibility of suffering from loss in software development process is called software risk.
* Loss can be anything from increase in production cost, development of poor quality software to not being able to complete the project on time.
* Various categories of risks associated with Software Project Management are enumerated below
  (i) Scheduled / Time Related / Delivery Related Planning Risks
  (ii) Budget / Financial Risk
  (iii) Operational / Procedural Risk
  (iv) Technical / Functional / Performance Risk
  (v) Other Unavoidable Risks.

UNIT - II

* Software Design - Software Design Fundamentals
* Design Standards - Design Type
* Design Model - Architectural Design, Software Architecture
* Software Design Methods
* Top Down, Bottom Up.
* Module Division (Refactoring)
* Module Coupling
* Component Level Design
* User Interface Design
* Pattern Oriented Design
* Web Application Design
* Design Reuse
* Concurrent Engineering in Software Design
* Design Life-Cycle Management

SOFTWARE DESIGN — UNIT - II

* Software design sits at the kernel of software engineering & is applied regardless of the software process model that is used
* Once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity & hence sets the stage for construction (code generation & testing)
* During design we make decisions that will ultimately affect the success of software construction & the ease with which software can be maintained.
* Importance of Design: Quality. Design is the place where quality is fostered in software engineering.
* Design is the only way that can accurately translate a customer's requirement into a finished software product or system.
* Without design, we risk building an unstable system - one that will fail when small changes are made; one that may be difficult to test, one whose quality cannot be accessed until late in the software process, when time is short and budget have already been spent.

SOFTWARE DESIGN FUNDAMENTALS

Fundamental software design concepts & getting things right include.

(i) **Abstraction** : A modular solution to any problem, leads to many levels of abstraction. Highest Level Abstraction - solution is stated in broad terms using the language of the problem environment. Lowest Level Abstraction - more detailed description of solution is provided. Procedural / Data Abstraction can be employed.

(ii) **Architecture** : It is the structure / organization of program components (modules), the manner in which these components interact & the structure of data that are used by the components.

(iii) **Patterns** : It describes a design structure that solves a particular design problem within a specific context & amid forces that may have an impact on the manner in which the pattern is applied & used. It intends to provide a description that enables a designer to determine a) whether the pattern is applicable to the current work b) whether the pattern can be reused & c) whether the pattern can serve as a design guide for developing a similar but functionally or structurally different pattern.

(iv) **Modularity** : It is a single attribute of software that allows a program to be intellectually manageable

(v) INFORMATION HIDING: Modules should be specified (5) + designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

(vi) FUNCTIONAL INDEPENDENCE: It is achieved by developing modules with "single-minded" function + an "aversion" to excessive interaction with other modules addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure.

(vii) REFINEMENT: Stepwise Refinement is a top-down design strategy. Refinement causes the designer to elaborate on the original statement, providing more + more detail as each successive refinement (elaboration) occurs.

(viii) REFACTORING: It is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.

(ix) DESIGN CLASSES: As the design model evolves, the software team must define a set of design classes that (i) refine analysis classes by providing design detail that will enable classes to be implemented + (ii) create a new set of design classes that implement a software infrastructure to support the business solution

5 different classes suggested are (i) User Interface classes (ii) Business Domain classes (iii) Process classes (iv) Persistent classes (v) System classes.

DESIGN STANDARDS

* Software design is an iterative process through which requirements are translated into a blueprint for constructing the software
* 3 characteristics that serve as a guide for the evaluation of a good design:
  (i) Design must implement all of the explicit requirements contained in the analysis model & it must accomodate all of the implicit requirements desired by the customer
  (ii) Design must be a readable, understandable guide for those who generate code & for those who test & subsequently support the software.
  (iii) Design should provide a complete picture of the software, addressing the data, functional & behavioral domains from an implementation perspective.
* Quality Guidelines:
  (i) A design should exhibit an architecture that (a) has been created using recognizable architectural styles/patterns (b) is composed of components that exhibit good design characteristics & (c) can be implemented in an evolutionary fashion thereby facilitating implementing & testing.

(ii) A design should be modular; that is, the software should be logically partitioned into elements/subsystem

(iii) It should contain distinct representations of data, architecture, interfaces & components

(iv) It should lead to data structures that are appropriat for the classes to be implemented & are drawn from recognizable data patterns

(v) It should lead to components that exhibit independent functional characteristics.

(vi) A design should lead to interfaces that reduces the complexity of connections between components & with the external environment

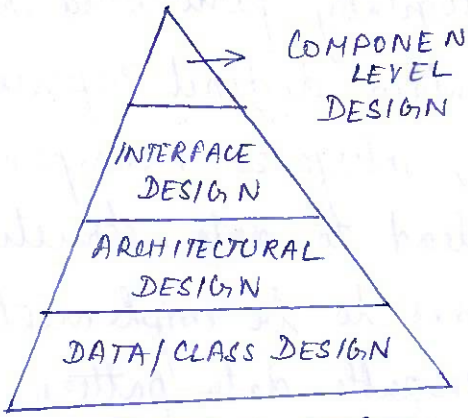
(vii) A design should be derived using a repeatable met that is driven by information obtained during software requirement analysis.

(viii) A design should be represented using a notation that effectively communicates its meaning.

* Quality Attributes:

- HP developed a set of software quality attributes given the acronyms FURPS
  - F - Functionality - Accessesed by evaluating features & capabili of program
  - U - Usability - accessed by human factors
  - R - Reliability - Evaluated by measuring MTTF, accuracy, recove from failure
  - P - Performance - Speed, response time, resource consumption throughput, efficiency
  - S - Supportability - ability to extend the program.

DESIGN MODEL



DESIGN MODEL

DATA DESIGN ELEMENTS

* Data design creates a model of data / information that is represented at a high level of abstraction (customer/ users view of data)
* The structure of data has always been an important part of software design
* At component level, design of data structures & the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
* At application level, the translation of data model into a database is pivotal to achieving the business objectives of a system.
* At business level, the collection of information stored in separate databases are reorganized into a datawarehouse that enables data mining or knowledge discovery that can have an impact on the success of the business itself.

ARCHITECTURAL DESIGN ELEMENTS

* It is equivalent to the floor plan of a house (overall layout of rooms, their size, shape & relationship to one another, etc).
* Floor plan gives us an overall view of the house
* Architectural design elements gives us an overall view of the software.
* Architectural model is derived from 3 sources (i) information about application domain for the software to be built (ii) specific analysis model elements such as data flow diagrams or analysis classes, their relationships & collaborations for the problem at hand (iii) availability of architectural patterns & styles.

INTERFACE DESIGN ELEMENTS :

* It is equivalent to a set of detailed drawings for doors, windows & external utilities of a house. (size & shape of doors & windows; the way connections like electrical, water, telephone come into house & distributed across rooms, location of doorbell, etc.)
* The interface design elements for software tell how information flows into & out of the system and how it is communicated among the components defined as part of the architecture
* 3 elements of interface design : (i) <u>user interface</u> (UI) (ii) <u>external interfaces</u> to other systems, devices, N/W or consumer of info. (iii) <u>internal interfaces</u> between various design components.

COMPONENT-LEVEL DESIGN ELEMENTS

* It is equivalent to a set of detailed drawings for each room in a house (wiring + plumbing within each room, location of switches, sinks, showers, drains, closets, flooring to be used, ...)
* The component level design (CLD) for software fully describes the internal detail of each software component.
* To accomplish this, CLD defines data structures for all local data objects + algorithmic detail for all processing that occurs within a component + an interface that allows access to all component operations.

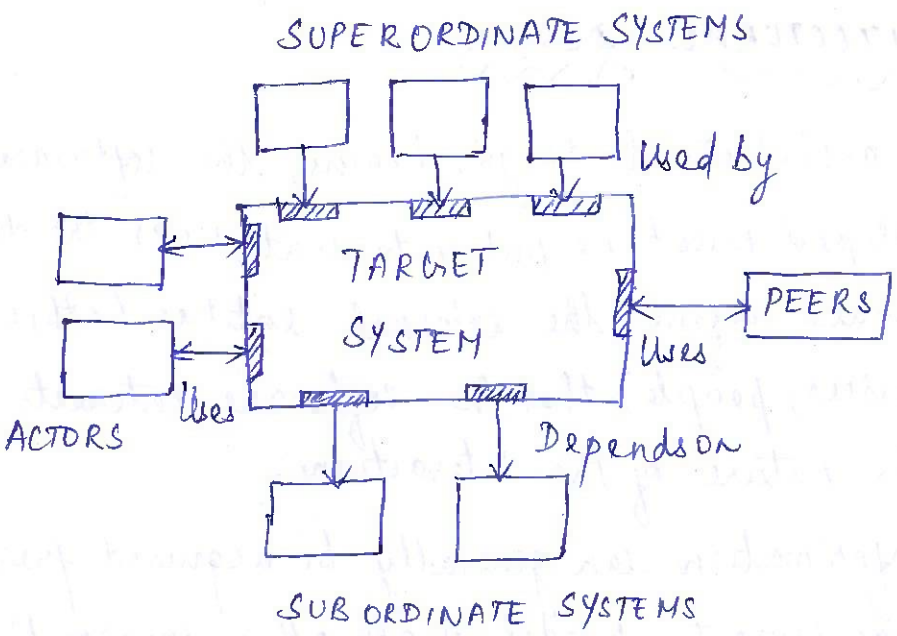
DEPLOYMENT LEVEL DESIGN ELEMENTS

* Deployment level design elements indicate how software functionality + subsystems will be allocated within the physical computing environment that will support software.
* Eg: SafeHome product involves 3 computing environments (i) home-based PC ii) Safe Home Control Panel (iii) a server

ARCHITECTURAL DESIGN

* As architectural design begins, the software to be developed must be put into context (ie) the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.
* Information can generally be acquired from the requirements model & all other information gathered during requirements engineering.
* An archetype is an abstraction that represents one element of system behavior.
* The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed.
* REPRESENTING THE SYSTEM IN CONTEXT

  - A software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries.
  - Systems that interoperate with the target system (TS) are represented as.
    (i) Superordinate Systems - use TS as part of higher level processing scheme
    (ii) Subordinate Systems - used by TS to provide data / processing to complete a functionality



(iii) Peer-level Systems - interact on a peer-to-peer basis (information is either produced / consumed by the peers & the TS)

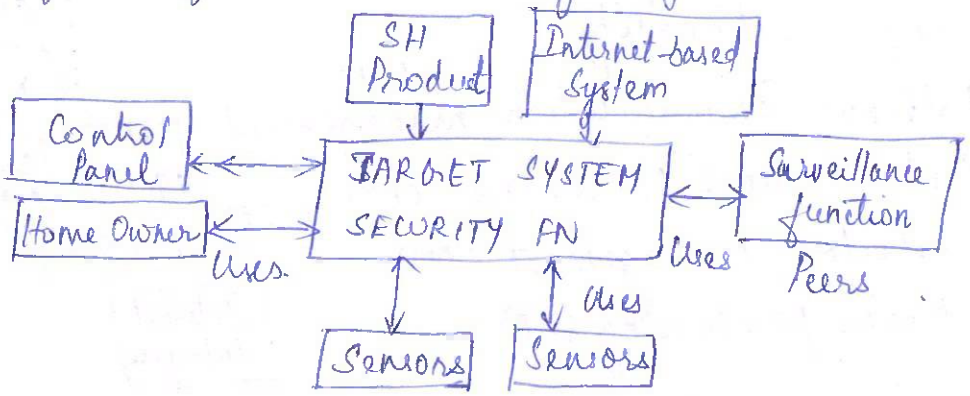
(iv) Actors - People / Devices that interact with TS by producing / consuming information that is necessary for requisite processing.

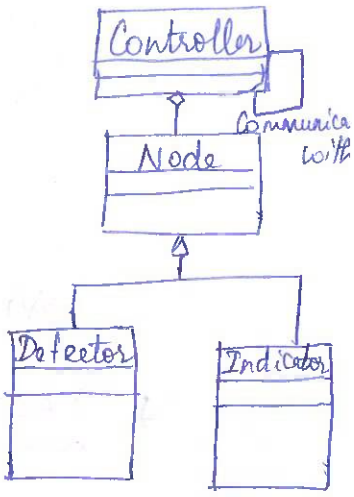
- Each of these external entities communicates with the target system through an interface (small shaded rectangles).

* DEFINING ARCHETYPES

- An archetype is a class that represents a core abstraction that is critical to the design of an architecture for the target system.
- There is relatively small set of archetypes required to design even relatively complex systems.
- The TS architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated in many different ways based on the behavior of the system.

• Eg: 'SafeHome' (SH) Security System.





• Archetypes:

(i) Node - collection of I/p & O/p elements of the home security function. Node comprised of a) various Sensors b) variety of alarm(O/p) indicate

(ii) Detector - encompasses all sensing equipment that feeds information into TS. (I/p)

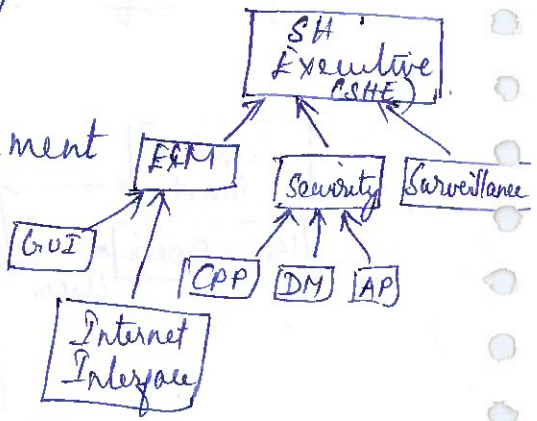
(iii) Indicator - represents all mechanism for indicating that an alarm condition is occurring (eg: alarm, bell, flash lights)

(iv) Controller - depicts the mechanism that allows the arming / disarming of a node. If controller reside on a network, they have the ability to communicate with one another.

* REFINING THE ARCHITECTURE INTO COMPONENTS

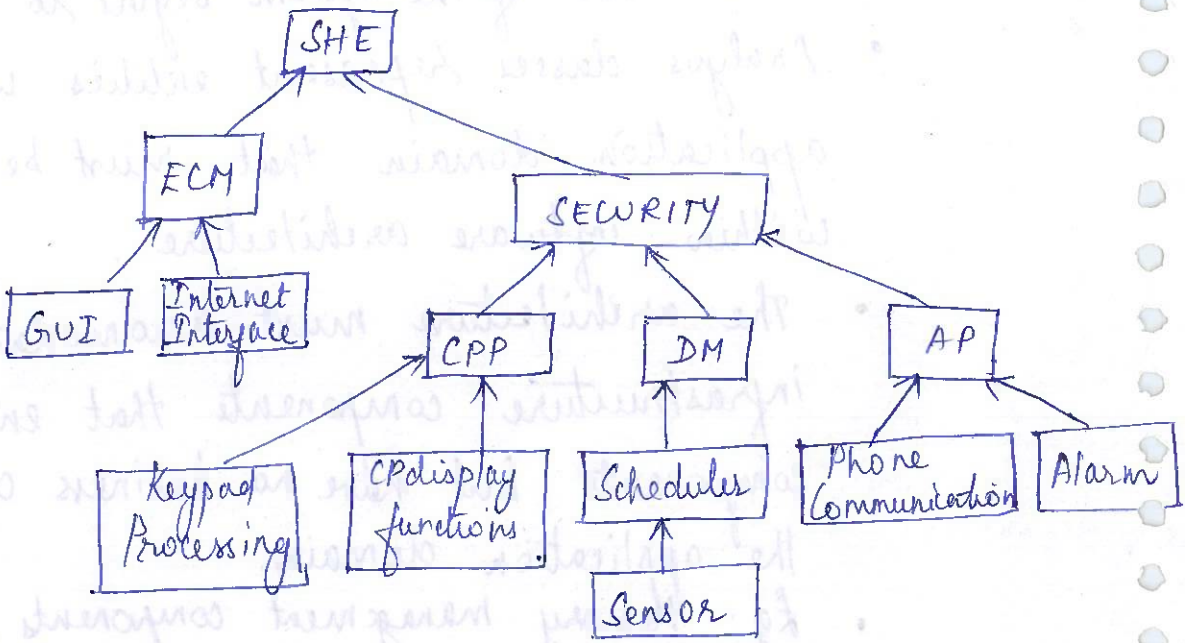
- As the software architecture is refined into components, the structure of the system begins to emerge
- Analysis classes represent entities within the application domain that must be addressed within software architecture.
- The architecture must accomodate many infrastructure components that enable application components but have no business connection to the application domain.
- Eg: Memory management components, communication components, database components & task management components are often integrated into the architecture.

- Eg: Safe Home might define components that address following functionality.
  (i) External Communication management (ECM)
  (ii) Control Panel processing (CPP)
  (iii) Detector management (DM)
  (iv) Alarm processing (AP)



* DESCRIBING INSTANTIATIONS OF THE SYSTEM

- The architectural design has been modeled to this point is at relatively high level.
- The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, ~~(Abstractions with problem)~~ overall structure of the system is defined & major software components have been identified
- However further refinement is still necessary & to accomplish this an actual instantiation of the architecture is developed.



SOFTWARE ARCHITECTURE

* The architecture of a system is a comprehensive framework that describes its form and structure its component
* Software architecture of a program or computing system is the structure or structures of the system which comprise software components, the externally visible properties of those components + the relationships between them.
* It is a representation that enables a software developer to
  (i) analyze the effectiveness of the design in meeting its stated requirements
  (ii) consider architectural alternatives at a stage when making design changes is still relatively easy
  (iii) reduce the risks associated with the construction of the software.

SOFTWARE DESIGN METHODS

* There are 2 methods for designing software products or components
  (i) Top - Down approach
  (ii) Bottom - Up approach

TOP - DOWN APPROACH

* In this approach, the top structure of the product is conceived and designed first

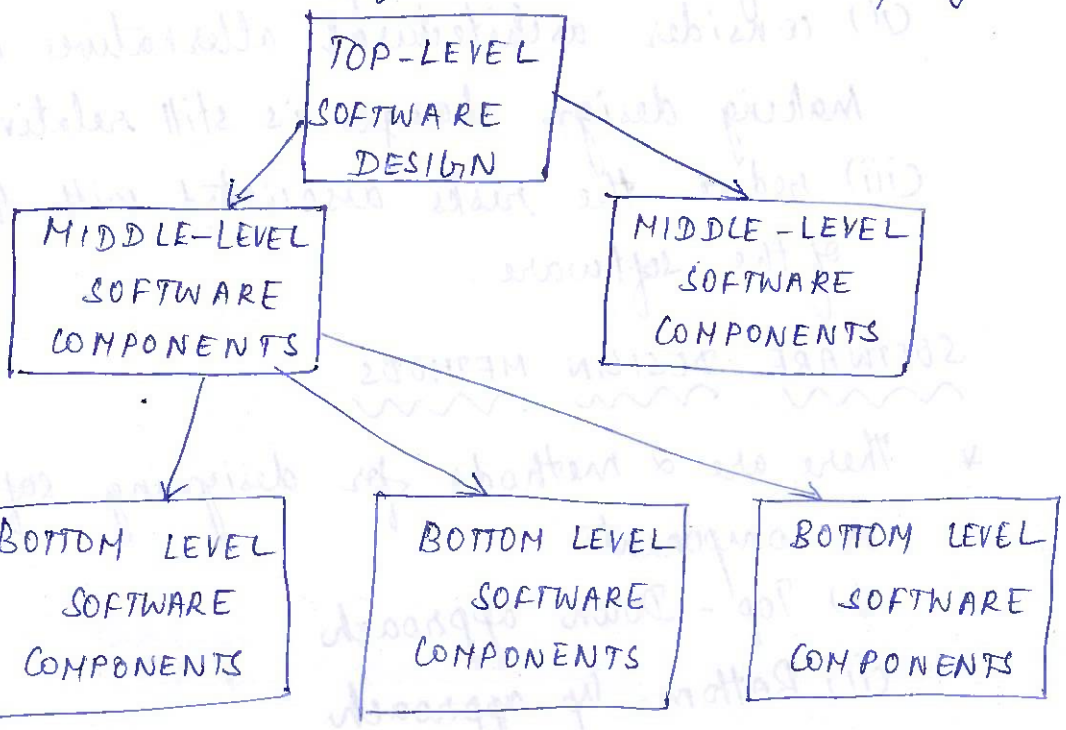
* Once the structure is obtained, components that will make the product are designed

* Benefits of the Top-Down Approach
  (i) Nonfunctional aspects are taken care of at the beginning of the design
  (ii) It results in a secure, robust & usable product

* Since it helps in creating reusable components it increases productivity as well as maintainability.

* Drawback of the Top-Down Approach
  (i) It is a risky model – the whole design is totally framed instead of adopting to incremental approach of building the design.

* It uses water-fall model to build projects.



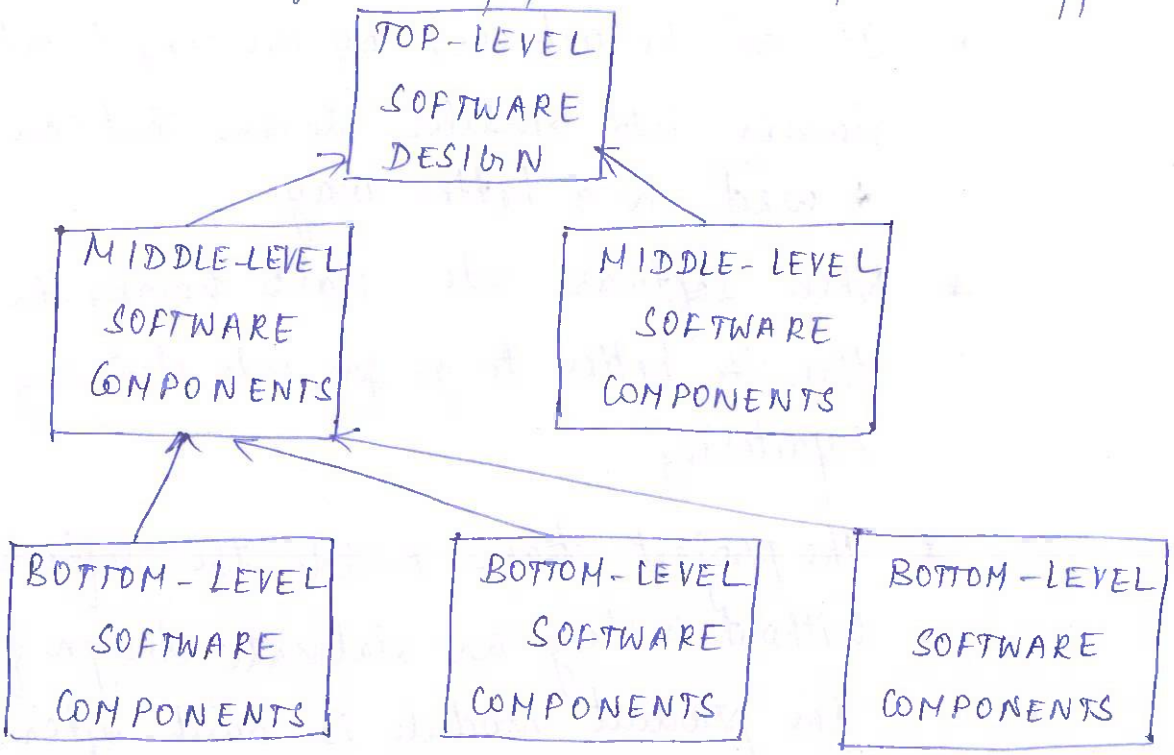
BOTTOM-UP APPROACH

* In this approach, the minute functions of the software product are structured and designed at first.

* Then the middle-level components are designed and finally the top-level components are structured and designed.

* Benefits of Bottom-Up approach
  (i) Leads to incremental building of design that ensures that any missing information can be accomodated in the design.

* With increasing use of incremental & iterative development methodologies, the bottom-up approach is becoming more popular than top-down approach.



MODULE DIVISION (REFACTORING).

* In case customers wants to increase the funtionality of the product over time, the product can be extended to take care of those increased needs.
* It becomes necessary to change the internal structure of the software code without changing external behavior of the software product.
* Refactoring is similar to the concept of normalization in relational databases.
* It can be achieved by dividing cumbersome classes into smaller classes that can be managed & used in a better way.
* When software code starts having the characteristics then its better to go for code cleaning or refactoring
* The project team builds the software product without making an elaborate design from start. One product module is built after another in subsequent iterations
* Characteristics of a software product that requires refactoring
  (i) Duplicate code
  (ii) Long functions
  (iii) Large number of call parameters
  (iv) Message Chaining

(v) Classes with many concepts
(vi) Large class size.

MODULE COUPLING

* One area similar to refactoring is coupling between modules.
* Changes in code result is more than normal occurance of defects as dependencies between modules keeps increasing with increase in size of the product.
* In order to reduce the chances of product defects, it reduces the number of calls (dependencies) among different modules & classes.
* Frequent refactoring can help in reducing module coupling among classes.

COMPONENT-LEVEL DESIGN

* It occurs after 1st iteration of architectural design.
* At this stage, the overall data & program structure of the software has been established.
* Goal: to translate design model into operational software
* Component is a modular building blocks for computer software
* Because components reside within the software architecture, they must communicate & collaborate with other components & entities existing outside boundaries

* In the context of OO s/w Engineering, a component contains a set of collaborating classes with each class within a component including all attributes & operations that are relevant to its implementation. Interfaces/Messages enable classes to communicate & collaborate. The designer elaborates a) analysis classes - components related to problem domain & b) infrastructure classes - components providing support services to problem domain.

* In context of conventional s/w Engineering, a component is a functional element of a program that incorporates processing logic, internal data structures that are required to implement processing logic & an interface that enables the component to be invoked & data to be passed to it. These components are called modules & serve one of 3 roles a) control component - coordinates the invocation of all other problem domain components b) problem domain component - implements a complete/partial function that is required c) infrastructure component - is responsible for functions that support the processing required in problem domain.

* OO & conventional views assume component is being designed from scratch. A Process related view emphasized the need to build systems that make use of existing software components (listed in a catalog). As software architecture is developed, components or design patterns are chosen from catalog & used to populate the architecture.

USER INTERFACE DESIGN

* Doors, Windows & Utility connections for computer make up the interface design of a system.
* Interface design focuses on 3 areas of concern
  (i) design of interfaces between software components
  (ii) design of interfaces between software & other 'non human producers & consumers of information
  (iii) design of interface between a human & the computer.

PATTERN ORIENTED DESIGN

* Each pattern describes a problem that occurs over & over again in our environment & that describes the core of the solutions to that problem in such a way that you can use the solutions a million times without ever doing it the same way twice.
* A design pattern can be categorized as a 3 part rule which expresses a relation between a certain context, a problem & a solution.

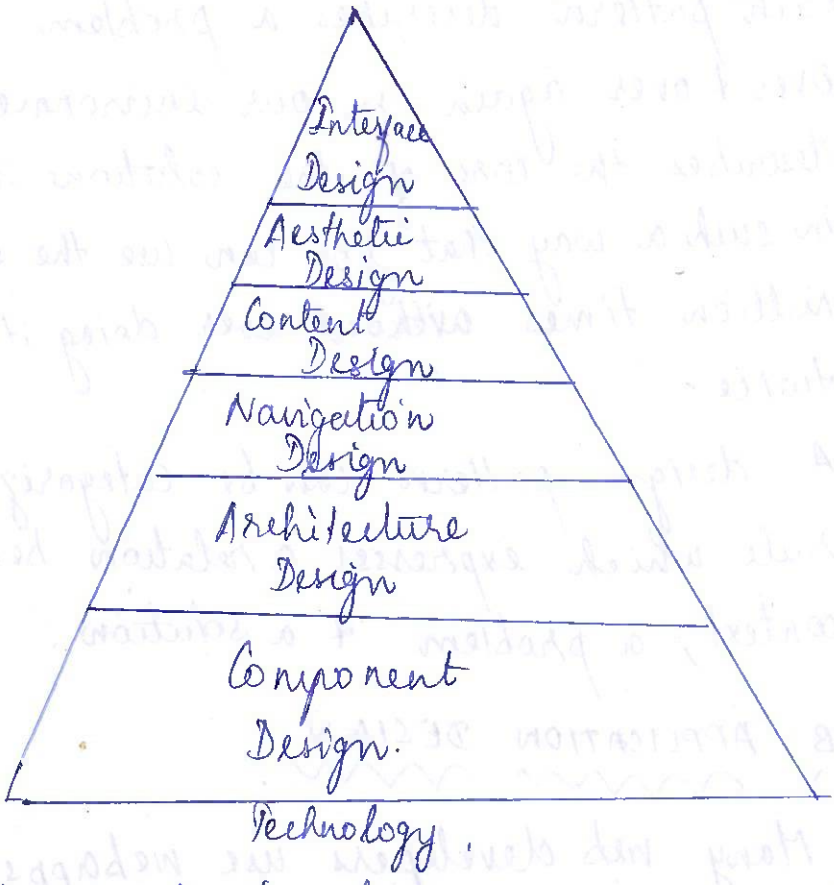
WEB APPLICATION DESIGN

* Many web developers use webapps for limited design
* WebApp Quality is perceived by what technical characteristics of quality it attributes to Web Engineer to correct, adapt, enhance & support the application over long term.

* It should exhibit the following characteristics
(i) Usability (ii) Functionality (iii) Reliability
(iv) Efficiency & (v) Maintainability.

* In addition following attributes are also extended into it
(i) Security (ii) Availability (iii) Scalability &
(iv) Time-to-market.

* Design goals include.
(i) Simplicity (ii) Consistency (iii) Robustness (iv) Navigability
(v) Visual Apeal & (vi) Compatibility.



* This depicts the design pyramid for web Engineering

DESIGN REUSE

* The design can be broken into many design parts representing each module of the product
* Each of the modules contain a lot of design information that can be represented as design component.
* The pieces of information in many information components can be reused.
* Method of reuse is known as Internal Design Reuse.
* A more design reuse mechanism is becoming available after the advent of the open source paradigms & SOA.
* The design reuse is in fact copying of existing design exactly and modifying it to suit to the current requirement.
* The owner of that application / component publishes full details to integrate another application with that of the owners application / component
* The full interface details are provided by the owner.

CONCURRENT ENGINEERING IN SOFTWARE DESIGN

* Concurrent Engineering deals with taking advance information from an earlier stage to a later stage in the evolution of a project.
* Project activities are planned ahead in time, most often there are dependencies between a previous task and the next one.
* The development team cannot start their job until they have a valid software design.
* Development language will be used & how the application can be partitioned for development work can be decided at the design stage itself.
* Maintenance & support functions will be done for the application. These can be determined at the design stage itself.

DESIGN LIFE CYCLE MANAGEMENT

* Software requirements go through design process steps to become a full-fledged software design.

* At the high level, system analysis is performed.
* System Analysis includes a study of requirements and finding feasibility of converting them into software design.
* Once the feasibility study is done, then the actual software design is made.
* The software design is in the form of activity diagrams, use cases, prototypes, etc.
* Once the design process is complete, these design documents are verified and validated through design reviews.
* Then the design is reviewed + approved, then the design phase is complete.

UNIT-III

* Software Construction,
* Coding Standards
* Coding Framework
* Reviews - Desk checks (Peer Reviews)
* Walkthroughs, Code Reviews, Inspections
* Coding Methods
* Structured Programming
* Object Oriented Programming
* Automatic Code Generation
* Software Code Reuse
* Pair Programming
* Test-Driven Development
* Configuration Management
* Software Construction Artifacts

TB: 2
CH: 12
Pg. No: 175-185

UNIT-III

SOFTWARE CONSTRUCTION

* Process of software construction consists of many tasks which includes not only software coding but also unit testing, integration testing, reviews & analysis.
* It is one of the most labor intensive phases in software development life cycle (SDLC) [Requirements Management, Software Design, Software Testing and Software deployment]
* It comprises 30% or more of the total effort in SDLC.
* The product the user sees at the end of SDLC is the result of software code written during software construction.
* Due to its intensive nature, SC phase is divided not only among developers but also teams are formed to work on parts of the software build to shrink construction time.
* Adv.: Software coding & other construction work is done in parallel known as Concurrent Engineering

* The construction process requires stringent coding standards to be followed so that the produced code is maintainable, testable & reliable.

* The process should be efficient so that resource utilization can be optimized & cost is minimized.

CODING STANDARDS.

* Developers are given software design specifications in the form of use cases, flow diagrams, UI mock ups, etc.

* Developers are supposed to write a code that matches these specifications & this specification to code conversion is totally dependent on construction team.

* They need some experience, skill & depends on the process they use to accomplish their job.

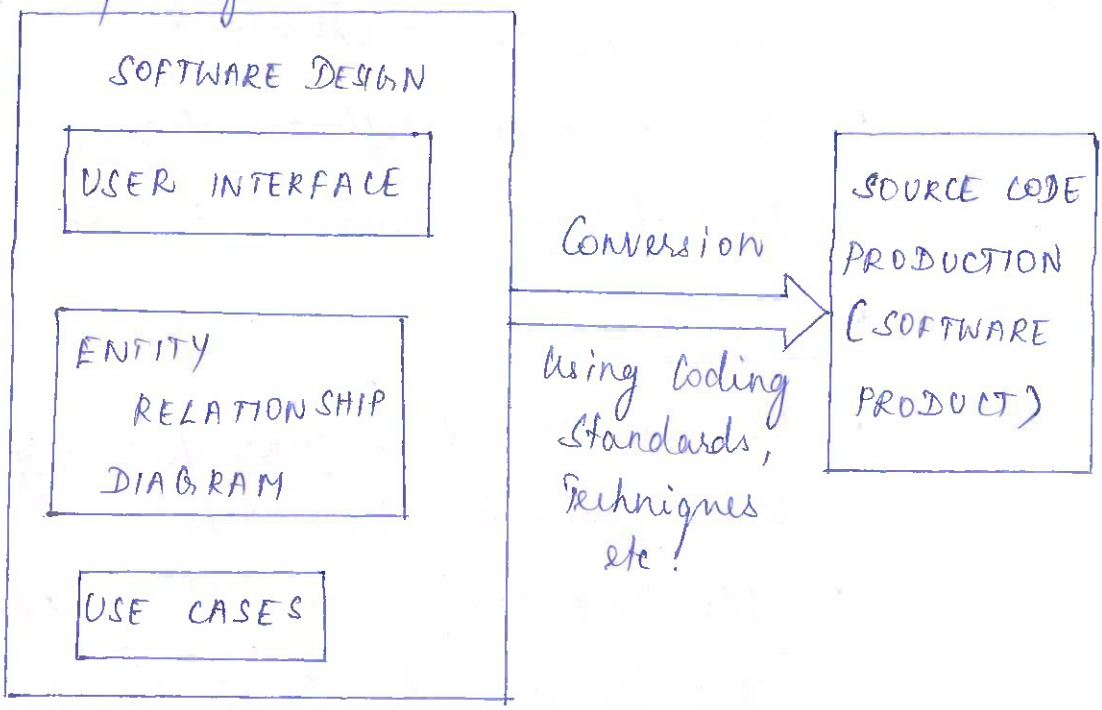
* They also need some standards in their coding so that the work is fast as well as has other benefits like maintainability, readability & reusability.

* Code written by one developer will be different from that written by any other developers.

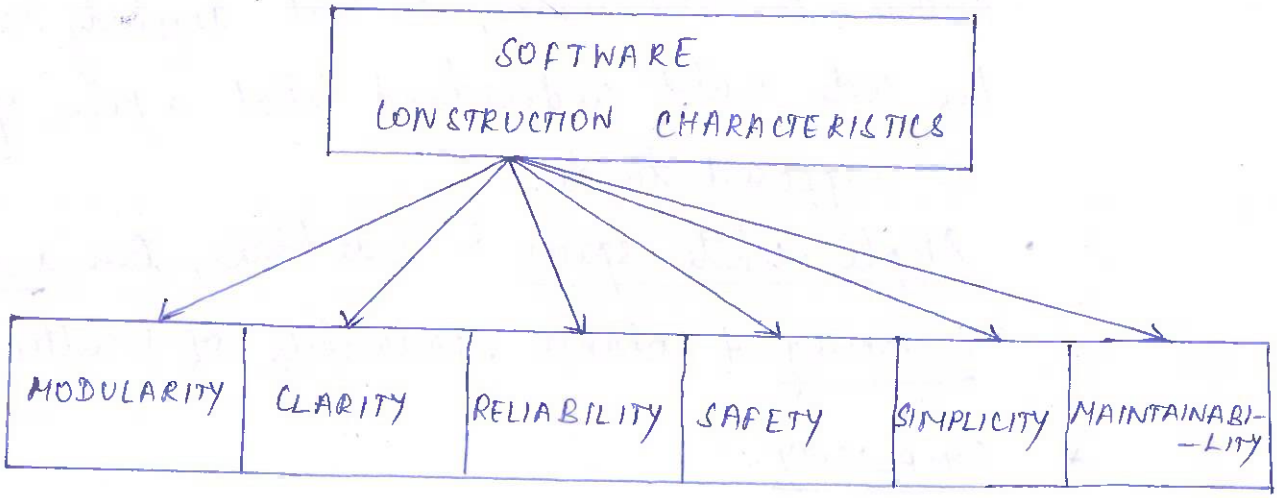
* This becomes a challenge in terms of comprehendi

the code while reusing the code, maintaining it, or simply reviewing it.

* A uniform coding standard across all construction teams working on the same project will make sure that this issue is minimized though cannot be completely eliminated.



* Software Construction Characteristics



* Coding Standards include standards for code modularity, clarity, simplicity, safety, reliability & maintainability

* MODULARITY : Produced software code should be modular in nature with each software code module holding each major function but should also process data.

- Each time a functionality is needed in software construction, that particular module of software code can be implemented.
- This increases software code reuse & thus enhances productivity of developers & code readability.

* CLARITY :

- The code should be clear for any person who would read the source code.
  - Standard naming conventions should be used so that the code has ample clarity.
  - Sufficient documentation should be embedded within the code block, so that anybody reading the code could understand what a piece of code is supposed to do.
  - Ample white spaces in code blocks, can avoid cramming & enhance readability of written code

* SIMPLICITY :

- Simplest piece of code can substitute unnecessary complex logic.
  - Simplicity makes code readable & helps in removing

any defects found in source code.

- It can be enhanced by adopting best practices for many programming paradigms.
- Eg: OOP adds great degree of simplicity, by employing abstraction & information hiding.
- Breaking the product to be developed into meaningful pieces (real life parts) makes software product simple.

* RELIABILITY :

- This can be achieved by sticking to standard processes for software construction
- During reviews, if any defects are found, they can be fixed easily if source code is neat, simple & clear, because critical defects leads to products that are of no use to end users.
- Reliable source code can be achieved by designing the software product with future enhancement into consideration as well as by having a solid structure on which the software product can be built.
- Writing pieces of code based on this structure, will lead to little chances of defects entering into source code.
- During enhancements the existing structure is not able to accomodate load of additional source code & this structure becomes shaky.

* In this case, development team can go for restructuring the software design & then write code based on new structure.

* SAFETY :

- Software products should provide for safety as faulty machine operation or exposure to a harmful environment might put human lives under risk.
- In such cases, software product must be ensured to operate correctly & chances of error are less (0.00001%)
- In some industries, fool proof software products are needed to ensure safety to human lives & these must have inbuilt safety harnesses.
- Eg: Medicine, Healthcare, Road safety & Hazardous material handling environments.

* MAINTAINABILITY :

- Maintenance costs are more than 70% of all costs including software development, implementation & maintenance.
- To limit maintenance cost, source code should be maintainable. so that it will be easy to change the source code for fixing defects during maintenance.

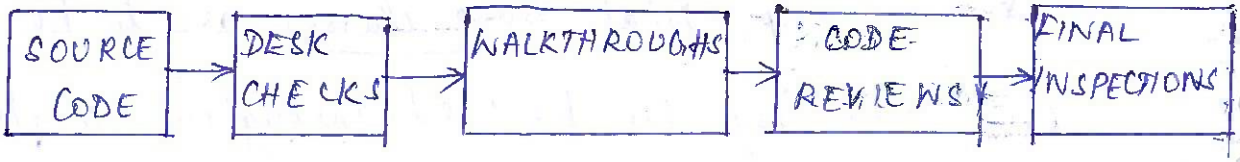
CODING FRAMENORK (CF)

* Coding framework involves setting up of an infrastructure based on which construction can take place & in case of software, CF should ensure a consistent coding production with standard code that will be easy to debug & test.
* In OOP, what base classes are to be made so that they can be used throughout construction is a part of the CF
* CF in general allows construction of common infrastructure of basic functionality which can be extended later by developers.
* This way of working increases productivity & allows for a robust & well structured software product (house building on a common solid foundation)

REVIEWS (QUALITY CONTROL)

* 70% of software defects arise from faulty software code
* Any construction rework means wasting a lot of effort & it is a fact that it is cheaper to fix any defects found during construction at the phase level itself.

* If those defects are allowed to go in software testing (next phase), then fixing defects becomes costlier.
* Hence review of software code & fixing defects is very important.
* Review techniques include Deskchecks, Walkthroughs, Code Reviews, Inspections, etc & these ensure quality of written code.



* These kinds of reviews are done at different stages in software code writing & also serve different purposes.
* While inspections provide final go/no go decision for approval of a piece of code, other methods are less formal & are meant for removing defects instead of deciding whether a piece of code is good enough or not.

* DESK CHECKS (PEER REVIEWS)

  - This is used when a complete review of the source code is not important.
  - Developer sends his piece of code to the designated team members who review the code & send feedback & comments to the developer as suggestions for improvement in the code.
  - Developer reads the feedback & may decide to incorporate or discard those suggestions.

* WALKTHROUGHS

- These are formal code reviews initiated by developer & sends an invitation for walkthrough to team members.
- At the meeting, the developer presents his method of coding & walks through his piece of code & the team members then make suggestions for improvement
- The developer can decide whether to incorporate those suggestion or discard them.

* CODE REVIEWS

- This is a formal method where the project manager calls for a meeting for code review of a developer.
- At the meeting, team members review the code & point out any code errors, defects or improper code logic for likely defects.
- An error log is also generated and is reviewed by the entire team.

* INSPECTIONS :

- This is the final review of software code in which it is decided whether to pass a piece of code for inclusion into the main software build.

CODING METHODS

* Many programming & coding methods were devised & evolved as a result.
* Early software products were of small size due to limited hardware capacity.
* With increasing hardware capacity, the size of software products has been increasing.
* Software product size affects the methods that can be used to construct specific sized software products
* Advancement in the field of computer science also allows discovery of better construction methods
* To address needs of different sized software product in tandem with advancement in computer science, different programming techniques evolved. These include structured programming, object-oriented programming, automatic code generation, test-driven development, etc.
* STRUCTURED PROGRAMMING (SP)
  - This became popular after mainframe computers became into existance.
  - Mainframe computers offered vast availability of computing power compared to primitive computers
  - Using SP, large programs, large programs could be constructed that could be used for making large

commercial & business applications.

- SP enabled programmers to store large pieces of code inside procedures & functions.
- These pieces of code could be called by any other procedures / functions.
- This enabled programmers to structure their code in an efficient way. Code stored inside procedures could be reused anywhere in the application by calling it.

* OBJECT ORIENTED PROGRAMMING

- In SP, data & structured code are seperate & accordingly are modeled separately.
- Objects on the other hand contain both data & structure.
- Eg: Car consits of chasis, engine, & wheels, etc
- Each of these objects has some specific properties & specific functions.
- When a real world objects are modeled with software system they use both data & structures into account in OOP.
- Unnecessary details about objects are hidden & in fact not available from outside.
- This kind of representation of objects makes them robust & a system built using them has relatively few problems

* AUTOMATIC CODE GENERATION

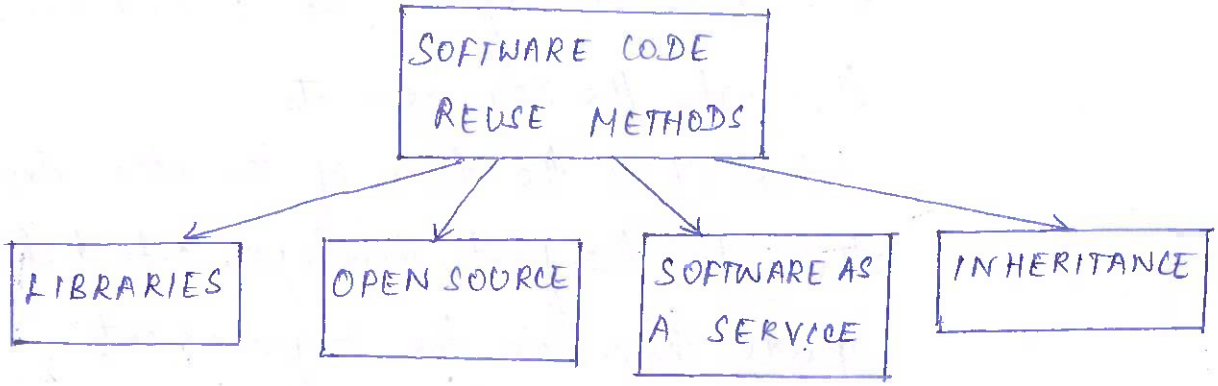
- Constructing & generating software code is a very labor intensive work.
- So automatic generation of software code is fascinating.
- There are business analyst platforms developed by many ERP software vendors that generate code automatically when analysts configure the product.
- These analyst platforms are built using any of the software product development methodologies.
- The generated code is specific to the platform & runs on the device (hardware & software environment) for which code is generated.
- Any code consists of many construction unit types. Some of these code types include control statements such as loop statements, if statements, etc., & data base access, etc.
- Generating all of the software code required to build a software application is still difficult.
- But companies like Sun Microsystems are working to develop such a system.

* SOFTWARE CODE REUSE

- This can reduce labor intensive nature of writing source code.
- Making block of source code to create a functionality or general utility library & using it at places in

source code wherever this functionality is required is an example of code reuse.

- In procedural programming, this is achieved by creating special functions & utility libraries and using them in the source code.
- In OOP, it is done at a more advanced level, where classes containing functions & data themselves can not only be reused but the classes can also be modified by creating child classes & using them in source code



- Apart from creating & using libraries & general purpose classes for code reuse, a more potential code reuse model has evolved recently. It is known as Service Oriented Architecture.

* TEST DRIVEN DEVELOPMENT

- Best used with iteration based projects especially with eXtreme Programming technique (XP)
- Before developers start writing source code, they create test cases & run the tests to see if they run properly & their logic is working.

- Once it is proved that their logic is perfect, only then they write the source code.
- So here, tests drive software development, & hence it is appropriately named test-driven development

* PAIR PROGRAMMING (PP)
  - It is a quality driven development technique employed in XP development model.
  - Here each development task is assigned to 2 developers, while one developer writes the code, the other developer sits behind him & guides him through the requirements.
  - When it is the turn of the other developer to write the code, the first developer sits behind him & guides him on the requirements.
  - So developers take turns for the coding & coaching work & this makes sure that each developer understands the big picture & helps them to write better code with lesser defects.

CONFIGURATION MANAGEMENT (CM)

* It plays an important role in the construction phase.
* Due to changes in requirements & design, an already developed source code needs to be changed. So it happens that the development team ends up with

many versions of source code during the project.

* If version control management is not handled properly, then many developers may start working on a wrong version of source code & that a lot of reworks may be needed in the end.
* There is one more dimension to CM, during construction as many software builds are maintained for different versions of the product being developed.
* These builds can break if a bad piece of code is checked into the build by any developer.
* When build is broken, then no other developer can check in his code.
* Here development is halted until the build is rebuilt with the correct code.
* If distributed teams located at far-flung locations with different time zones are involved and a central build is being maintained CM becomes more complicated
* It will be difficult to communicate & manage the build process in such scenario.
* In such scenarios, <u>smoke test</u> application can be deployed, which can run whenever a new code is checked-in in the build.
* If the smoke test fails, that means the build has failed & thus the automated system can email the

build information to concerned people

* If the build fails, then the developer who had checked-in in the code gets the message & immediately tries to fix the build.
* Once the build is fixed, then other developers can check-in their code.

SOFTWARE CONSTRUCTION ARTIFACTS

* This (SC) phase is one of the most labor intensive phases in SDLC as it generates the complete source code of the application
* Apart from source code, documentation is also made so that when any maintenance is required on the built application, the source code could be well understood & changing any source code will be easy.
* Review reports are also generated after reviews are conducted.

UNIT - IV

* Introduction to Testing
* Verification
* Validation
* Test Strategy & Planning
* Test Project Monitoring & Control
* Design - Master Test Plan, Types
* Test Case Management
* Test Case Reporting.

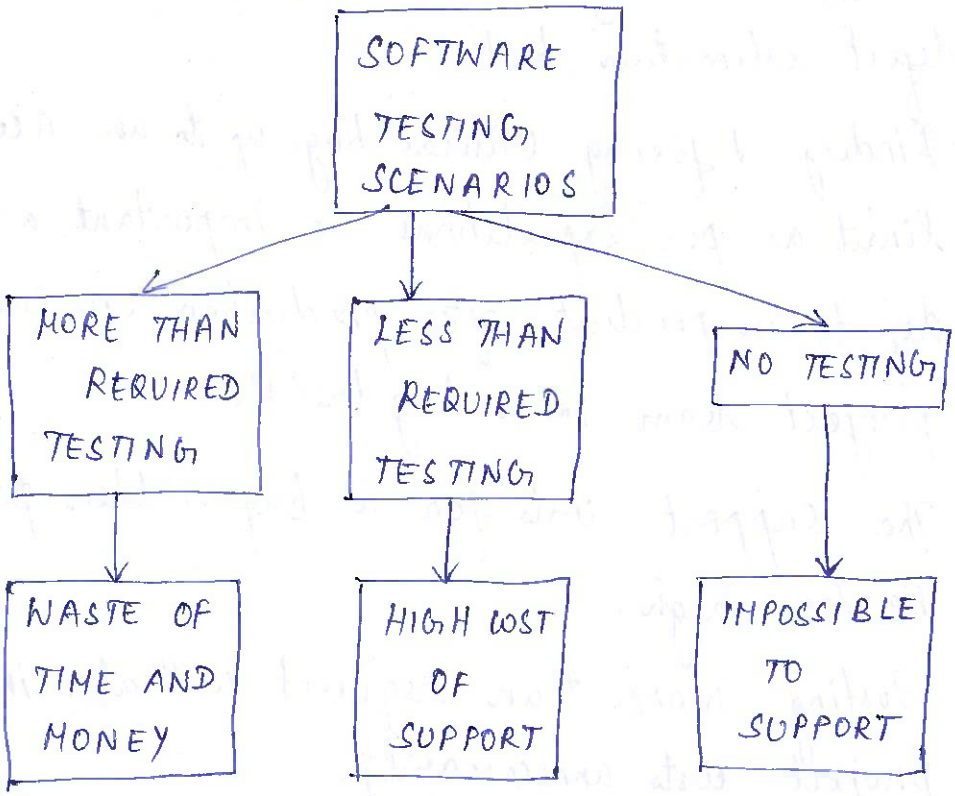
TB : 2
CH : 13
Pg. No. : 187 - 196.

UNIT - IV

INTRODUCTION TO TESTING

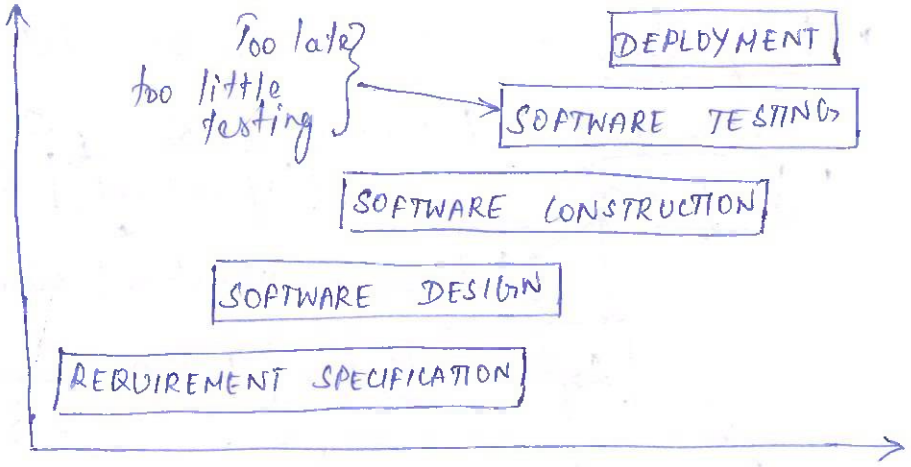
* Though exact number of defects in a software product is difficult to find, it can be predicted using some defect estimation tools.
* Finding & fixing critical bugs up to an acceptable limit as per expectations is important as identifying defects in product after production can end the project team in a big trouble.
* The support costs for a bug ridden product will be too high.
* Testing more than required will also increase project costs unnecessarily.
* When project starts, the customer specifies what level of quality for the product is expected.
* The project manager needs to first make sure that the processes to be followed for building the product are at least so good that the produced product will have a certain level of quality with a certain level of defects.
* Then he should have a test plan such that the product defects are further reduced by finding defects & fixing them.

* So the testing phase must be well planned with required budget, schedule & testing processes that will ensure that a certain number of critical defects are identified & fixed.



PROBLEMS WITH TRADITIONAL DEVELOPMENT MODEL

* Traditionally, software testing was done only after software was constructed.
* This limited the scope of software testing in SDLC
* By the time software was constructed already faulty requirement specifications & faulty software design had resulted in defect ridden software.
* Removing all defects originating from different phases of the project in one go is a huge challenge.

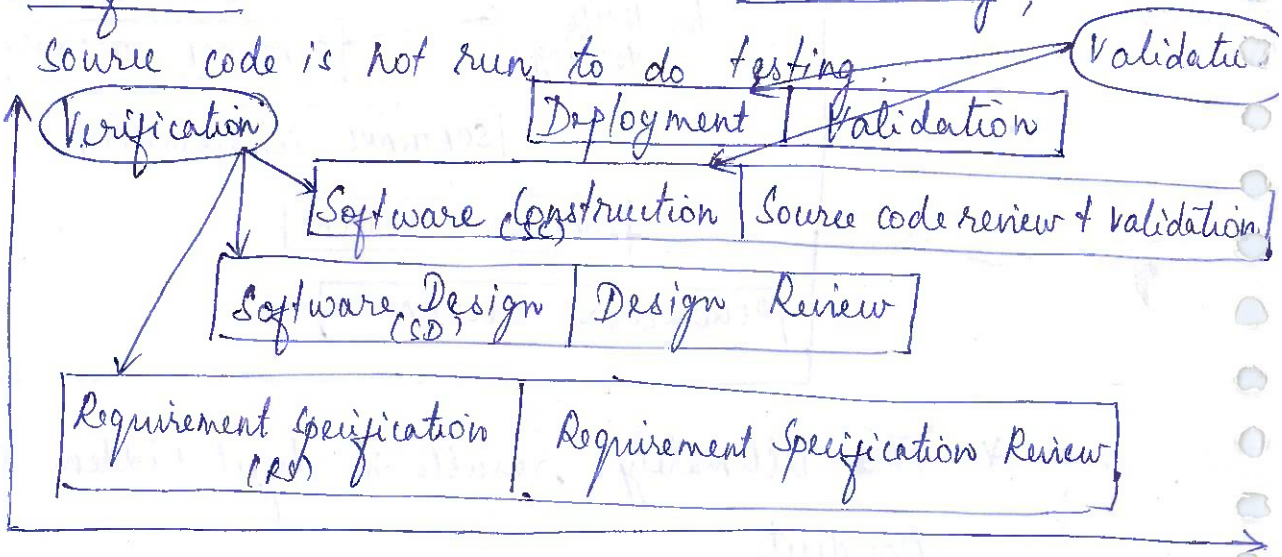


* This ultimately results in defect ridden software products.
* It will be costly to remove defects late in the life cycle & will also involve considerable amount of time to detect & fix all the defects. This is likely infeasible.
* Better approaches should be devised to make better quality software products.

VERIFICATION & VALIDATION

* A better approach to traditional approach is verification & validation
* Software Testing in most quality standards is divided into Validation & Verification.
* While verification implies that the developed software is working as intended by checking the requirement specification, design, source code, etc, in static mode, validation implies that the software has been validated to be working after running it & checking whether all functionality meets the requirements.

* Verification are also known as Static Testing, since the source code is not run to do testing.



* In the above fig. RS, D, SC (code) during SDLC is tested using static methods.
* The Requirement specifications are reviewed for completeness, clarity, design ability, testability, etc.
* The Software design is reviewed for robustness, security, implementability, scalability, complexity, etc.
* The Source code is reviewed for dead code, unused variables, faulty logic, constructs, etc.
* Once source code is ready to be run as a system, validation testing can be started.
* Validation testing is also known as Dynamic Testing. as in this case, the source code is actually run to determine that it is running as per specifications.
* During validation, unit, integration, system & finally user acceptance testing are performed.

* Unit Testing is done to ensure each unit piece of source code is free from defects.
* Once unit testing is done, then this piece of code is integrated with the main source code build.
* But before integrating to the main build, it is strong advisable to do local integration testing on the developer's own computer.
* Only when the source code runs smoothly & all integration tests pass, the source code should be integrated with the main build.
* When all the of the source code is thus integrated, the main build is ready for system testing.
* All system tests are then performed & defects are fixed.
* When the system testing is over, and in fact the software product is shipped to customers, they do user acceptance testing.

TEST STRATEGY AND PLANNING

* Usually testing is vast and it is considered as a seperate project itself.
* In such cases, testing is known as an independent verification & validation project.
* Hence seperate project plan is made for that project & is linked to the parent software development project.

* There are many techniques available to execute software test projects & it depends on the type of test project.
* However most test projects must have a test plan & a test strategy before the project is ready for execution.
* Often due to time constraints, testing cycles are cut short by project managers, leading to a half-tested product & hence large number of product defects are left undetected & ultimately end users discover these defects.
* Fixing defects at this stage is costly. Moreover, they cannot be fixed one at a time. They are to be taken in batches & are incorporated in maintenance project plans.
* This leads to excessive costs in maintaining the software
* It is cheaper to trap those bugs during the testing cycle & fix them.
* It is said "Testing costs money but not testing costs more"
* Test strategies should include things like test prioritization, automation strategy, risk analysis, etc.
* Test planning should include a work breakdown structure, requirement review, resource allocation, effort estimation, tools selection, setting up communication channels, etc.

* TEST PRIORITIZATION (TP)

- TP should be made even before testing starts.
- All parts of software product will not be used by end users with same intensity, some parts will be used extensively & others will be seldom used.
- So the extensively used parts should not have any defects at all & hence must be tested thoroughly.
- To achieve this, prioritization must be applied, with high priority given to tests done for critical parts of product & low priority on uncritical parts.
- Testing is done for high priority areas first & once thoroughly done on these parts, then start testing low priority areas.

* RISK MANAGEMENT

- Test manager should also do plan for all known risks that could impact the test project.
- If proper risk mitigation plan is not done & a mishap occurs, then the test project schedule could be jeopardized, costs could escalate, and/or quality could go down.
- Some risks that can have severe adverse impact on a test project include (i) unrealistic schedule (ii) resource unavailability (iii) skill unavailability

(iv) frequent requirement changes, etc..

- Requirement changes pose a serious threat to testing effort, because for each requirement change, the whole test plan gets changed. The test team has to revise its schedule for additional work as well as to access impact of the change on the test cases they have to recreate.
- Some test engineers estimate much less effort than it actually should be. In that case, the test manager would have trouble in explaining why testing is taking more the scheduled time schedule. In such cases, even after loading testing engineers more than 150%, the testing cycle gets delayed. This also happens because the marketing team agrees on unrealistic schedules with the customer, in order to get the project. Even the test manager at that time feels that somehow he will manage it, but later on it proves impossible to achieve.
- When software development market along with the software testing market, is hot, software professionals have many job offers in hand, they leave the project at short notice & the test manager has to find a replacement fast
- Sometimes a project may have some kind of testing for which test professionals are hard to find. &
- In both of the above cases, test manager may not be able to start tasks in need of adequate resources.

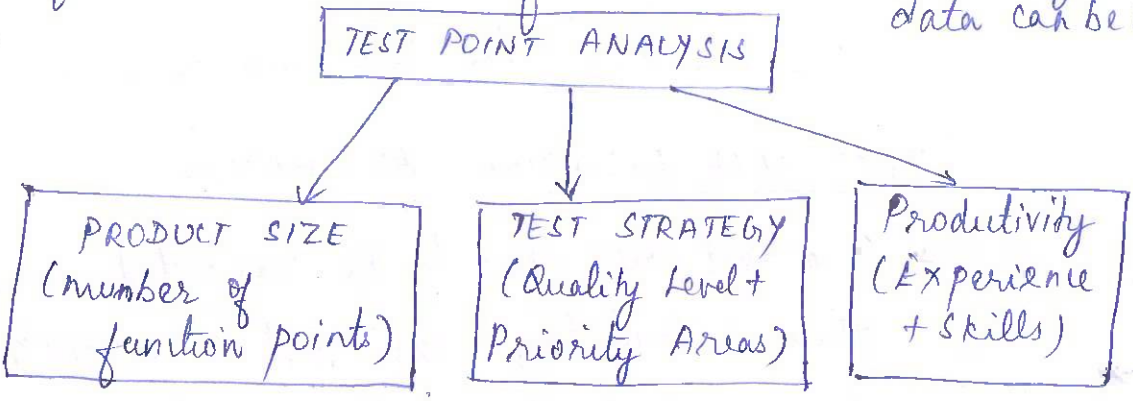
- For test professional resources, a good alternative resource planning is required. The test manager, in consultation with human resource manager, keep a line of professionals who may join in case of need in projec
- For scheduling problems, the test manager has to ensure in advance that schedules do not get affected. He has to keep a buffer in the schedule for any eventuality
- To keep a tab on the project budget, the test manager has to ensure that the schedule is not unrealistic + also has to load his test engineers appropriately. If some test engineers are not loaded adequately, then project costs may go higher. For this reason, if any test professionals do not have enough assignments on one project, they should be assigned work from other projects.

* EFFORT ESTIMATION

- For making scheduling, resource planning & budget for a test project, the test manager should make a good effort estimate.
- Effort estimate should include information such as a) project size b) productivity + c) test strategy.
- While project size + test strategy information comes after consultation with the customer, the productivity figure comes from experience & knowledge of the team members of the project team.
- Delphi technique uses brainstorming sessions to arrive at effort estimate figures after discussing the project detail with the project team.

- Initial estimates from each team member are then discussed with other team members in an open environment to condense it into final estimate for each project task.
- In an experience - based technique, instead of group sessions, the test manager meets each team member & asks him his estimate for the project work he has been assigned.
- Effort estimation is one area where no test manager can have a good grasp, at the initial stages of the project as many details are not clear but as the project unfolds, after executing some of its related tasks, things become clearer, & the test manager can comfortably give an EE for the remaining project tasks
- Project stakeholders would want to know cost estimate at the very beginning & when the project would be delivered. Unfortunately, test managers are not equipped to provide accurate schedule & costs for the projects at those initial stages
- Best solution is to find a relatively objective method of effort estimation & provide the requested information.
- Test Point Analysis : TPA
  - Methods for EE include test point analysis, Wideband Delphi technique & experience based estimation.

- Here in TPA, 3 inputs required are project size, test strategy + productivity.
- Project size is determined by calculating the number of test points (TPs) in the software application.
- Test points, in turn, are calculated from function points
- The number of function points (FPs) is calculated from the number of functions & function complexity.
- If no. of FPs in an application has been calculated by the development team, then TPs are calculated from available FP information. Otherwise rough FP data can be used



- A test strategy is derived from 2 pieces of information from the customer (i) what will be the quality level for the application (ii) which features of the application will be used most frequently.
- Productivity is derived from knowledge + experience of the test team members.
- Past productivity data from previously executed projects make sense to make productivity figures more realistic.
- In case of iterative development, testing cycles will be short + iterative in nature. Test manager makes test effort calculations accordingly

TEST AUTOMATION

* Most testing tasks are done manually, as they are still difficult to automate.
* But Automation helps in evaluation & hence care should be taken not to automate blindly as initial effort for automation is more than manual testing.
* Testing tasks include a) requirements & design document review b) test case scenario creation c) test case creation d) test case execution e) test case management & f) defect tracking.
* Of all these, good automation tools exist only for test case execution & test case management.

TEST CASE EXECUTION AUTOMATION

* If a test case has to be executed only a few times, then automating the test case will be more expensive compared to manually executing the test case.
* Only if it is executed atleast 13 times it make sense to automate
* A test case will be executed more than once because in software development, new versions of the software gets developed to cater to the needs of the market with new versions holding old features in addition to new features.
* The older version of the s/w was tested using existing or newly created test cases at that time & with new features added, it should be ensured that old features still work.
* Old functional tests now become regression test cases (RTC) & the suite of RTC keeps increasing with newer

* At some point the RT suite becomes so large that manually executing tests becomes a liability (no body wants large no of repeated executions of test cases)
* Most software vendors have minor releases of their product each quarter, so an ever increasing suite of RTC has to be run each quarter + hence takes a considerable amount of time
* As the software has to be released fast, the project manager cannot wait just because RTC are still being implemented + hence automating the whole suite of RTC is going to be profitable.
* Current trend for automation is to create a <u>keyword framework</u> as follows.
  (i) For each major function create a keyword
  (ii) Write automation script for that function + then save the fn. with same name as the keyword.
  (iii) After all functions are created, relate functions with test cases that would have been already created before automation scripts were written.
  (iv) Run the scripts (it will cover test cases + same as done manually)

  This way of inducing automation is known as <u>KeyWord Driven Automation Framework</u>

  <u>Adv:</u>

  (i) Allows for reuse of script + make automation creation modular.
  (ii) Makes maintenance of scripts easier.
  (iii) If any Test case changed, the whole script does not have to be changed (the script for which keyword was affected only changes).

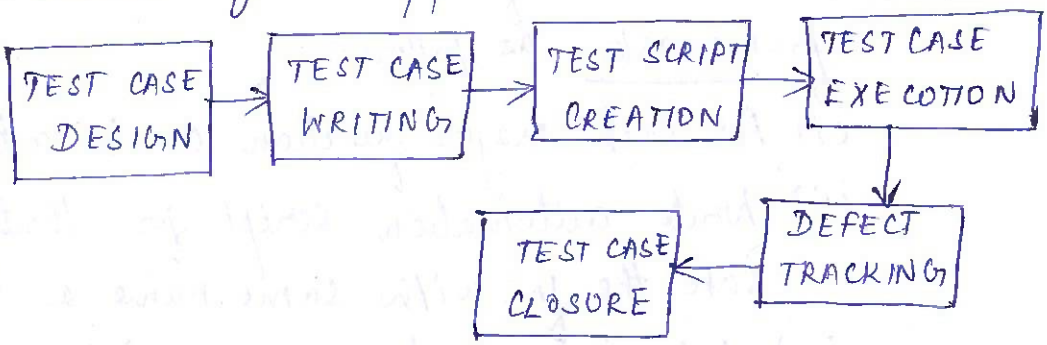
(TCM)

TEST CASE MANAGEMENT AUTOMATION

* TCM is good for automation & some tools exist to facilitate it.
* They allow keeping many versions of test cases & a repository of automation scripts, which allows team located at many sites to work more effectively.

TEST PROJECT MONITORING & CONTROL (TPMC)

* Test projects involve a large variety of activities including test case design, test case management, test case automation, test execution, defect tracking, verifying, & validating the application under test.



* TEST CASE DESIGN
  - This ensures test cases are designed properly.
  - This is done by Test Manager & he ensures which kind of tests are to be designed, how many test cases have to be written for particular modules & which test areas are priority areas.
  - Test Types :

    An application may have to be tested for (i) functionality (ii) performance (iii) usability (iv) compatibility & many others.

▲ For each kind of testing, a set of test cases has to be written & executed + finally the system should be verified & validated. For applications with many versions, regression tests also have to be performed. Though managing all these kinds of testing is a big task, a good test manager will first divide the testing tasks on basis of types & tasks can be further divided into modules so that he can allocate testing tasks to test engineers appropriately.

▲ Tests can also be segregated depending on project phase, (i) system testing (ii) integration testing (iii) acceptance (end users) testing.

* (Integration testing is performed when the application needs to be integrated with any other external application)

* TEST CASE MANAGEMENT (TCM)

▲ It involves (i) managing different versions of test case (ii) keeping tracks of changes in them. (iii) keeping seperate repository of test cases based on type of tests (iv) creating & managing automation scripts.

* TEST BED PREPARATION (TBP)

▲ TBP involves installing the application on a machine that is accessible to all test teams but should be free from interference from unauthorized users

- It should resemble the production environment as closely as possible, including software + hardware configurations.
- For all types of testing, it is very important that the "application under test" (AUT) should be tested under an environment that is close to the environment under which the proposed application will be deployed for production. This makes test bed preparation very important.
- The application should be installed on a dedicated server that has same configuration as proposed production environment. This server should be used only for the purpose of testing.
- It should be installed centrally, so that even distributed teams, contractors or service providers can easily access it using remote desktop sharing or any peer to peer networking protocol over the Internet. Better if application can be accessed directly over Internet.
- No testing should be done on applications deployed on local test engineer's machine. But acceptable to have a local copy of application for preliminary testing, but not when defects are to be logged & verified by many people as it is important to reproduce the defect when developer or any one else asks for it.
- In case of disputes, if a defect cannot be reproduced, then it becomes difficult for the test team to justify why a defect has been logged when others cannot reproduce

- This is the reason for which test bed should be prepared very carefully & kept as isolated from any other environment as much as possible to preserve its integrity.
- The test data preparation is a very tricky affair. The test data should closely resemble what the end users use in their daily transactions. For this, the test team can get some business data already used by end users. The test bed should be populated with a similar kind of data.

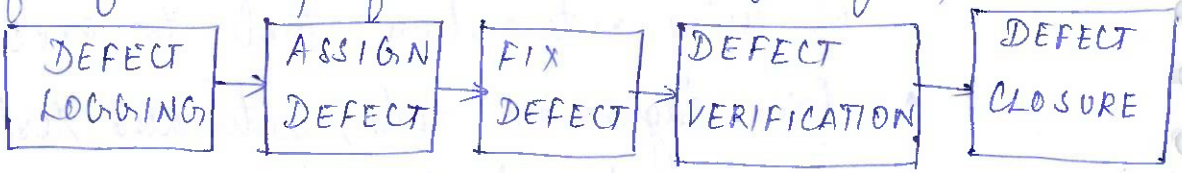
* TEST CASE EXECUTION

- It involves executing prepared test cases manually or using automation tools to execute them.
- For regression tests, automated test execution is a preferred method. After each test case is executed, it may pass or fail. If it fails then defects have to be logged.
- Exit criteria for test case execution cycle are generally defined in advance & when certain level of quality of the application is reached, then test execution stops.

* DEFECT TRACKING (DT)

- It is one of the most important activities in a test project. During DT it is ensured that defects are logged & get fixed.

- All defects & their fixing are tracked carefully.
- Defect count per hour per day is a common way of measuring performance of a test team. If the testing is done for an In-house software product, traditionally, it is ~~used~~ not a performance evaluation measurement.
- What really counts are the number of defects found in production when the software product was deployed & used by end users.
- But it is too late a performance measurement. What if many of the test team members left before the product was deployed? In fact this is a reality, given the high attrition rate (20% in many corporations) of software professionals. Once they are gone, there is no



point in measuring the performance.
- Hence a better measurement would allow for more immediate results, which is achieved by measuring the defect count per hour per day.
- Then there is the case of outsourced test projects. If the contract is only for testing upto deployment & not afterward, then measurement does not make sense after the contract has ended.
- A good defect tracking application should be deployed on a central server that is accessible to

include test plan document, test strategy document, test cases, test cycle logs, defect list, verification & validation reports and product certification.

- Management Artifacts
  - Customers are concerned not only with project cost & schedule, but they are also concerned with critical defects, which the test team has either detected or not.
  - So the management artifacts (metrics) include project cost compliance, project schedule compliance and quality (number of critical defects caught versus number of critical defects which went into production).
  - Some other management artifacts include traceability matrix, defect density rate, resource loading, etc.

all test & deployment teams.

- Each defect should be logged in such a way that it could be understood by both development & testing teams.
- Generally, the defects should be reproducible, but in many instances, this is difficult & in such cases, a good resolution should be made by the test & development managers.

* TEST REPORTING

- During execution of a test project, many initial & final reports are made. But status reports also need to be made.
- Test reports include test planning reports, test strategy reports, requirement document review comments, number of test cases created, automation scripts created, test execution cycle reports, defect tracking reports, etc.
- Some other reports include traceability matrix reports, defect density, test execution rate, test creation rate, test automation script writing rate, etc.

* TEST ARTIFACTS

- Software testing involves making a test strategy, test project plan, resource requirements, test case repository creation, running test cycles, defect tracking, bug verification & validation & finally certifying the developed product. So the test artifacts

UNIT - V

* Product Release
* Product Release Management
* Implementation
* User Training,
* Maintenance Introduction
* Maintenance Types - {Corrective, Adaptive, Perfective, Preventive
* Maintenance Cost
* Maintenance Process
* Maintenance Life Cycle
* Software Release
* Software Maintenance
* Maintenance Techniques

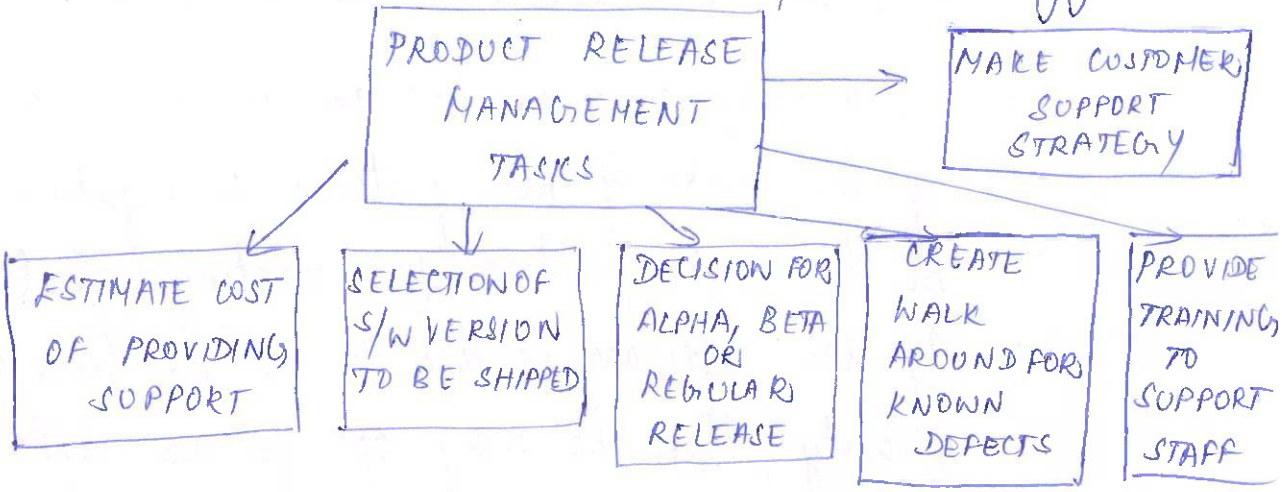
TB : 2
CH : 14
Pg. No. : 201 - 210

UNIT - V

PRODUCT RELEASE & MAINTENANCE

INTRODUCTION

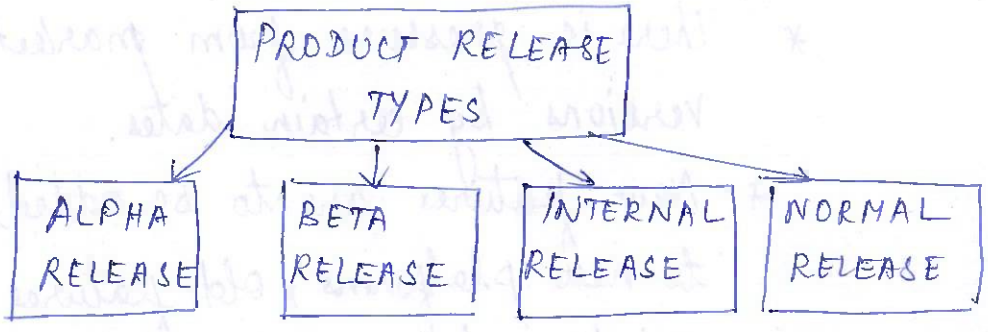
* The completed software product has to be taken to the customer's site and implemented for users utilization.
* It should be ensured that all tasks are completed eg: which version should be released, walk again for known bugs, etc, custome support strategy



PRODUCT RELEASE MANAGEMENT (PRM)

* Project teams working for software product vendors struggle to keep pace with release of the software product.
* There is pressure from market to launch new versions by certain dates.
* New features are to be added, porting the product to new platforms, old features are to be enhanced, existing bugs are to be removed & yet it has to meet the deadline.

* Good product release strategies should be incorporated
* Depending on the situation, the project manager may need to convince the management to cut short some of the product features, to meet the deadline as well as meet quality standards.
* Sometimes project manager may be blamed for its poor quality issues.
* Bargaining also has to be done for other requirements of bug fixes, feature enhancements, etc.
* If quality concerns are paramount, then moving some of the tasks of new features to a future release may be the best solution for meeting quality standards.
* If the software vendor is not too sure about product quality, then he may opt for an alpha/beta release.
* In that case, the product will be released only among a few selected groups & not in the market as a whole.
* The controlled product is the best option in these conditions



* PRM in fact is a dynamic environment & if properly planned is not done at a minute level

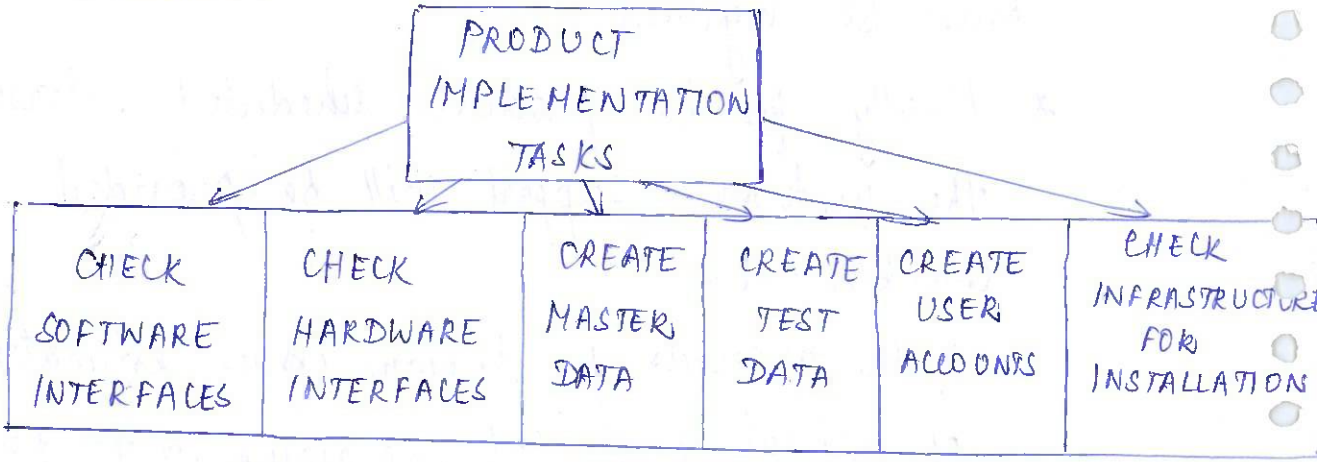
and constant vigilance is not applied over project activities, then a huge mess can be created & there will be no time to clear it. Hence project manager must be vigilant.

* Finally for the product's scheduled release, how the customer support will be provided should be chalked out.
* Walk arounds for known issues, estimated number of critical bugs still remaining in the product, training for the support staff, etc. should be done.
* The cost of support, depending on the number of estimated users, walk arounds & remaining bugs should be figured out.
* These measure will ensure that the product is transitioned into market without facing major difficulties.

PRODUCT IMPLEMENTATION

* The product that has been developed & thoroughly tested now needs to be implemented at customer site.
* All master data must be prepared & test transaction data for testing the implemented product
* All required hardware & software should be readily available for installing the software product.

* It must be made sure that the hardware & software interfaces for integrating the product with existing legacy systems & infrastructure are developed & tested.



* Care should be taken that the product will run smoothly on customer premises without any interference with their existing applications.
* Often project teams run into problems during implementations due to unforeseen circumstances or negligence on part of the production team or customer's team. Therefore, prepare a list of your own requirements & hand it over to your customer's support team so that they are prepared when you arrive for implementation.

USER TRAINING

* It should be checked if the user manual prepared by the team is upto date & in synch with the version of the software product that is to be implemented at the customer site.
* It is not possible to provide training to all users.

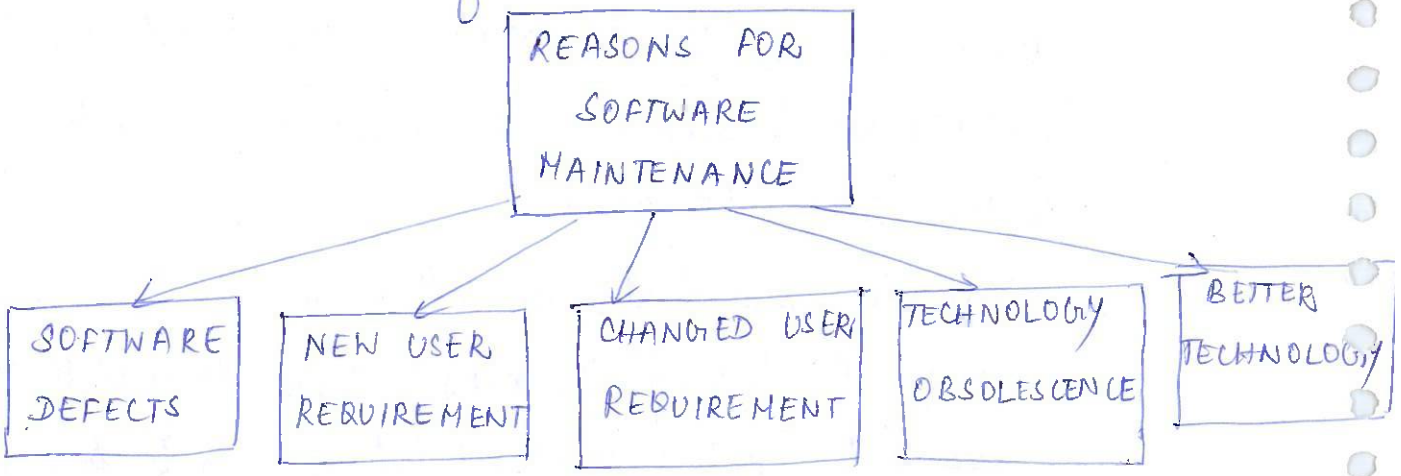
* A list of roles that are needed to operate the product should be prepared.
* This list should be given to the end users & ask them to select one user per role to receive the training
* In addition to user manual, a tutorial should also be prepared to include probable scenarios that may arise during the operation of the product. This will provide a step-by-step guide for using the product under those scenarios.
  * This is important, for otherwise if the users do not learn it during training, then they will contact later after implementation & ask to provide information as to how to use the product in those circumstances.
  * This might lead to a waste of support team's time
  * Hence it is better to train them, during user training, rather than face user requests later

MAINTENANCE INTRODUCTION

* Software products do not age or wear out like physical products.
* There are some factors which make it absolutely necessary.

(i) Technology Obsolescence : The software platform (operating system, medium of user, interface) or the hardware platform on which the software product runs gets obsolete

(ii) Software Defects : There are major software defects in the product & it is difficult to operate. For this reason a software patch may be needed to be applied so that these defects are removed



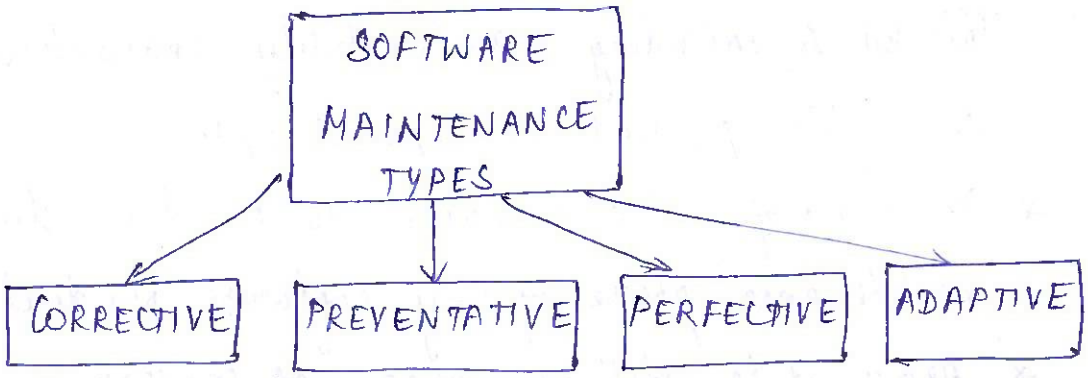
(iii) Change in User Requirements : The business organization that was using the software product has seen a change in business transactions or business workflows that are not supported by the software product.

* More than 70% of all costs associated with software product development, implementation & support & maintenance is consumed in the activities of supporting & maintaining software products.
* Something must be done to minimize support & maintenance costs as compared to development & implementation costs.
* Ways should be found to build such a software product.

* This led to including maintainability characteristics during the entire product development cycle.
* Yet a lot of work remains to be done during the maintenance phase of any software product.
* Hence it is still an area of concern.

MAINTENANCE TYPES.

* Software maintenance is of 4 types:
  (i) Corrective (ii) Adaptive (iii) Preventive (iv) Perfective maintenance.
* If the software has some defects, then it will take a corrective maintenance to rectify it.
* If there are some changes in the operating (system) environment of the software product, then the product can be made useful by doing adaptive maintenance.
* If there is an insecurity that although the product is running fine, in future we may have difficulty in using it, then preventive maintenance is employed.
* If there are some deficiencies in the product that must be rectified, then preventive maintenance will fit the bill.



CORRECTIVE

* Even after thorough reviews & testing, the software product contains many defects when it goes into production. These defects are uncovered as users start using the application
* They are logged with the support staff & after a sizable number of errors are detected, the software vendor instructs his maintenance team to create a patch to rectify them.
* The maintenance team then makes a plan & fixes those defects.
* After application of the patch containing the fixes, the software starts running without these defects.

ADAPTIVE

* The operating environment in which a software product runs in operation includes the hardware & software platform as well as the interfaces for human & other machine instructions.
* If any of these change over time, it becomes difficult to run the software product.

* In such cases, it becomes necessary to do adaptive maintenance so that the software product becomes reusable.
* This kind of maintenance may involve changing the interface or porting the application to another hardware / software platform.

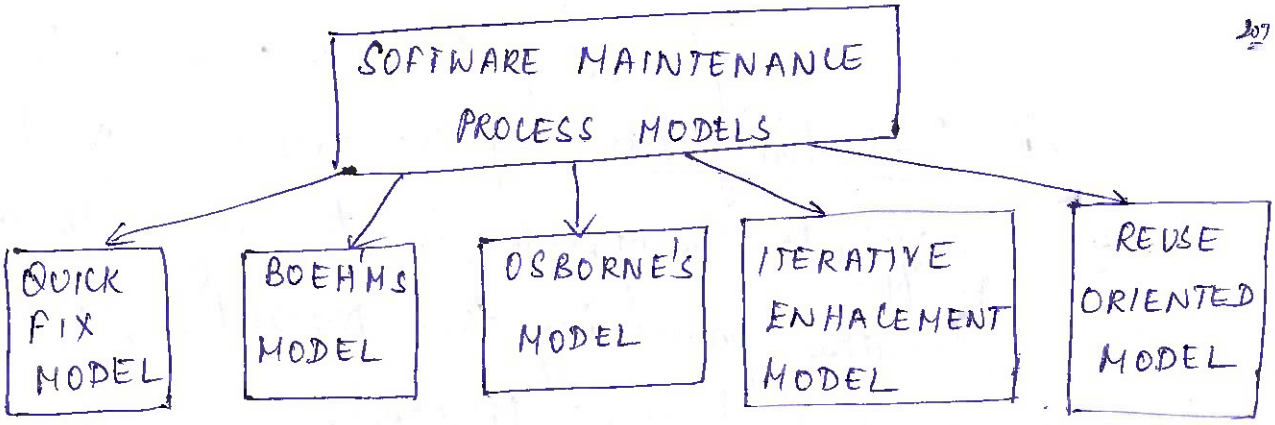
PERFECTIVE

* Generally after a lapse of time, there are likely changes in business or operative (~~environment~~), or there may be changes in (~~hardware / software~~) environment. (ie) a business transaction may have changed or an altogether new business transaction would be preferred.
* All these kind of requirements, require perfective maintenance.

PREVENTIVE

* Generally after a lapse of time, there are likely changes in business / operative environment or there may be changes in hardware / software environment.
* These changes are bound to occur & they affect the way the software product operates.
* Many of these changes can be perceived in advance
* In such cases, preventive maintenance on the software product can make sure that the

Boehm's model, Osborne's model, iterative enhancement model & reuse oriented model



Quick Fix Model: This is a simplest model. Whenever any defects with the software products are found they are immediately fixed. There is no planning involved in the whole process & it is mostly an ad hoc approach.

Boehm's Model: It is based on economic models & often involves calculating ROI for any planned maintenance. If ROI turns good, then it is carried out or else it is dropped.

Osborne's Model: Osborne realized that difficulties in carrying out maintenance work are due to gaps in communication. He proposed 4 steps to prevent this situation. He stated that a maintenance plan should include all change requests in the form of maintenance requirements. A quality assurance plan should accompany the maintenance plan. Metric should be developed to measure & assess

quality of work carried out during maintenance. Finally reviews should be held after maintenance work to assess quality of work done.

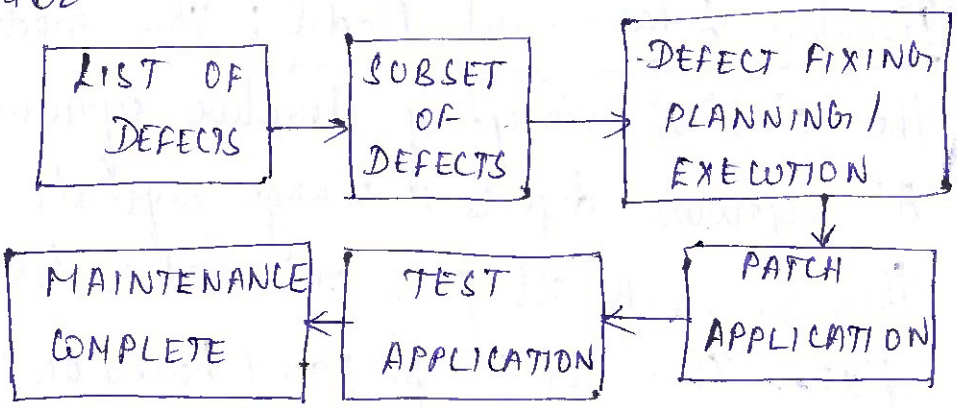
Iterative Enhancement Model: This model is based on the similar concept of iterative software development. All software defects & change requests are logged & then a small set from this list is taken for making fixes. This set is prepared based on the priority changes required. High priority fixes are done before low priority fixes.

Reuse Oriented Model: This type of process is adopted for component-based software products. For fixing any defects, existing components are analyzed & then the appropriate changes are made.

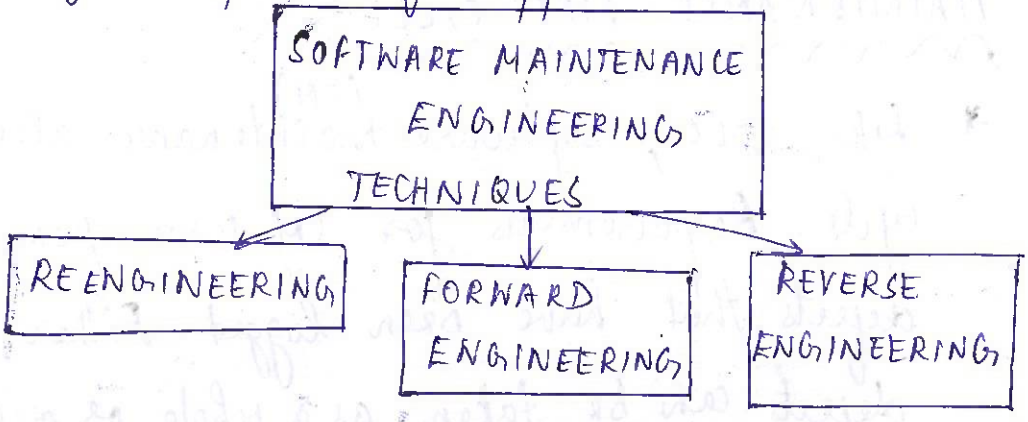
MAINTENANCE LIFE CYCLE:

* Like SDLC, software maintenance (SM) also has a life cycle. Requirements for SM come from the list of defects that have been logged. Either the list of defects can be taken as a whole or a subset of defects from the list can be taken for a fixing plan.
* It makes a lot of sense to go for an iterative approach similar to the concept of iterative software development.

* Hence highly visible, important & priority defects are fixed first & other defects which do not make much impact on operation of the products are tackled later



* In software maintenance life cycle, testing is a crucial phase. This phase also consumes a lot of time & effort. But the value addition in all this effort & time spent helps in reducing defects, which in the long run is a much cheaper alternative compared to no testing / cursory testing & later spending money in providing support.



MAINTENANCE TECHNIQUES

* Maintenance of software products sometimes becomes a tough proposition.

(8)

* There is no proper documentation that can be used for understanding how the product is designed & constructed
* Sometimes there is no documentation at all. Even if documentation is there, it is not upto date. This out-of-date documentation is not of much use for any maintenance work.
* Sometimes even if the documentation is up to date, the maintenance work ~~(depending~~ is difficult due to dirty design or construction work.
* All these situations call for some spefic techniques for maintenance work depending on the situation. Some of the common maintenance techniques include reengineering, reverse engineering & forward engineering

* REENGINEERING:

* It is also known as reuse engineering. This is a standard method for maintenance work for component-based software products.
* Details about all components in software products are well known.
* When any maintenance work is needed, from the list of defects, each defect is specifically analyzed to find out the root cause of the defect.
* Once this analysis is successful, then fixing that defect becomes easy.

SOFTWARE MAINTENANCE

* The software vendor keeps all of the production instances of its software product as its data center (also known as operation center).
* All previous versions of the software as well as the current working version of the software product run at this center as production instances of different version of their software product.
* The maintenance team makes sure that all versions of the product are available for users.
* They run sanity test scripts daily on all instances.
* If any problems are found, they are immediately resolved.
* These scripts are run at night. If problems are found, then, it is made sure that they are rectified before office hours start & people start using the application.
* In package software or custom built software that are not used in an SaaS environment, this kind of quick fix is not possible.
* So in those cases, a maintenance plan is made to fix all or most defects found by users during a time span of 3 months or more.
* But with SaaS environments, this kind of maintenance is not needed at all.
* All defects are quickly & easily fixed, without hampering work of end users.