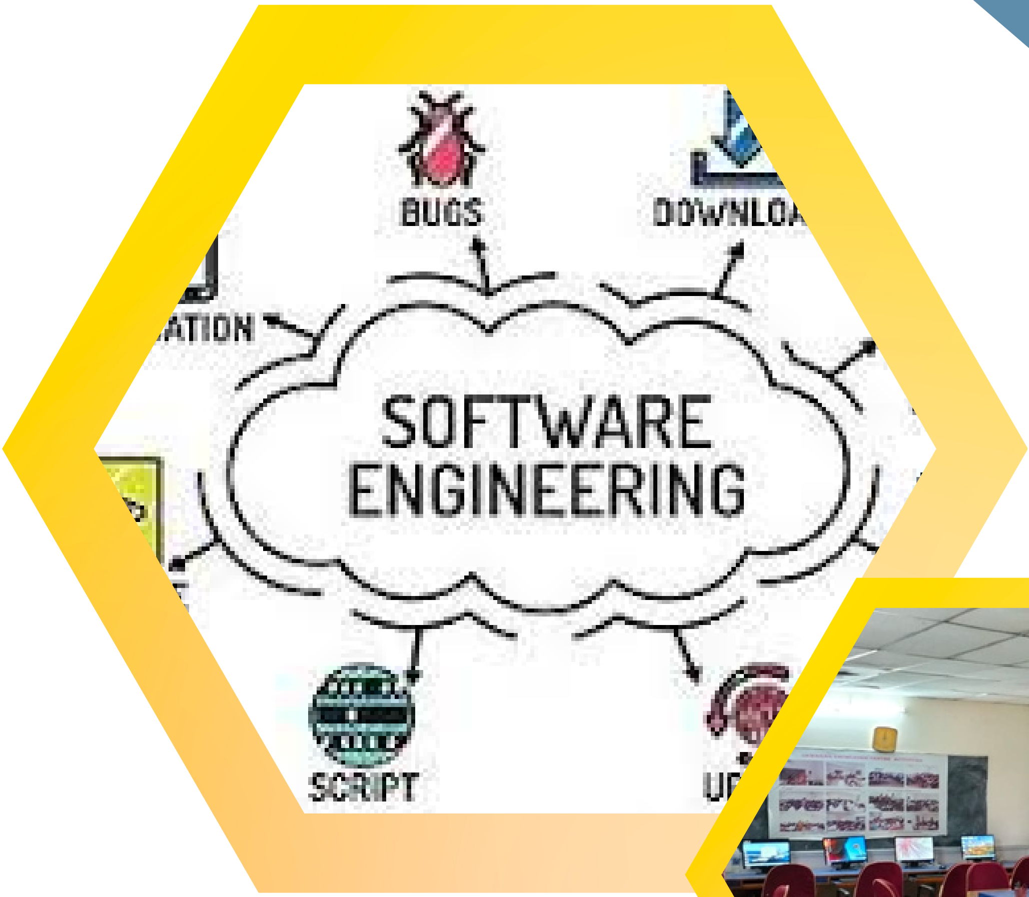
GOVERNMENT DEGREE COLLEGE

NARASANNAPETA-SRIKAKULAM DIST.-532421.

Accredited by NAAC 'B' Grade

(Affiliated to DR.B.R.Ambedkar University)

STUDY MATERIAL



Detartment of Computer Science

III YEAR V SEMESTER

Paper VI : Software Engineering

UNIT I

INTRODUCTION: Software Engineering Process paradigms - Project management - Process and Project Metrics – software estimation - Empirical estimation models - Planning - Risk analysis - Software project scheduling.

UNIT II

REQUIREMENTS ANALYSIS : Requirement Engineering Processes – Feasibility Study – Problem of Requirements – Software Requirement Analysis – Analysis Concepts and Principles – Analysis Process – Analysis Model

UNIT III

SOFTWARE DESIGN: Software design - Abstraction - Modularity - Software Architecture - Effective modular design - Cohesion and Coupling - Architectural design and Procedural design - Data flow oriented design.

UNIT IV

USER INTERFACE DESIGN AND REAL TIME SYSTEMS : User interface design - Human factors - Human computer interaction - Human - Computer Interface design - Interface design - Interface standards.

UNIT V

SOFTWARE QUALITY AND TESTING : Software Quality Assurance - Quality metrics - Software Reliability - Software testing - Path testing – Control Structures testing - Black Box testing - Integration, Validation and system testing - Reverse Engineering and Re-engineering.
CASE tools –projects management, tools - analysis and design tools – programming tools - integration and testing tool - Case studies.

UNIT - I

INTRODUCTION

UNIT - I

INTRODUCTION

- Software Engineering Process Paradigms
- Project Management
- Process and Project Metrics
- Software Estimation
- Empherical Estimation Models
- Planning
- Risk Analysis
- Software Project Scheduling

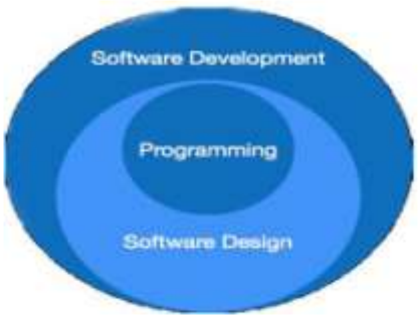
1. **Explain the Software Engineering Process paradigms?**

Software:

➢ **Software** is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

➢ The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Software Paradigms:



Programming paradigm is a subset of Software design paradigm which is further a subset of Software development paradigm.

1. Software Development Paradigm: It includes various researches and requirement gathering which helps the software product to build.

- ➢ Requirement gathering
- ➢ Software design
- ➢ Programming

2. Software Design Paradigm: This paradigm is a part of Software Development and includes

- ➢ Design
- ➢ Maintenance
- ➢ Programming

3. Programming Paradigm: This paradigm is related closely to programming aspect of software development. This includes Coding

- Testing
- Integration

1.WHAT IS SOFTWARE ENGINEERING?

Software Engineering: The practices that follow while developing an automated application
(or)

The technology encompasses (include) a process, a set of methods and array of tools that we call software engineering.

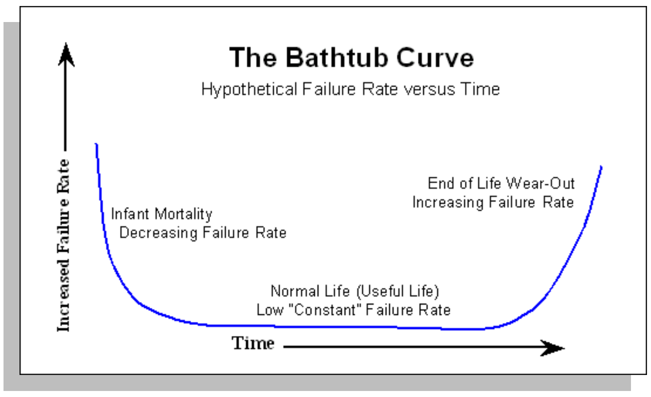
- **Software:** Software is collection of programs.
- **Program:** Program is Set of related statements.

Characteristics of software:

1. Software is developed but not manufactured.

2. "Does not wear out" that is can't be produced instantly. ie s/w does not degrade with time as hardware does.

3. Failure curve of hardware (Bath tub problem)as a function of time



- Practices in software engineering is(**list of the umbrella activities**)

1. Writing program using code reusability

2. ER-Diagram (or) UML - Diagrams

3. Testing

4. Quality Assurance

5. Deployment

6. Feedback

7 Risk management

8) Measurement

Note: Software transforms data into information.

2. Explain about Software Engineering Process Paradigms (Models)

Software Engineering: The practices that follow while developing an automated application.

Process models help in

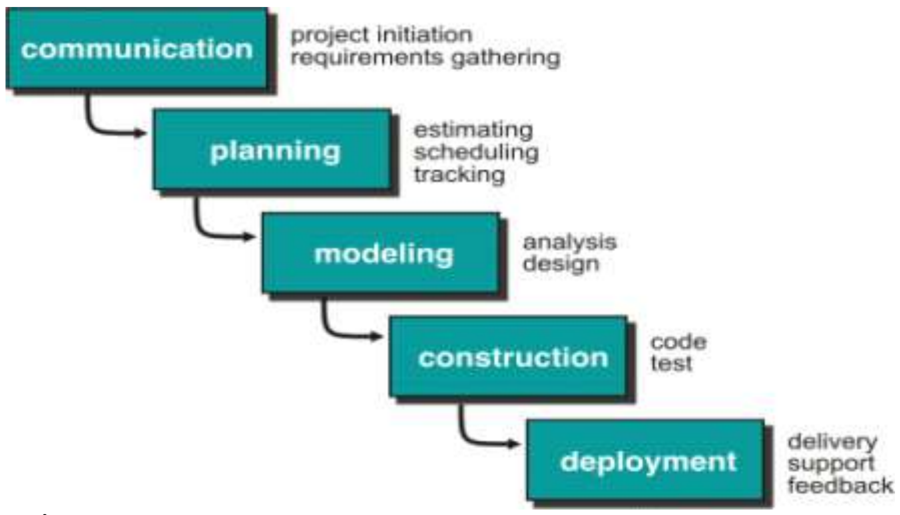
- ✓ Software development
- ✓ Guides software engineer through a set of frame work activities

Models

Prescriptive Process Models:

1) The Waterfall Model
2) Incremental Process Models
3) Evolutionary Process Models

THE WATERFALL MODEL (OR) SYSTEM DEVELOPMENT LIFE CYCLE (OR) CLASSICAL LIFE CYCLE (OR) LINEAR SEQUENTIAL



<u>**Communication:**</u>

- **Project initiation :** Feasibility of proposed system (system to be developed) is considered

a. **Economical feasibility:** Whether there is economy to develop the developed system.

b. **Technical feasibility**: Whether there is technology resources to develop the proposed system.

c. **Operational feasibility**: Whether day to day operations are possible.

d. **Behavioral feasibility:** Is the solutions to behavioral problems arises between the people.

Requirement gathering:

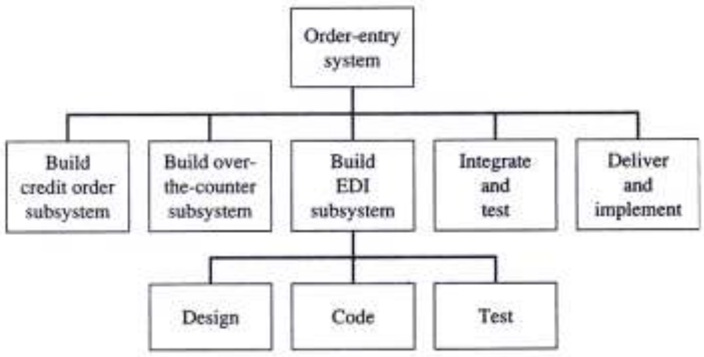
- ✓ Software engineer communicate with stack holder(client) to understand the information domain(how to develop product) and formulates design specifications.
- ✓ Effective communication between the software engineer and stack holder is needed.

Planing:

a. **Estimating:** Resources need for software development is to be estimated.

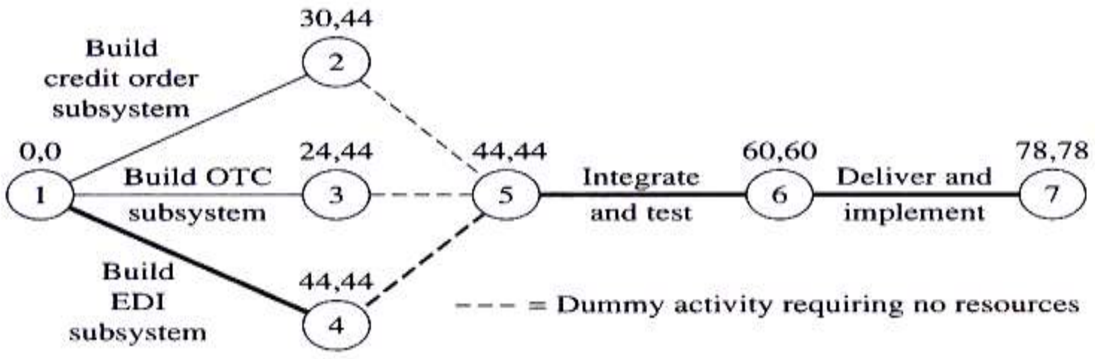
b. Several techniques(methods) are adopted to monitor progress against plan

1. Work Breakdown Structures (WBS)
2. PERT (Program Evaluation and Review Technique)
3. GANTT Chart



WBS for Order Entry System

PERT chart for Order entry system



<u>Modeling:</u>

It can be divided into two ways

1. Aalysis
2. Design

Analysis : System Analyst must understand

I. Information domain

II. Required functions

III. Behavior of information domain

IV. Performance

V. Requirements reviewed and documented

Design : Design is multi step process focused on

I. Data Dictionary (Tables, Attributes, Data types of attributes, Constraints on tables, Relationships between the tables)

II. User interfaces (Screens)

III. Algorithmetic details

IV. Data structures

V. DFD' (Data Flow Diagrams)

VI. UML (Uniform Modeling Language) Diagrams

VII. Flowcharts (System flowcharts, Program flowcharts, Control flow charts)

<u>Construction:</u>

Coding: The process of writing application programs to given specifications.

Testing: The process of verifying the application program for uncovered errors or irregularities.

<u>Deployment</u>:

Deployment : Installing product to customer

a. Delivery
b. Support
c. Feed back

a. **Delivery:** Dispatching product to customer for usage

b. **Support** *:* Rectifying problems of end user

- Software undergo for change after it is deliver to customer.
- Change will occur because
  a) Errors has been uncounted.
  b) To accommodate changes in its external environment.
  c) Customer requires functional or performance enhancement.

c. **Feedback**: Details given by end user.

Feedback from end users can be collected using

i) Interviews

ii) Questionnaires

iii) Onsite observations

Limitations:

1. Difficult to state all requirements explicitly.
2. Difficult to accommodate natural changes.
3. Customer must have patience.
4. If requirements are formulate poorly the entire system fails

Advantages of waterfall model

The waterfall model is simple and easy to understand, implement, and use.

- All the requirements are known at the beginning of the project, hence it is easy to manage.
- It avoids overlapping of phases because each phase is completed at once.
- This model works for small projects because the requirements are understood very well.
- This model is preferred for those projects where the quality is more important as compared to the cost of the project.

Disadvantages of the waterfall model

- This model is not good for complex and object oriented projects.
- It is a poor model for long projects.
- The problems with this model are uncovered, until the software testing.

The amount of risk is high

When to use the water fall model

This model is used only when the requirements are very well known, clear and fixed Technology is understood , the project is short, there are no ambiguous requirements

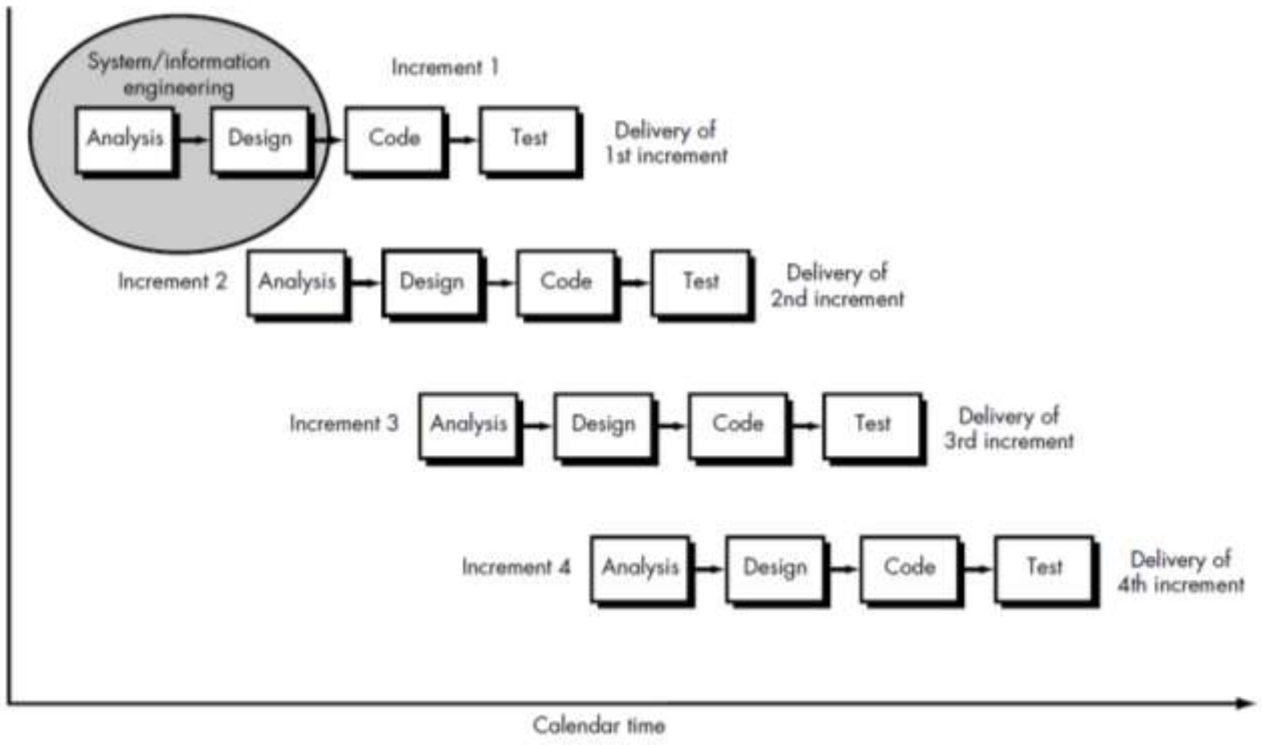
2.INCREMENTAL MODEL

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Objectives :

(a) Software development

(b) Guides software engineer through a set of frame work activities



Each increment is developed using water fall model

E.g. Entity: Bank

Increment 1: Automating teller operation

Increment 2: Automating ATM operation

Increment 3: Automating internet banking operation

Increment 4: Automating mobile banking operation

Each increment includes the following steps

Communication:

Project initiation : Feasibility of proposed system (system to be developed) is considered

Economical feasibility: Whether there is economy to develop the developed system.

Technical feasibility: Whether there is a technology resource to develop the proposed system.

Operational feasibility: Whether day to day operations are possible.

Behavioral feasibility: Is the solutions to behavioral problems arises between the people.

Requirement gathering:

Software engineer communicate with stack holder to understand the information domain and formulates design specifications.

Effective communication between the software engineer and stack holder is need to formulate better requirements.

Planning:

Estimating: Resources need for software development is estimated.

Several techniques are adopted to monitor progress against plan

1. Work Breakdown Structures (WBS)
2. PERT (Program Evaluation and Review Technique)
3. GANTT Chart

Modeling:

It can be divided in to two ways

a. **Analysis** and b.**Design**

Analysis : System Analyst must understand

- ➢ Information domain
- ➢ Required functions
- ➢ Behavior of information domain
- ➢ Performance
- ➢ Requirements reviewed and documented

Design : Design is multi step process focused on

- ➢ Data Dictionary (Tables, Attributes, Data types of attributes, Constraints on tables, Relationships between the tables)

- User interfaces (Screens)
- Algorithmetic details
- Data structures
- DFD$^{'s}$ (Data Flow Diagrams)
- UML (Uniform Modeling Language) Diagrams
- Flowcharts (System flowcharts, Program flowcharts, Control flow charts)

Construction:

Coding: The process of writing application programs to given specifications.

Testing: The process of verifying the application program for uncovered errors or irregularities.

Deployment: Installing product to customer

a. Delivery
b. Support
c. Feed back

a. **Delivery**: Dispatching product to customer for usage

b. **Support** *:* Rectifying problems of end user

- Software undergo for change after it is deliver to customer.
- Change will occur because
  - Errors have been uncounted.
  - To accommodate changes in its external environment.
  - Customer requires functional or performance enhancement.

c. **Feedback**: Details given by end user.Feedback from end users can be collected using

- Interviews
- Questionnaires
- Onsite observations

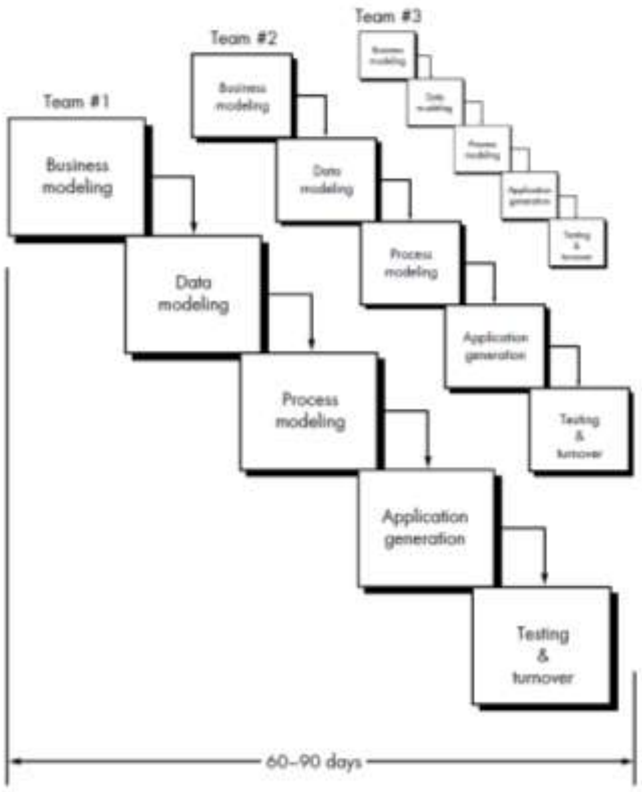
Advantages:

1. Useful when staffing is unavailable.

2. Each increment can be implemented with fewer people.
3. Core product can be developed & reviewed.
4. Increment can be planned to manage technical risks**.**

3 THE RAD MODEL (RAPID APPLICATION DEVELOPMENT)

- Objectives : (a) Software development in very short time period (b) Guides software engineer through a set of frame work activities



- High speed adoption of linear sequence model.
- RAD approach support the following phases.

1. Business modeling
2. Data modeling
3. Process modeling
4. Application generation
5. Testing & Turnover

<u>Business modeling:</u>

The information flow among business functions is modeled in a way that answers the following questions.

1. What information derives the business process?
2. What information is generated?
3. Who generates it?

Data modeling:

Database objects are defined.

a) Entities
b) Attributes
c) Key Attributes
d) Relations
e) Constraints

Process modeling:

Process description is created for (a) Retrieval of data (b) Deletion of data (c) Addition of data (d) Modification of data

Application Generation:

- ✓ Forth generation techniques are adopted for writing application
- ✓ Reuse the existing programs
- ✓ Create re useable components when necessary
- ✓ Automated tools are used for the construction of S/W

Testing and Turnover

- Many of the components have already been tested. This reduces overall testing time.
- The new components are tested and variety of test cases of exercised.

Limitations :

1. Sufficient human resources is required.
2. Software engineers are committed to rapid fire activities.
3. If commitment is lacking entire project will fail.
4. If a system cannot be modularized RAD will be problematic.
5. RAD will not be appropriated & technical risks are high.

ADVANTAGES

- An incremental software process model
- Having a short development cycle

- High-speed adoption of the waterfall model using a component based construction approach
- Creates a fully functional system within a very short span time of 60 to 90 days
- Multiple software teams work in parallel on different functions
- Modeling encompasses three major phases: Business modeling, Data modeling and process modeling
- Construction uses reusable components, automatic code generation and testing

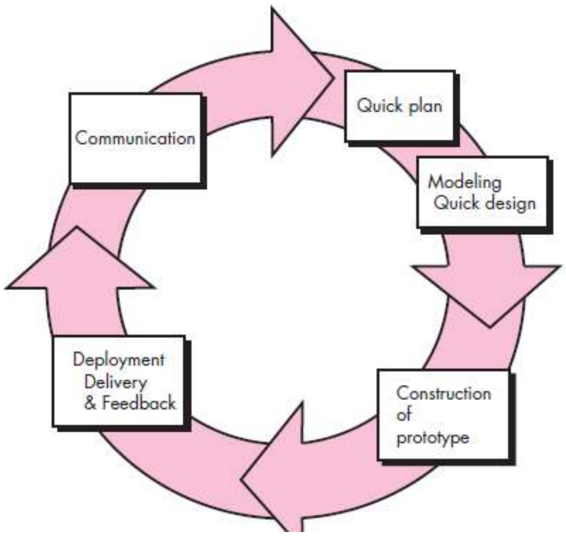
DIS-ADVANTAGES

- Not all types of application are appropriate for RAD
- Requires a number of RAD teams
- Requires commitment from both developer and customer for rapid-fire completion of activities otherwise it fails
- If system cannot be modularized properly project will fail.
- Not suited when technical risks are high

Evolutionary Process Models

1. PROTOTYPING

Objective: Software development when requirements are ***hazy*** (unclear).



The basic idea in **Prototype model** is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Prototype model is a software development model. By using this prototype, the client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system. Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements.

Advantages of Prototype model:

- Users are actively involved in the development
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier.
- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily

Disadvantages of Prototype model:

- Leads to implementing and then repairing way of building systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Incomplete application may cause application not to be used as the full system was designed Incomplete or inadequate problem analysis

When to use prototype model

1) It is used when the desired system needs to have a lot of interaction with the end users.
2) Typically online system, web interfaces have a very high amount of interaction with end users.
3) It is ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system.

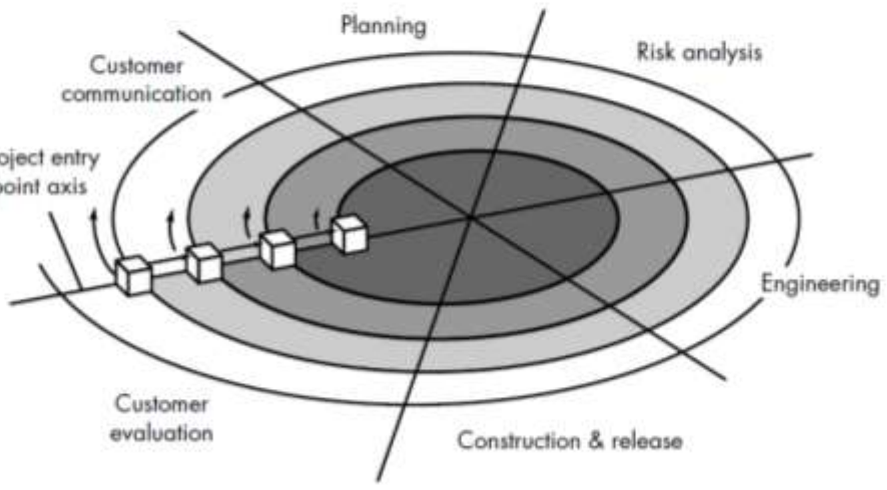
THE SPIRAL MODEL

Objective: To develop **effective** and **efficient** system.

Effectiveness: The working program must satisfy the needs of the user.

Efficient: The working program must use minimum resources to produce throughput.

Software is updated periodically according to the changing requirements of the customer.



The Spiral model include the following phases

- ✓ Customer communication
- ✓ Planning
- ✓ Risk analysis
- ✓ Engineering
- ✓ Construction & release
- ✓ Customer evaluation

<u>Customer communication:</u>

Software engineer communicate with stack holder to understand the information domain and formulates design specifications.

Effective communication between the software engineer and stack holder is need to formulate better requirements.

<u>Planning:</u>

Estimating: Resources need for software development is estimated.

Several techniques are adopted to monitor progress against plan

a. Work Breakdown Structures (WBS)

b. PERT (Program Evaluation and Review Technique)

c. GANTT Chart

<u>Risk analysis:</u>

Asses both *technical risks* and *management risks*.

<u>Engineering:</u>

Various software engineering techniques are adapted to develop the system

E.g. Top-down approach, Bottom-up approach, Physical design

<u>Construction & release:</u>

Coding: The process of writing application programs to given specifications.

Testing: The process of verifying the application program for uncovered errors or irregularities.

<u>Customer evaluation:</u>

Feedback*:* Details given by end user.

Feed back from end users can be collected using

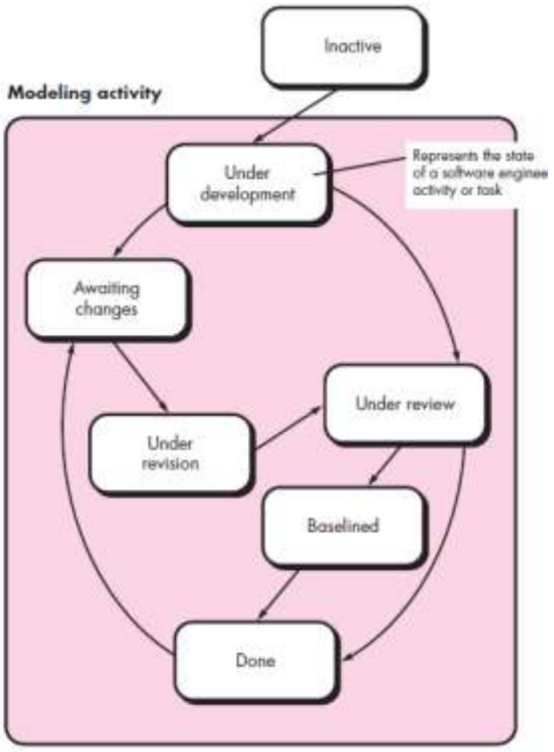
i) Interviews

ii) Questionnaires

iii) Onsite observations

CONCURRENT DEVELOPMENT MODEL (OR) CONCURRENT ENGINEERING

The *concurrent development model,* sometimes called *concurrent engineering,* allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter. For example, the modelling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design
For example, early in a project the communication activity (not shown in the figure)has completed its first iteration and exists in the **awaiting changes** state. The modelling activity (which existed in the **inactive** state while initial communication was completed,now makes a transition into the **under development** state. If, however, thecustomer indicates that changes in requirements must be made, the modeling activitymoves from the **under development** state into the **awaiting changes** state.Concurrent modeling defines a series of events that will trigger transitions fromstate to state for each of the software engineering activities, actions, or tasks.



Forexample, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements model isuncovered. This generates the event *analysis model correction*, which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.

SPECIALIZED PROCESS MODELS

1. THE FORMAL METHODS MODEL

- Formal method: Mathematical Specification
- Formal mathematical specifications enables a software engineer to specify develop & verify computer based system.
- This notation is also called as "clean room software engineering ".
- Formal methods eliminating many problems that are difficult to overcome.
- Ambiguity, incompleteness and inconsistency can be discovered and corrected more easily.
- Formal methods serve as basis for program verification.
- Formal methods offers promise of defect software
- Time consuming and expensive.

- Extensive training is required for s/w engineering
- Few software developers have the necessary background to apply formal methods.

2. COMPONENT BASE DEVELOPMENT

- Component :Working program.
- Components are developed by the vendors.
- Developed using spiral model.
- Evolutionary in nature

Component based developed model incorporate the following steps:

1. Developer must understands the information domain.
2. S/W architecture is designed.
3. Comprehensive testing is conducted.

3. ASPECT ORIENTED SOFTWARE DEVELOPMENT

- Aspect : Software component.
- Complex software is developed.
- Complex software includes (a) Localized features (b) Functions (c) Information content

Features:

1.User interfaces 2.Collaborative work

3.Distribution 4.Persistency

5.Memory management 6.Transaction processing 7.Security integrity

Types:

i) User interface aspects (Event based programming)

ii) Distribution aspects (Receiving and transporting)

iii) Persistency aspects (Storing, Retrieving and Indexing)

iv) Authentication aspect(Encoding and Decoding)

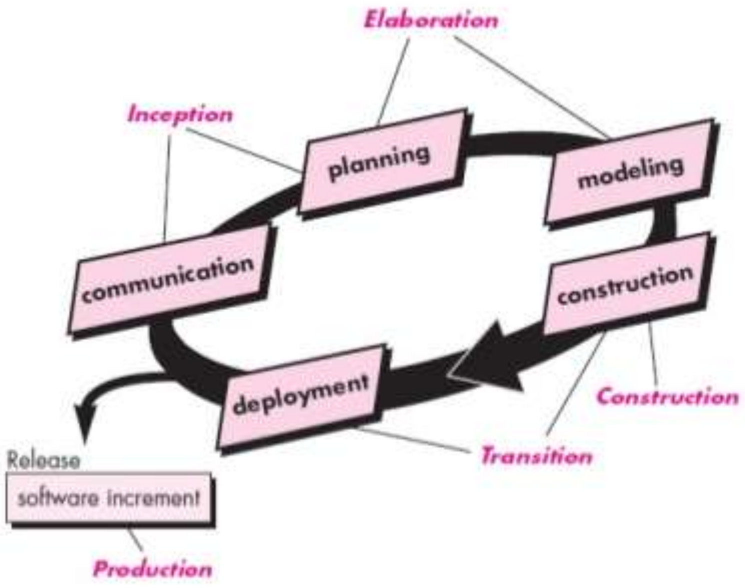
v) Transaction aspect (Atomicity, Concurrency control and Logging Strategy)

4. THE UNIFIED PROCESS

Unified process: Attempt to draw on the best features and characteristics of conventional software process.

Unified process has the following phases

- Inception
- Elaboration
- Construction
- Transaction



2.PROJECT MANAGEMENT

1.Write about process management

Effective software project management focuses on **the four P's: people, product, process, and project.**.

1) The People

The people management defines the following key practice areas for software people

- RECRUITING
- SELECTION
- PERFORMANCE MANAGEMENT
- TRAINING
- COMPENSATION
- CAREER DEVELOPMENT
- ORGANIZATION AND WORK DESIGN
- TEAM/CULTURE DEVELOPMENT.

❖ **The Players(Stack holders):**

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

1. **Senior managers:** who define the business issues that influence on the project.
2. **Project** (technical) **managers :**who must plan, motivate, organize, and control the practitioners who do software work.
3. **Practitioners** who deliver the technical skills that are helps to develop a product or application.
4. **Customers** who specify the requirements for the software to be engineered(developed)
5. **End-users** who interact with the software once it is released for Production use.

❖ **Team Leaders:**

✓ Project management is a people-intensive activity

✓ MOI model of leadership:

- ✓ **Motivation.** The ability to encourage (by “push or pull”) technical people to produce to their best ability.
- ✓ **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
- ✓ **Ideas or innovation**. The ability to encourage people for development of a particular software product or application

❖ **The Software Team:**

- ✓ Many human organizational structures for software developmentas there are organizations that develop software
- ✓ **Project factors that should be considered when planning the structure of software engineering teams:**
  - The difficulty of the problem to be solved.
  - The size of the resultant program(s) in lines of code or function points
  - The time that the team will stay together (team lifetime).
  - The degree to which the problem can be modularized.
  - The required quality and reliability of the system to be built
  - The rigidity of the delivery date.

❖ **Agile Team:**

An Agile team is a self organizing team that has anatomy to plan and make technical decisions.

❖ **Coordination and Communication Issues:**

- ✓ There are many reasons that software projects get into trouble.
- ✓ Interoperability has become a key characteristic of many systems.
- ✓ New software must communicate with existing software and conform to predefined constraints imposed by the system or product.
- ✓ To deal with them effectively, a software engineering team must establish effective methods for coordinating the people who do the work.
- ✓ Formal communicationis accomplished through “writing, structured meetings, and other relatively noninteractive and impersonal communication channels” .

- ✓ Electronic communication encompasses electronic mail, electronic bulletinboards, and by extension, video-based conferencing systems.

2)The Product: Before a project can be planned

- ✓ product objectives
- ✓ scope should be established,
- ✓ alternative solutions should be considered,
- ✓ technical and management constraintsshould be identified.

Without this information, it is impossible to estimates the cost.

Software Scope:

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

Problem Decomposition:

Problem decomposition, sometimes called partitioning or problem elaboration, is an activity that sits at the core of software requirements analysis .

- Decomposition is applied in two major areas:

  (1) The functionality that must be delivered and

  (2) The process that will be used to deliver it.

- Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation

3) The Process

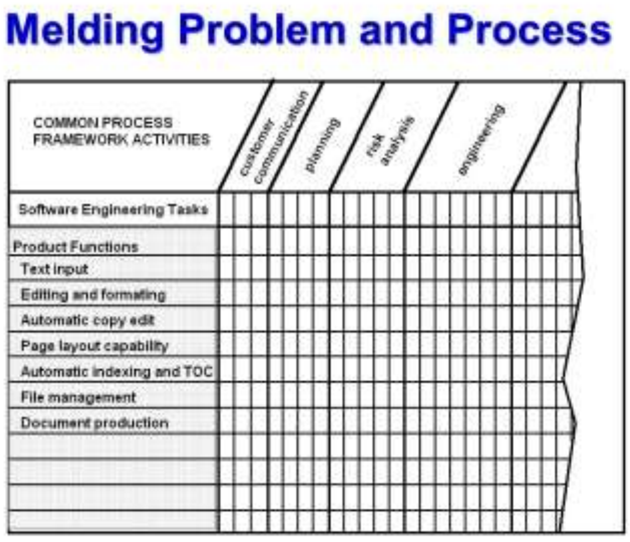
- ✓ A software process provides the framework from which a comprehensive plan for software development can be established..

- ✓ umbrella activities—such as software quality assurance, software configuration management,and measurement—overlay the process model
- ✓ A wide array of software engineering paradigms include:
  - • The linear sequential model
  - • The prototyping model
  - • The RAD model
  - • The incremental model
  - • The spiral model
- ✓ The project manager must decide which process model is most appropriate for

  (1)The customers who have requested the product and the people who will do the work.

  (2) The characteristics of the product itself.

  (3) The project environment in which the software team works.



Melding Problem and Process

COMMON PROCESS FRAMEWORK ACTIVITIES	customer communication	planning	risk analysis	engineering	
Software Engineering Tasks					
Product Functions					
Text input					
Editing and formatting					
Automatic copy edit					
Page layout capability					
Automatic indexing and TOC					
File management					
Document production					

Process Decomposition

- Once the process model has been chosen, the common process framework (CPF) is adapted to it.
- “How do we accomplish this CPF activity?”

- For example, a small,relatively simple project might require the following work tasks for the customer communication activity:

  1. Develop list of clarification issues.
  2. Meet with customer to address clarification issues.
  3. Jointly develop a statement of scope.
  4. Review the statement of scope with all concerned.
  5. Modify the statement of scope as required.

4)The Project

- ✓ In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs that lead to good project management.
- ✓ In order to manage a successful software project, we must understand what can go wrong and how to do it right.
- ✓ In a software projects one defines ten signs that indicate that an information systems project is in jeopardy:

1. Software people don’t understand their customer’s needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
4. The chosen technology changes.
5. Business needs change [or are ill-defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

Write about the W5HH PRINCIPLES

- Barry Boehm [BOE96] states:
- “you need an organizing principle that scales down to provide simple [project] plans for simple projects.”
- He calls it the WWWWWHH principle, after a series of questions that lead to a definition of key project characteristics and the resultant project plan:
- **Why is the system being developed?**
  The answer to this question enables all parties to assess the validity of
  business reasons for the software work.
- **What will be done, by when?**

  The answers to these questions help the team to establish a project schedule by identifying key project tasks that are required by the customer.

- **Who is responsible for a function?**

  The role and responsibility of each member of the software team must be defined.

- **Where are they organizationally located?**

  Not all roles and responsibilities reside within the software team itself. The customer, users, and other stakeholders also have responsibilities.

- **How will the job be done technically and managerially?**

  Once productscope is established, a management and technical strategy for the project mustbe defined.

- **How much of each resource is needed?**

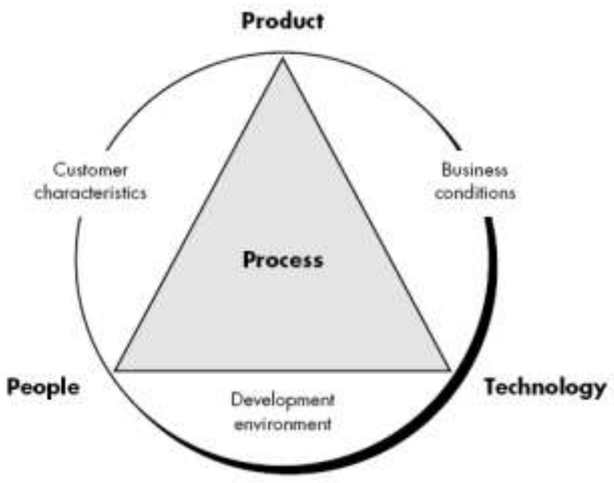
  The answer to this question is derived by developing estimates based on answers to earlier questions

PROCESS AND PROJECT METRICS

1 Write about Process and Project Metrics

❖ **Metrics**

- Software process and project metrics are quantitative measures
- They are a management tool
- They offer the effectiveness of the software process and the projects that are conducted using the process as a framework.
- Basic quality and productivity data are collected
- These data are analyzed, compared against past averages, and assessed
- The goal is to determine whether quality and productivity improvements have occurred
- The data can also be used to pinpoint problem areas
- Remedies can then be developed and the software process can be improved



❖ **Metrics in the Process Domain**

- Process metrics are collected across all projects and over long periods of time.
- They are used for making <u>strategic</u> decisions.
- The only way to know how/where to improve any process is to

  - Measure specific <u>attributes</u> of the process.
  - Develop a set of meaningful <u>metrics</u> based on these attributes.
  - Use the metrics to provide <u>indicators</u> that will lead to a strategy for improvement.

- We measure the effectiveness of a process by deriving a set of metrics based on <u>outcomes</u> of the process such as
  - Errors uncovered before release of the software
  - Defects delivered to and reported by the end users
  - Work products delivered
  - Human effort expended
  - Calendar time expended
  - Conformance to the schedule
  - Time and effort to complete each generic activity

Etiquette of Process Metrics

- Use common sense and organizational sensitivity when interpreting metrics data
- Provide regular feedback to the individuals and teams who collect measures and metrics
- Don’t use metrics to evaluate individuals
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them
- Never use metrics to threaten individuals or teams

❖ **Metrics in the Project Domain**

- Project metrics enable a software project manager to
  - Assess the status of an ongoing project
  - Track potential risks
  - Uncover problem areas before their status becomes critical
  - Adjust work flow or tasks
  - Evaluate the project team’s ability to control quality of software work products

- Many of the same metrics are used in both the process and project domain
- Project metrics are used for making tactical decisions
  - They are used to adapt project workflow and technical activities

Use of Project Metrics

- The first application of project metrics occurs during estimation
  - Metrics from past projects are used as a basis for estimating time and effort
  - As a project proceeds, the amount of time and effort expended are compared to original estimates
- Project metrics are used to
  - Minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
  - Assess product quality on an ongoing basis and, when necessary, to modify the technical approach to improve quality

2 .What are the Categories of Software Measurement

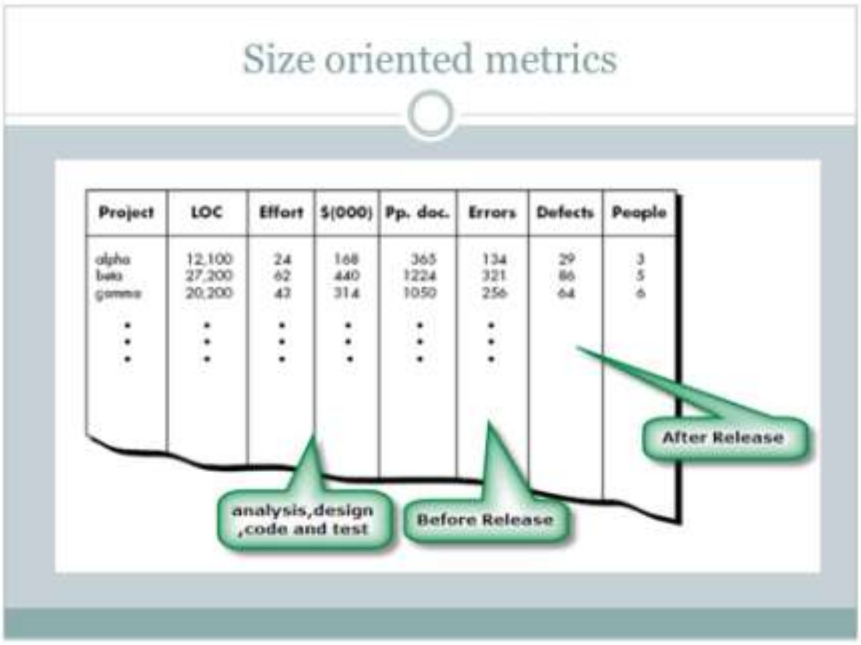
There are the two Categories of Software Measurement

- **Direct measures of the**
  - Software process (cost, effort, etc.)
  - Software product (lines of code produced, execution speed, defects reported over time, etc.)
- **Indirect measures of the**
  - Software product (functionality, quality, complexity, efficiency, reliability, maintainability, etc.)

Project metrics can be consolidated to create process metrics for an organization by using

❖ **Size-oriented Metrics**

- Derived by normalizing quality and/or productivity measures by considering the size of the software produced
- Thousand lines of code (KLOC) are often chosen as the normalization value



Size oriented metrics

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
⋮	⋮	⋮	⋮	⋮	⋮		

analysis,design ,code and test
Before Release
After Release

- Metrics include
  - Errors per KLOC - Errors per person-month
  - Defects per KLOC - KLOC per person-month
  - Dollars per KLOC - Dollars per page of documentation
  - Pages of documentation per KLOC
- Size-oriented metrics are not universally accepted as the best way to measure the software process
- Opponents argue that KLOC measurements
  - Are dependent on the programming language
  - Penalize well-designed but short programs
  - Cannot easily accommodate nonprocedural languages
  - Require a level of detail that may be difficult to achieve

❖ **Function-oriented Metrics**

- Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value
- Most widely used metric of this type is the function point:
  FP = count total * [0.65 + 0.01 * sum (value adj. factors)]

- Function point values on past projects can be used to compute, for example, the average number of lines of code per function point (e.g., 60)
- LOC Per Function Point

Language	**Average**	**Median**	**Low**	**High**
Ada	154	--	104	205
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
COBOL	77	77	14	400
Java	55	53	9	214
PL/1	78	67	22	263
Visual Basic	47	42	16	158

Web Engineering Project Metrics :

Measures that can be collected are

- **Number of static Web** pages the end-user has no control over the content displayed on the page
- **Number of dynamic Web** pages end-user actions result in customized content displayed on the page
- **Number of internal page links** (internal page links are pointers that provide a hyperlink to some other Web page within the WebApp
- **Number of persistent data** objects As the number of persistent data objects grows the complexity of webapps also grows
- **Number of external systems interfaced** As the requirement for interfacing grows, system complexity and development effort also increases

- **Number of static content objects** They encompass graphical,audio,video information incorporated into the webapp
- **Number of dynamic content objects** Generated based on end-user actions
- **Number of executable functions** An executable function provides some computational service to end-user. As number of functions grows construction effort increasesWe can define a metric that reflects degree of end-user customization for webapp to effort expended on web-project

  Nsp=number of static pages

  Ndp=number of dynamic pages

  Customization index C=Ndp/(Ndp+Nsp)

Object-oriented Metrics

- The set of metrics for oo projects are

- **Number of scenario scripts (i.e., use cases)**
  - This number is directly related to the size of an application and to the number of test cases required to test the system
- **Number of <u>key</u> classes (the highly independent components)**
  - Key classes are defined early in object-oriented analysis and are central to the problem domain
  - This number indicates the amount of effort required to develop the software
- **Number of <u>support</u> classes**
  - This number indicates the amount of effort and potential reuse
- **Average number of support classes per key class**
  - Estimation of the number of support classes can be made from the number of key classes
  - GUI applications have between <u>two and three times</u> more support classes as key classes
  - Non-GUI applications have between <u>one and two times</u> more support classes as key classes
- **Number of subsystems**

- A subsystem is an aggregation of classes that support a function that is visible to the end user of a system

3.Metrics for Software Quality Correctness

These are the measures of software quality

Correctness. A program must operate correctly or it provides little value to its users. Correctness is the degree to which the software performs its required function.
Maintainability. Software maintenance and support accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements.
A simple time-oriented metric is ***mean-time-to-change*** **(MTTC),** the time it takes to analyze the change request
Integrity. Software integrity has become increasingly important in the age of cyber terrorists and hackers. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security.
To measure integrity, two additional attributes must be defined: threat and security.
Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled.

The integrity of a system can then be defined as:

$$\text{Integrity} = \Sigma\,[1 - (\text{threat} \times (1 - \text{security}))]$$

Defect Removal Efficiency

- Defect removal efficiency provides benefits at both the project and process level.
- It indicates the percentage of software errors found before software release
- It is defined as **DRE = E / (E + D)**
  - is the number of errors found before delivery of the software to the end user
  - is the number of defects found after delivery

4 .Integrating Metrics within the Software Process

To be an effective aid in process improvement and/or cost and effort estimation, baseline data must have the **following attributes:**

(1) Data must be reasonably accurate—"guestimates" about past projects are to be avoided,

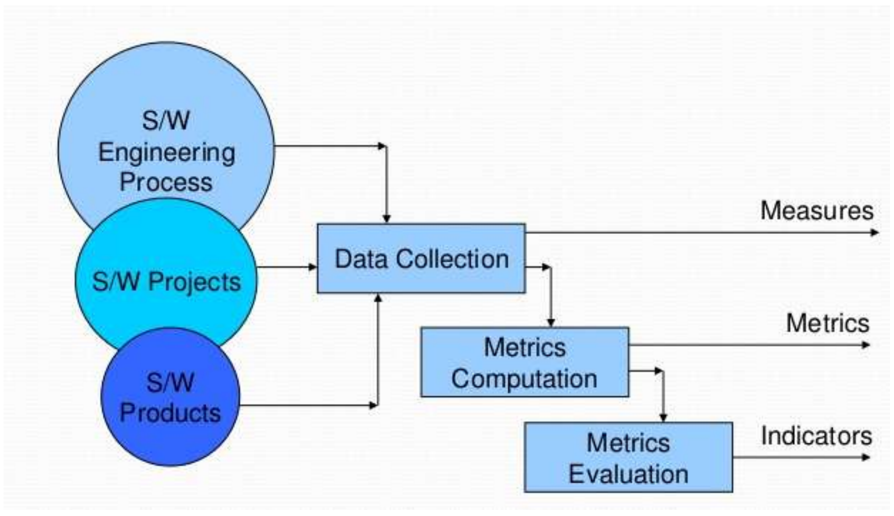
(2) Data should be collected for as many projects as possible

(3) Measures must be consistent (for example,a line of code must be interpreted consistently across all projects for which data are collected)

(4) Applications should be similar to work that is to be estimated—it makes little sense to use a baseline for batch information systems work to estimate a real-time, embedded application.

Metrics Collection, Computation, and Evaluation:

The process for establishing a metrics baseline is illustrated in below diagram. Ideally, data needed to establish a baseline have been collected in an ongoing manner.



SOFTWARE ESTIMATION AND ESTIMATION MODELS

Write about software estimation

- Estimating is important activity need not be conducted in a haphazard manner.
- Estimation risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule.
- If project scope is poorly understood or project requirements are subject to change, uncertainty and risk become dangerously high.
- project manager should not become obsessive about estimation.

❖ **The Project Planning Process**

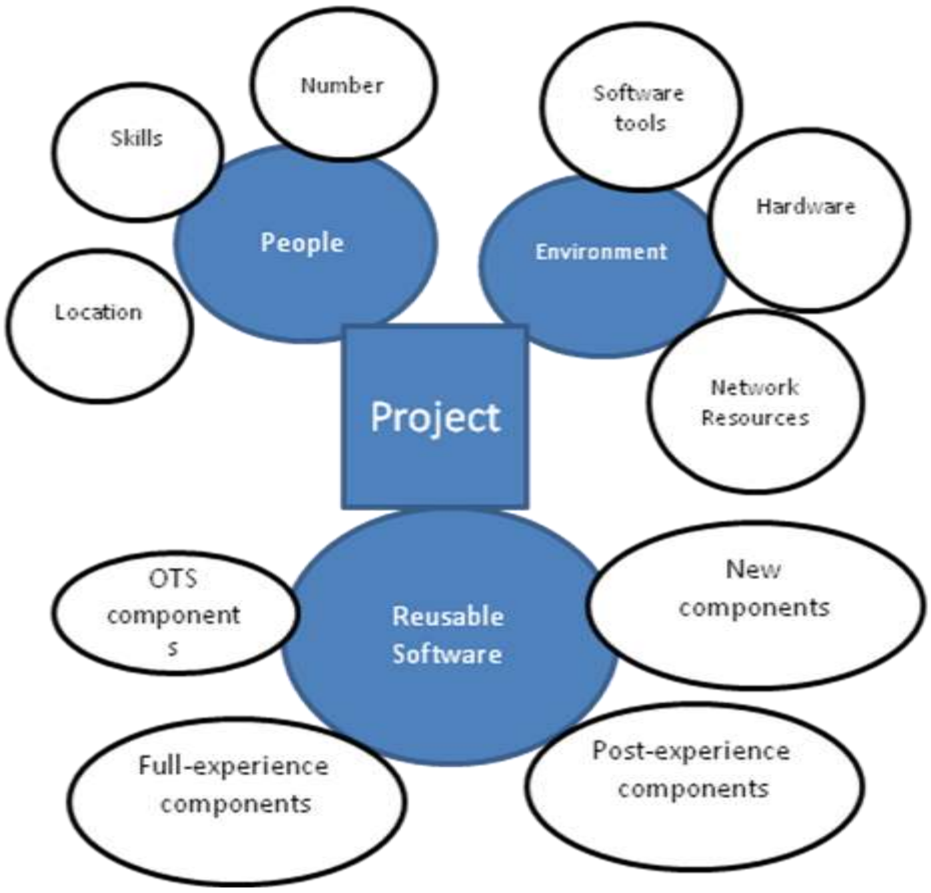
- The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
- These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses.

Software scope and feasibility

- Software scope describes the functions and features that are to be delivered to end users.
- Scope is defined by using one of the two techniques:
  1. A narrative description of software scope is developed after communication with all stakeholders.
  2. A set of use-cases is developed by end-users
- Functions described in the statement of scope are evaluated prior to the beginning of estimation.
- Because both cost and schedule estimates are functionally oriented,

Resources

- The second software planning task is estimation of the resources required to accomplish the software development effort. Figure



1. Human Resources

- The planner begins by evaluating scope and selecting the skills required to complete development.
- Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, and client/server) are specified.
- The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made.

2 .Reusable Software Resources

- Component-based software engineering (CBSE)5 emphasizes reusability—that is, the creation and reuse of software building blocks.
- **Off-the-shelf components.** Existing software that can be acquired from a third party or that has been developed internally for a past project.
- **Full-experience components**. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project.

- **Partial-experience components**. . Software components that must be built by the soft-ware team specifically for the needs of the current project.

3.Environmental Resources

- The environment that supports the software project, often called the software engineering environment (SEE), incorporates hardware and software.
- Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice

Software Project Estimation

To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve100% accurate estimates after the project is complete!).

2. Base estimates on similar projects that have already been completed.

3. Use relatively simple decomposition techniques to generate project cost and

Effort estimates.

4. Use one or more empirical models for software cost and effort estimation.

DECOMPOSITION TECHNIQUES

- Software project estimation is a form of problem solving.
- We decompose the problem, from two different points of view:
    - Decomposition of the problem
    - Decomposition of the process.
- Estimation uses one or both forms of partitioning.
- But before an estimate can be made, the project planner must understand the scope of the software to be built and generate an estimate of its "size."

Software Sizing

- The accuracy of a software project estimate is predicated on a number of things:

(1) The degree to which the planner has properly estimated the size of the product to be built;

(2) The ability to translate the size estimate into human effort, calendar time,

✷ **Problem-Based Estimation**

- LOC and FP data are used in two ways during software project estimation:

  (1) As an estimation variable to "size" each element of the software

  (2) As baseline metrics collected from past projects with estimation variables to develop cost and effort projections.

- LOC and FP estimation are distinct estimation techniques
- In general, LOC/pm or FP/pm averages should be computed by project domain.
- That is, projects should be grouped by team size, application area, complexity, and other relevant parameters.

An Example of LOC-Based Estimation

- As an example of LOC and FP problem-based estimation techniques,
- let us consider a software package to be developed for a computer-aided design application for mechanical components.
- Using the *System Specification a*s a guide, a preliminary statement of software scope can be developed:

For our purposes, we assume that further refinement has occurred and that the following major software functions are identified:

• User interface and control facilities (UICF)

• Two-dimensional geometric analysis (2DGA)

• Three-dimensional geometric analysis (3DGA)

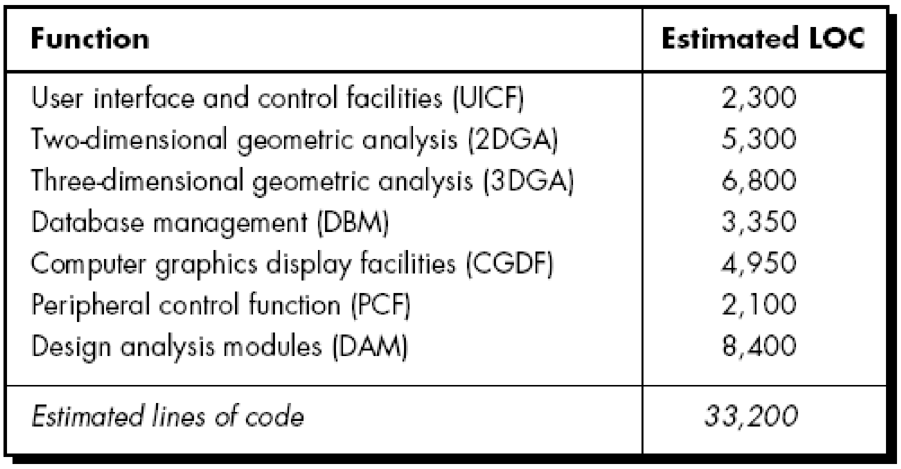
• Database management (DBM)

• Computer graphics display facilities (CGDF)

• Peripheral control function (PCF)

• Design analysis modules (DAM)

Following the decomposition technique for LOC, an estimation table, shown in Figure 5.3, is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic 4600 LOC, most likely—6900 LOC, and pessimistic—8600 LOC.

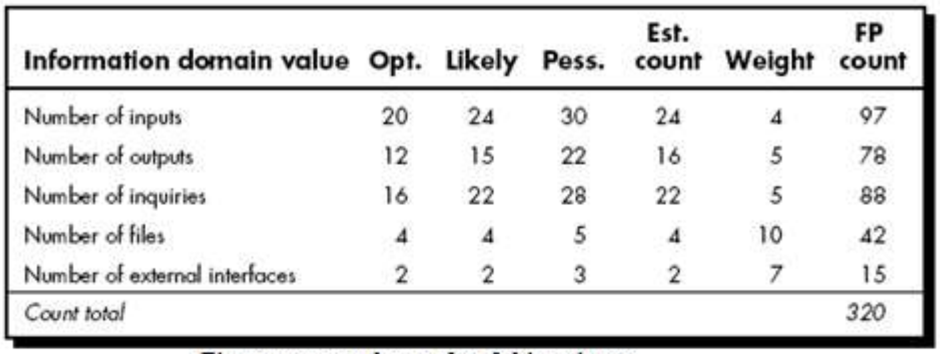


Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

An Example of FP-Based Estimation

- Decomposition for FP-based estimation focuses on information domain values rather than software functions.
- Referring to the function point calculation table presented in Figure
- The project planner estimates inputs, outputs, inquiries, files, and external interfaces for the CAD software.

- For the purposes of this estimate, the complexity weighting factor is assumed to be average.

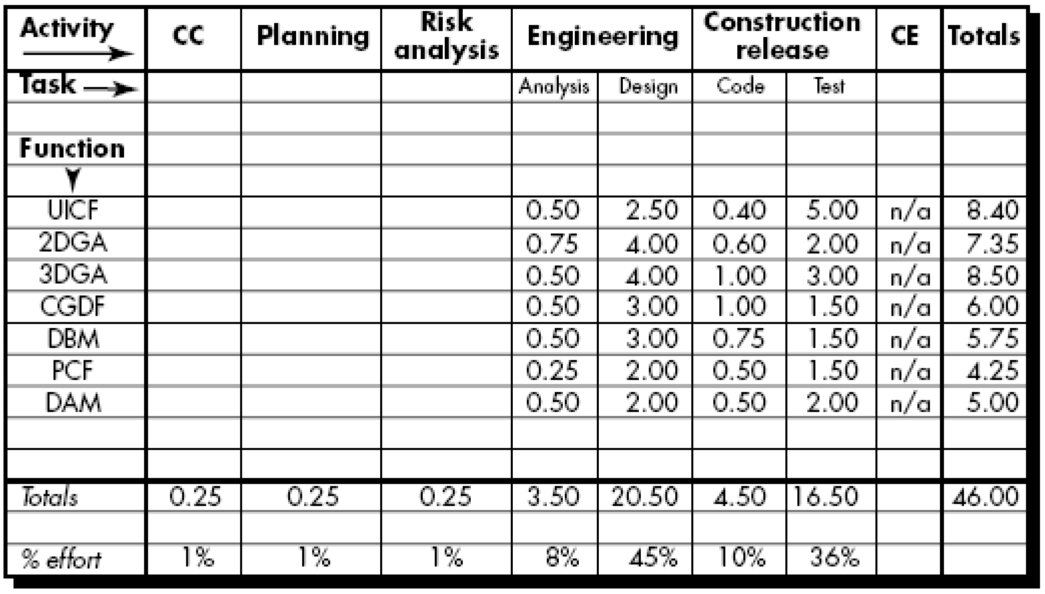


Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of inputs	20	24	30	24	4	97
Number of outputs	12	15	22	16	5	78
Number of inquiries	16	22	28	22	5	88
Number of files	4	4	5	4	10	42
Number of external interfaces	2	2	3	2	7	15
Count total						*320*

Figure presents the results of this estimate

✷Process-Based Estimation

- The process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated.
- Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope.
- A series of software process activities must be performed for each function.
- Functions and related software process activities may be represented as part of a table similar to the one presented in Figure.



Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function ↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

Explain EMPIRICAL ESTIMATION MODELS

- An *estimation model* for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.

The 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 models takes the form .

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

where *A*, *B*, and *C* are empirically derived constants,

E is effort in person-months,

e_v is the estimation variable (either LOC or FP).

Among the many LOC-oriented estimation models proposed in the literature are

$E = 5.2 \times (\text{KLOC})^{0.91}$	Walston-Felix model
$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$	Bailey-Basili model
$E = 3.2 \times (\text{KLOC})^{1.05}$	Boehm simple model
$E = 5.288 \times (\text{KLOC})^{1.047}$	Doty model for KLOC > 9

FP-oriented models have also been proposed. These include

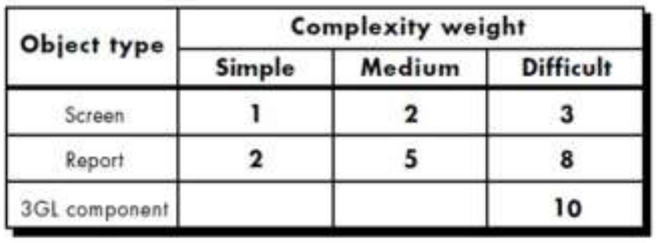
$E = -91.4 + 0.355\ \text{FP}$	Albrecht and Gaffney model
$E = -37 + 0.96\ \text{FP}$	Kemerer model
$E = -12.88 + 0.405\ \text{FP}$	Small project regression model

The COCOMO II Model (COnstructive **CO**st **MO**del**)**

Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.

Post-architecture-stage model. Used during the construction of the software.



Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

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

where NOP is defined as new object points.

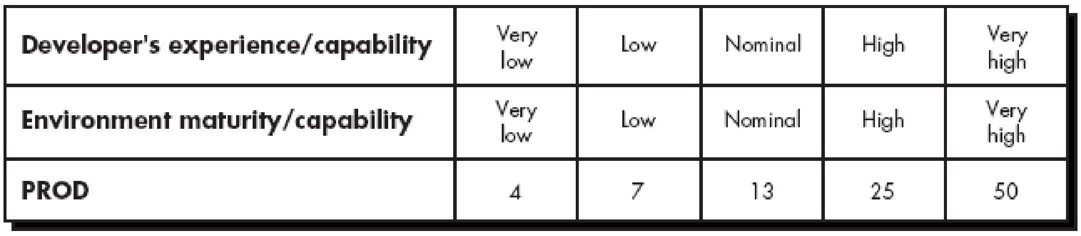
To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived. Figure 26.7 presents the productivity rate

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

for different levels of developer experience and development environment maturity.

Once the productivity rate has been determined, an estimate of project effort is computed using

$$\text{Estimated effort} = \frac{\text{NOP}}{\text{PROD}}$$



Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

The Software Equation

The software equation [PUT92] is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project.

$$E = \frac{\text{LOC} \times B^{0.333}}{P^3} \times \frac{1}{t^4}$$

where E = effort in person-months or person-years

t = project duration in months or years

B = “special skills factor”16

P = “productivity parameter” that reflects:

- Overall process maturity and management practices
- The extent to which good software engineering practices are used
- The level of programming languages used
- The state of the software environment
- The skills and experience of the software team

RISK MANAGEMENT

Definition of Risk

- A risk is a potential problem – it might happen and it might not

(or)

- Risk concerns future happenings
- Risk involves change in mind, opinion, actions, places, etc.
- Risk involves choice and the uncertainty that choice entails

Two characteristics of risk

- Uncertainty – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)
- Loss – the risk becomes a reality and unwanted consequences or losses occur

❖ **Types of Risk (Risk Categorization)**

- **Project risks**
  - They threaten the project plan
  - If they become real, it is likely that the project schedule will slip and that costs will increase

Technical risks

- They threaten the quality and timeliness of the software to be produced
- If they become real, implementation may become difficult or impossible

Business risks

- They threaten the viability of the software to be builtIf they become real, they jeopardize the project or the product

Sub-categories of Business risks

- **Market risk** – building an excellent product or system that no one really wants
- **Strategic risk** – building a product that no longer fits into the overall business strategy for the company
- **Sales risk** – building a product that the sales force doesn't understand how to sell
- **Management risk** – losing the support of senior management due to a change in focus or a change in people

- **Budget risk** – losing budgetary or personnel commitment

- **Known risks**
  - Those risks that can be uncovered after careful evaluation of the project plan.

  Predictable risks
  - Those risks that are extrapolated from past project experience (e.g., past turnover)

  Unpredictable risks
  - Those risks that can and do occur, but are extremely difficult to identify in advance.

❖ **RISK STRATAGIES**

✓ **Reactive risk strategies**
  - "Don't worry, I'll think of something"
  - The majority of software teams and managers rely on this approach
  - Nothing is done about risks until something goes wrong
  - Crisis management is the choice of management techniques

✓ **Proactive risk strategies**
  - Primary objective is to avoid risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner

❖ **Steps for Risk Management**

1. Identify possible risks; recognize what can go wrong
2. Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
3. Rank the risks by probability and impact
   - Impact may be negligible, marginal, critical, and catastrophic
4. Develop a contingency plan to manage those risks having high probability and high impact

❖ **Risk Identification:**

✓ Risk identification is a systematic attempt to specify threats to the project plan

✓ By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary

- Generic risks
  - Risks that are a potential threat to every software project

  Product-specific risks
  - Risks that can be identified only by those a with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
  - This requires examination of the project plan and the statement of scope
  - "What special characteristics of this product may threaten our project plan?"

❖ **Risk Projection/Estimation Steps :**

1. Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
2. Delineate the consequences of the risk
3. Estimate the impact of the risk on the project and product

4 .Note the overall accuracy of the risk projection so that there will be no misunderstandings

Risk Summary	Risk Category	Probability	Impact (1-4)	RMMM

Contents of a Risk Table

An effective strategy for dealing with risk must consider three issues

Risk mitigation (i.e., avoidance)   Risk monitoring

Risk management and contingency planning

Risk Mitigation Monitoring and Management

Its goal is to assist project team in developing a strategy for dealing with risk

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for risk mitigation. For example, assume that high staff turnover is noted as a project risk, r1. Based on past history and management intuition, the likelihood, l1, of high turnover is estimated to be 0.70 percent, and the impact, x1, is projected at level 2.

To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are

- Meet with current staff to determine causes for turnover
- Mitigate those causes that are under your control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work
- Assign a backup staff member for every critical technologist.

As the project proceeds, **risk monitoring activities** commence

In the case of high staff turnover, the following factors can be monitored:

- General attitude of team members based on project pressures.
- The degree to which the team has jelled.
- Interpersonal relationships among team members.
- Potential problems with compensation and benefits.
- The availability of jobs within the company and outside it.

In addition to monitoring these factors, the project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well underway and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team.

In addition, the project manager may temporarily refocus resources to those functions that are fully staffed, enabling newcomers who must be added to the team to “get up to speed.” Those individuals who are leaving are asked to stop all work and spend their last weeks in “knowledge transfer mode.”

It is important to note that RMMM steps incur additional project cost. For example, spending the time to "backup" every critical technologist costs money.

RMMM PLAN

1)Risk Avoidance(mitigation) Proactive planning for risk avoidance. This is achieved by developing a plan for risk mitigation.

2)Risk Monitoring what factors can we track that will enable us to determine if the risk is becoming more or less likely?

- Assessing whether predicted risk occur or not
- Ensuring risk aversion steps are being properly applied
- Collection of information for future risk analysis
- Determine which risks caused which problems

PROJECT SCHEDULING

What is project scheduling? Explain different techniques for project scheduling?

Project Scheduling

- Project scheduling is concerned with the techniques that can be employed to manage the activities that need to be undertaken during the development of a project.

Basic Principles:
Like all other areas of software engineering, a number of basic principles guide software project scheduling:

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both theproduct and the process are refined.

Interdependency. The interdependency of each compartmentalized activity ortask must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

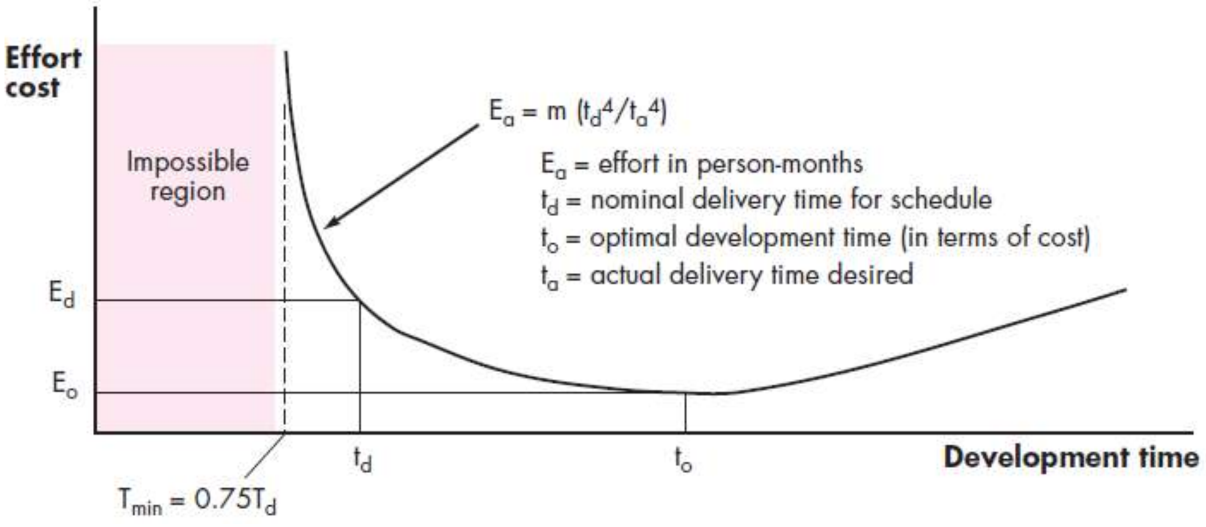
Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time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

Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time.

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

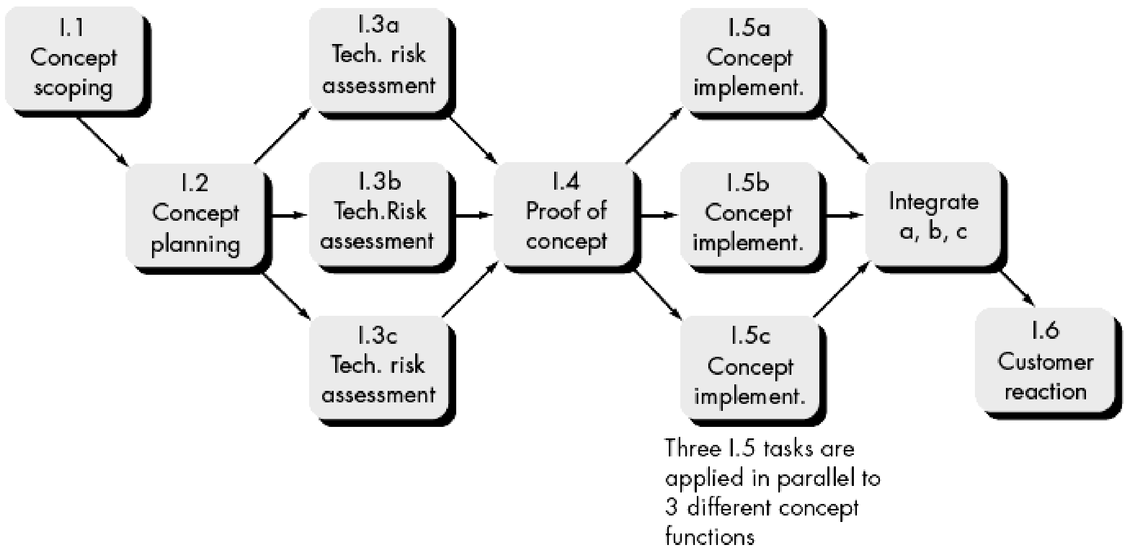
Defined outcomes. Every task that is scheduled should have a defined outcome.
For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone.



DEFINING A TASK NETWORK:

A *task network,* also called an *activity network,* is a graphic representation of the task flow for a project.



Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort.

Program evaluation and review technique (PERT) and the **critical path method** (CPM) are two project scheduling methods that can be applied to software development. Interdependencies among tasks may be defined using a task network. Tasks,sometimes called the project **work breakdown structure** (WBS), are defined for the product as a whole or for individual functions.

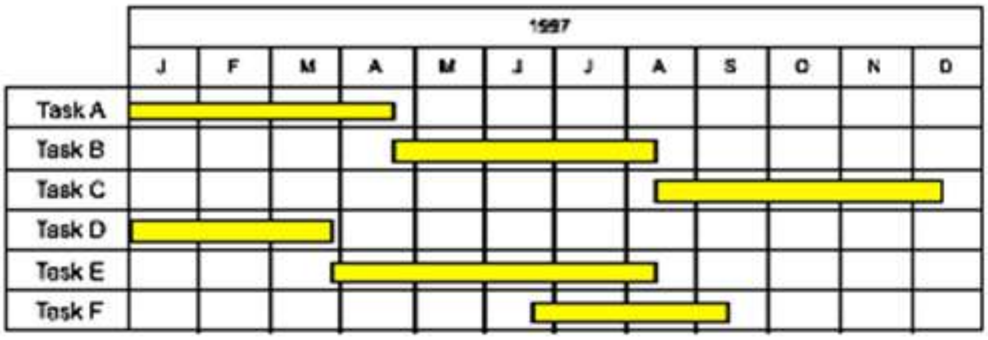
Both PERT and CPM provide quantitative tools that allow you to

(1) Determine the critical path—the chain of tasks that determines the duration of the project

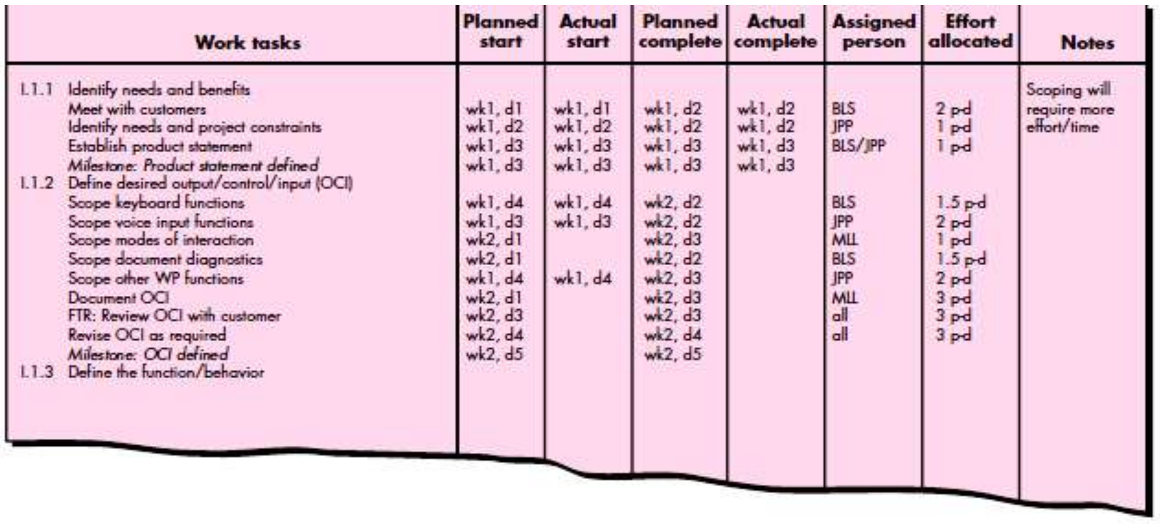
(2) Establish “most likely” time estimates for individual tasks by applying statistical models

(3) Calculate “boundary times” that define a time “window” for a particular task

Time-Line Charts (or) Gantt charts
The use of Gantt charts started during the industrial rev-olution of the late 1800's. An early industrial engineer named Henry Gantt developed these charts to improve factory efficiency. Gantt charts are developed using bars to represent each task. The length of the bar shows how long the task is expected to take to complete. Duration is easily shown on Gantt charts.



Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce *project tables*—a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information



Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

CPM models the activities and events of a project as a network. Activities are shown as nodes on the network and events that signify the beginning or ending of activities are shown as arcs or lines between the nodes

Steps in CPM Project Planning

1.Specify the individual activities.

2.Determine the sequence of those activities.

3.Draw a network diagram.

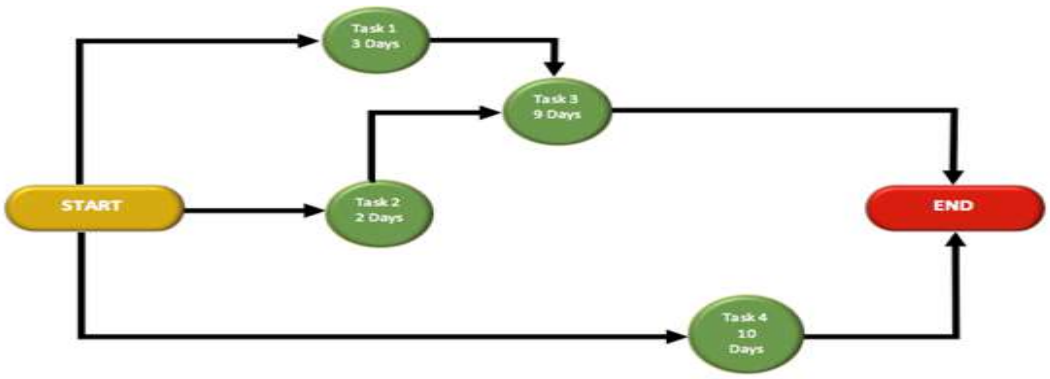
4.Estimate the completion time for each activity.

5.Identify the critical path (longest path through the network)

6.Update the CPM diagram as the project progresses

PERT

The Program Evaluation and Review Technique.(PERT) is a network model that allows for randomness in activity completion times.PERT is typically represented as an activity on arc network, in which the activities are rep-resented on the lines and milestones on the nodes.



Steps in the PERT Planning Process

PERT planning involves the following steps:

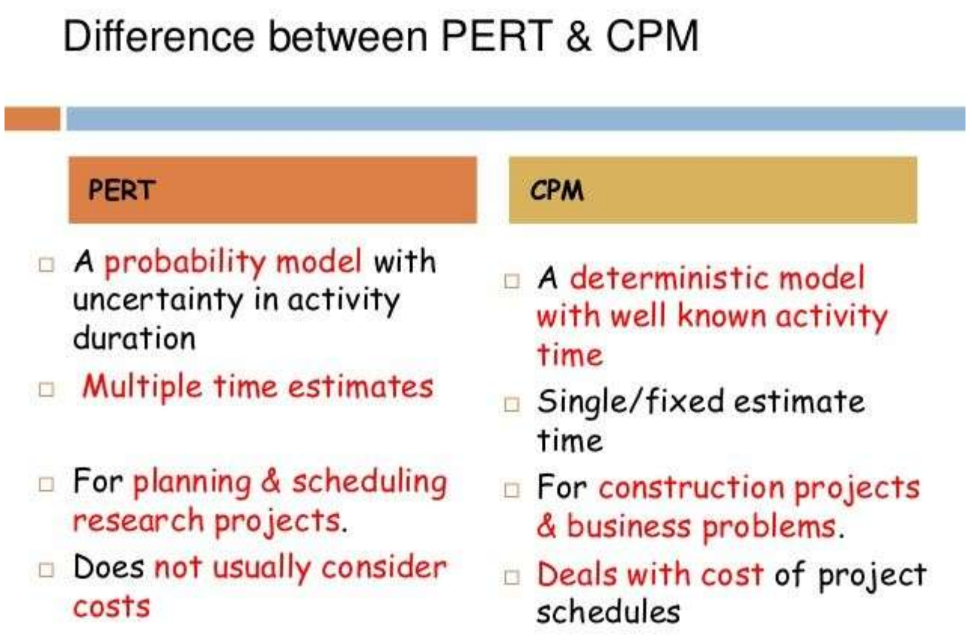
1.Identify the specific activities and milestones.

2.Determine the proper sequence of the activities.

3.Construct a network diagram.

4.Estimate the time required for each activity.

5.Determine the critical path.6.Update the PERT chart as the project progresses.



Difference between PERT & CPM

PERT	CPM
A probability model with uncertainty in activity duration	A deterministic model with well known activity time
Multiple time estimates	Single/fixed estimate time
For planning & scheduling research projects.	For construction projects & business problems.
Does not usually consider costs	Deals with cost of project schedules

UNIT - II

REQUIREMENTS ANALYSIS

UNIT - II

REQUIREMENTS ANALYSIS

- Requirement Engineering Processes
- Feasibility Study
- Problems of Requirements
- Software Requirement Analysis
- Analysis Concepts and Principles
- Analysis Process
- Analysis Model

REQUIREMENT ANALYSIS

- The software requirements are description of features and functionalities of the target system.
- Requirements convey(tells) the expectations of users from the software product.
- The requirements can be obvious or hidden, known or unknown, expected or unexpected from client's point of view.

Requirement Engineering

- The process to gather the software requirements from client, analyze and document them is known as requirement engineering.
- The goal of requirement engineering is to develop and maintain descriptive 'System Requirements Specification' document.

Requirement Engineering Process

It is a four step process, which includes –

- Feasibility Study
- Requirement Gathering
- Software Requirement Specification
- Software Requirement Validation

Feasibility study

- When the client approaches the organization for getting the desired product developed, it comes up with rough idea about what all functions the software must perform and which all features are expected from the software.
- Based on this the analysts does a detailed study about whether the desired system and its functionality are feasible to develop.
- This feasibility study is focused towards goal of the organization.
- The output of this phase should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

Requirement Gathering

- If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user.

- Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.

Software Requirement Specification

- SRS is a document created by system analyst after the requirements are collected from various stakeholders.
- SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, Security, Quality, Limitations etc.
- The requirements received from client are written in natural language.
- It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended useful by the software development team.

SRS should come up with following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.
- Conditional and mathematical notations for DFDs etc.

Software Requirement Validation

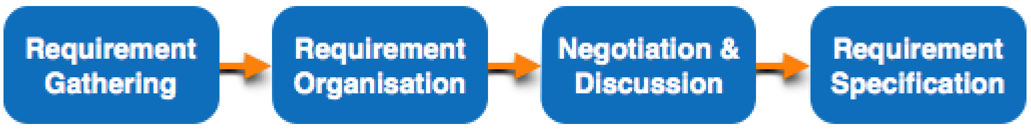
- After requirement specifications are developed, the requirements mentioned in this document are validated.
- User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly.
- Requirements can be checked against following conditions -

- If they can be practically implemented

- If they are valid and as per functionality and domain of software
- If there are any ambiguities
- If they are complete
- If they can be demonstrated

Requirement Elicitation Process

Requirement elicitation process can be depicted using the folloiwng diagram:



- **Requirements gathering -** The developers discuss with the client and end users and know their expectations from the software.
- **Organizing Requirements -** The developers prioritize and arrange the requirements in order of importance, urgency and convenience.
- **Negotiation & discussion -** If requirements are ambiguous or there are some conflicts in requirements of various stakeholders.
    - If they are, it is then negotiated and discussed with stakeholders.
    - The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness.
    - Unrealistic requirements are compromised reasonably.
- **Documentation -** All formal & informal, functional and non-functional requirements are documented and made available for next phase processing.

Requirement Elicitation Techniques Requirements Elicitation is the process to find out the requirements for an intended software system by communicating with client, end users, system users and others in the software system development.

There are various ways to discover requirements

Interviews

Interviews are strong medium to collect requirements. Organization may conduct several types of interviews such as:

- **Structured (closed) interviews**, where every single information to gather is decided in advance, they follow pattern .
- **Non-structured (open) interviews**, where information to gather is not decided in advance, more flexible and less biased.
- **Oral interviews**
- **Written interviews**
- **One-to-one interviews which are held between two persons across the table.**
- **Group interviews which are held between groups of participants.** They help to uncover any missing requirement as numerous people are involved.

Surveys

Organization may conduct surveys among various stakeholders by querying about their expectation and requirements from the upcoming system.

Questionnaires

- A document with pre-defined set of objective questions and respective options is handed over to all stakeholders to answer, which are collected and compiled.

Task analysis

- Team of engineers and developers may analyze the operation for which the new system is required.

Domain Analysis

- Every software falls into some domain category.
- The expert people in the domain can be a great help to analyze general and specific requirements.

Brainstorming

- An informal debate is held among various stakeholders and all their inputs are recorded for further requirements analysis.

Prototyping

- Prototyping is building user interface without adding detail functionality for user
- It helps giving better idea of requirements.
- The prototype is shown to the client and the feedback is noted.
- The client feedback serves as an input for requirement gathering.

Observation

- Team of experts visit the client's organization or workplace.
- They observe the actual working of the existing installed systems.
- They observe the workflow at client's end and how execution problems are dealt.
- The team itself draws some conclusions which aid (help)to form requirements expected from the software.

Software Requirements Characteristics

- Gathering software requirements is the foundation of the entire software development project.
- Hence they must be clear, correct and well-defined.

A complete Software Requirement Specifications must be:

- Clear
- Correct
- Consistent
- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source

Software Requirements

- We should try to understand what sort of requirements may arise in the requirement elicitation phase and what kinds of requirements are expected from the software system.

Broadly software requirements should be categorized in two categories:

Functional Requirements

Requirements, which are related to functional aspect of software fall into this category.

They define functions and functionality within and from the software system.

Examples -

- Search option given to user to search from various invoices.
- User should be able to mail any report to management.
- Users can be divided into groups and groups can be given separate rights.
- Should comply business rules and administrative functions.
- Software is developed keeping downward compatibility intact.

Non-Functional Requirements

- Requirements, which are not related to functional aspect of software, fall into this category.
- They are implicit or expected characteristics of software, which users make assumption of.

Non-functional requirements include -

- Security
- Logging
- Storage
- Configuration
- Performance
- Cost
- Interoperability
- Flexibility
- Disaster recovery
- Accessibility

Requirements are categorized logically as

- **Must Have** : Software cannot be said operational without them.
- **Should have** : Enhancing the functionality of software.
- **Could have** : Software can still properly function with these requirements.
- **Wish list** : These requirements do not map to any objectives of software.

User Interface requirements

- UI is an important part of any software or hardware or hybrid system. A software is widely accepted if it is -
    - easy to operate
    - quick in response
    - effectively handling operational errors
    - providing simple yet consistent user interface
- A system is said be good if it provides means to use it efficiently. User interface requirements are briefly mentioned below
    - Content presentation ,Easy Navigation ,Simple interface
    - Responsive , Consistent UI elements , Feedback mechanism
    - Default settings , Purposeful layout, Strategical use of color and texture., Provide help information , User centric approach
    - Group based view settings.

Software System Analyst

- System analyst in an IT organization is a person, who analyzes the requirement of proposed system and ensures that requirements are conceived and documented properly & correctly.
- Role of an analyst starts during Software Analysis Phase of SDLC.
- It is the responsibility of analyst to make sure that the developed software meets the requirements of the client.

System Analysts have the following responsibilities:

- Analyzing and understanding requirements of intended software
- Understanding how the project will contribute in the organization objectives
- Identify sources of requirement
- Validation of requirement
- Develop and implement requirement management plan
- Documentation of business, technical, process and product requirements
- Coordination with clients to prioritize requirements and remove and ambiguity

- Finalizing acceptance criteria with client and other stakeholder

Q2) FEASIBILTY STUDY

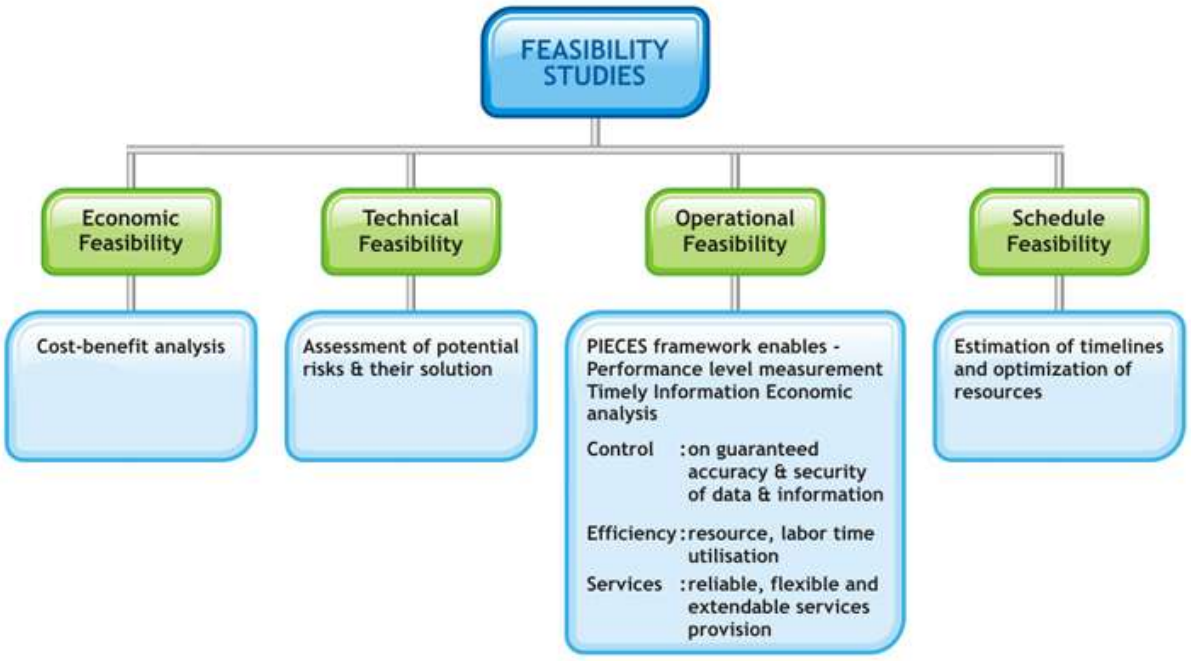
Feasibility is defined as the practical extent to which a project can be performed successfully. To evaluate feasibility, a feasibility study is performed, which determines whether the solution considered to accomplish the requirements is practical and workable in the software

Information such as resource availability, cost estimation for software development, benefits of the software to the organization after it is developed and cost to be incurred on its maintenance are considered during the feasibility study.

The objective of the feasibility study is to establish the reasons for developing the software that is acceptable to users, adaptable to change and conformable to established standards.

Types of Feasibility

1) Technical feasibility
2) Operational feasibility
3) Economic feasibility
4) Schedule Feasibility


FEASIBILITY STUDIES
Economic Feasibility
Technical Feasibility
Operational Feasibility
Schedule Feasibility
Cost-benefit analysis
Assessment of potential risks & their solution
PIECES framework enables - Performance level measurement Timely Information Economic analysis
Control :on guaranteed accuracy & security of data & information
Efficiency :resource, labor time utilisation
Services :reliable, flexible and extendable services provision
Estimation of timelines and optimization of resources

Technical feasibility assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget.
Technical feasibility also performs the following tasks.

- Analyzes the technical skills and capabilities of the software development team members
- Determines whether the relevant technology is stable and established
- Ascertains that the technology chosen for software development has a large number of users so that they can be consulted when problems arise or improvements are required.

Operational feasibility assesses the extent to which the required software performs a series of steps to solve business problems and user requirements. Operational feasibility also performs the following tasks.

- Determines whether the problems anticipated in user requirements are of high priority
- Determines whether the solution suggested by the software development team is acceptable
- Analyzes whether users will adapt to a new software
- Determines whether the organization is satisfied by the alternative solutions proposed by the software development team

Economic feasibility determines whether the required software is capable of generating financial gains for an organization. It involves the cost incurred on the software development team, estimated cost of hardware and software, cost of performing feasibility study, and so on.

It focuses on the issues listed below.

- Cost incurred on software development to produce long-term gains for an organization
- Cost required to conduct full software investigation (such as requirements elicitation and requirements analysis)
- Cost of hardware, software, development team, and training.

Schedule Feasibility - Does the company currently have the time resources to undertake the project? Can the project be completed in the available time?

3) PROBLEMS OF REQUIRMENTS

Problem 1: Customers don't (really) know what they want

Possibly the most common problem in the requirements analysis phase is that customers have only a vague idea of what they need, and it's up to you to ask the right questions and perform the analysis necessary to turn this amorphous vision into a formally-documented software requirements specification that can, in turn, be used as the basis for both a project plan and an engineering architecture.

To solve this problem, you should:

- Ensure that you spend sufficient time at the start of the project on understanding the objectives, deliverables and scope of the project.
- Make visible any assumptions that the customer is using, and critically evaluate both the likely end-user benefits and risks of the project.
- Attempt to write a concrete vision statement for the project, which encompasses both the specific functions or user benefits it provides and the overall business problem it is expected to solve.
- Get your customer to read, think about and sign off on the completed software requirements specification, to align expectations and ensure that both parties have a clear understanding of the deliverable.

Problem 2: Requirements change during the course of the project

The second most common problem with software projects is that the requirements defined in the first phase change as the project progresses. This may occur because as development progresses and prototypes are developed, customers are able to more clearly see problems with the original plan and make necessary course corrections; it may also occur because changes in the external environment require reshaping of the original business problem and hence necessitates a different solution than the one originally proposed. Good project managers are aware of these possibilities and typically already have backup plans in place to deal with these changes.

To solve this problem, you should:

- Have a clearly defined process for receiving, analyzing and incorporating change requests, and make your customer aware of his/her entry point into this process.
- Set milestones for each development phase beyond which certain changes are not permissible — for example, disallowing major changes once a module reaches 75 percent completion.
- Ensure that change requests (and approvals) are clearly communicated to all stakeholders, together with their rationale, and that the master project plan is updated accordingly.

Problem 3: Customers have unreasonable timelines

It's quite common to hear a customer say something like "it's an emergency job and we need this project completed in X weeks". A common mistake is to agree to such timelines before actually performing a detailed analysis and understanding both of the scope of the project and the resources necessary to execute it. In accepting an unreasonable timeline without discussion, you are, in fact, doing your customer a disservice: it's quite likely that the project will either get delayed (because it wasn't possible to execute it in time) or suffer from quality defects (because it was rushed through without proper inspection).

To solve this problem, you should:

- Convert the software requirements specification into a project plan, detailing tasks and resources needed at each stage and modeling best-case, middle-case and worst-case scenarios.
- Ensure that the project plan takes account of available resource constraints and keeps sufficient time for testing and quality inspection.
- Enter into a conversation about deadlines with your customer, using the figures in your draft plan as supporting evidence for your statements. Assuming that your plan is reasonable, it's quite likely that the ensuing negotiation will be both productive and result in a favorable outcome for both parties.

Problem 4: Communication gaps exist between customers, engineers and project managers

Often, customers and engineers fail to communicate clearly with each other because they come from different worlds and do not understand technical terms in the same way. This can lead to confusion and severe miscommunication, and an important task of a project manager, especially during the requirements analysis phase, is to ensure that both parties have a precise understanding of the deliverable and the tasks needed to achieve it.

To solve this problem, you should:

- Take notes at every meeting and disseminate these throughout the project team.
- Be consistent in your use of words. Make yourself a glossary of the terms that you're going to use right at the start, ensure all stakeholders have a copy, and stick to them consistently.

Problem 5: The development team doesn't understand the politics of the customer's organization

An effective manager is one who views the organization as a "contested arena" and understands the importance of power, conflict, negotiation and coalitions. Such a manager is not only skilled at operational and functional tasks, but he or she also understands the importance of framing agendas for common purposes, building coalitions that are united in their perspective, and persuading resistant managers of the validity of a particular position.

These skills are critical when dealing with large projects in large organizations, as information is often fragmented and requirements analysis is hence stymied by problems of trust, internal conflicts of interest and information inefficiencies.

To solve this problem, you should:

- Review your existing network and identify both the information you need and who is likely to have it.
- Cultivate allies, build relationships and think systematically about your social capital in the organization.
- Persuade opponents within your customer's organization by framing issues in a way that is relevant to their own experience.
- Use initial points of access/leverage to move your agenda forward.

4) SOFTWARE REQUIREMENTS ANALYSIS

Requirements Analysis

- Requirements Analysis is the process of understanding the customer needs and expectations from a proposed system or application and is a well-defined stage in the Software Development Life Cycle model.
- Requirements are a description of how a system should behave or a description of system properties or attributes.
- It can alternatively be a statement of ‘what’ an application is expected to do, not ‘how’.

- Software engineering task that bridges the gap between system level requirements engineering and software design.
- Software requirements analysis may be divided into give areas of effort
    - Problem recognition
    - Evaluation and synthesis
    - Modeling
    - Specification
    - Review
- Requirements analysis encompasses those tasks that go into determining the requirements of a new or altered system, taking account of the possibly conflicting requirements of the various stakeholders, such as users.
- Requirements analysis is critical to the success of a project.
- It is sometimes referred to loosely by names such as requirements gathering, requirements capture, or requirements specification.
- The term “requirements analysis” can also be applied specifically to the analysis proper.
- Requirements must be measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.
- Five common errors in requirements analysis:
    - Customers do not really know what they want
    - Requirements change during the course of the project
    - Customers have unreasonable timelines
    - Communication gaps exist between customers, engineers and project managers
    - The development team does not understand the politics of the customer’s organization

Requirements Elicitation for Software

- Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process.
    - Initiating the Process
        - The most commonly used requirements elicitation technique is to conduct a meeting or interview

- Facilitated Application Techniques
    - This approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of solution, negotiate different approaches and specify a preliminary set of solution requirements
- Quality Function Deployment
    - A technique that translates the needs of the customer into technical requirements for software
    - QFD identifies three types of requirements
        - Normal requirements – the objectives and goals that are stated for a product or system during meeting with the customer
        - Expected requirements – these requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them
        - Exciting requirements – these features go beyond the customer’s expectations and prove to be very satisfying when present
- Interviews
    - A “traditional” means of eliciting requirements.
    - It is important to understand the advantages and limitations of interviews and how they should be conducted.

- Use-Cases
    - As requirements are gathered as part of informal meetings, FAST, or QFD, the software engineer can create a set of scenarios that identify a thread of usage for the system to be constructed.
- Prototype
    - A valuable tool for clarifying unclear requirements.
    - They can act in a similar way to scenarios by providing users with a context within which they can better understand what information they need to provide.

        - There is a wide range of prototyping techniques, from paper mock-ups of screen designs to beta-test versions of software products, and a strong overlap of their use for requirements elicitation and the use of prototypes for requirements validation.

5) ANALYSIS PRINCIPLES

- All analysis methods are related by a set of operational principles
    - The information domain of a problem must be represented and understood
    - The functions that the software is to perform must be defined
    - The behavior of the software must be represented
    - The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered fashion
    - The analysis process should move from essential information toward implementation detail
- In addition to these operational analysis principles, a set of guiding principles for requirements engineering are suggested as
    - Understand the problem before you begin to create the analysis model
    - Develop prototypes that enable a user to understand how human/machine interaction will occurs
    - Record the origin of an the reason for every requirement
    - Use multiple view of requirements
    - Rank requirements
    - Work to eliminate ambiguity

Specification

- Specification principles
    - Separate functionality from implementation
    - Develop a model of the desired behavior of a system that encompasses data and functional response of a system to various stimuli from the environment
    - Establish the context in which software operates by specifying the manner in which other systems components interact with software

  - Define the environment in which the system operates and indicate how "a highly intertwined collection of agents react to stimuli in the environment (changes to objects) produced by those agents"
  - Create a cognitive model rather than a design or implementation model. The cognitive model describes a system as perceived by its user community
  - Recognize that "the specifications must be tolerant of incompleteness and augmentable." A specification is always a model-an abstraction-of some real situation that is normally quite complex. Hence, it will be incomplete and will exist at many level of detail
  - Establish the content and structure of a specification in a way that will enable it to be amenable to change

- Representation
  - Representation format and content should be relevant to the problem
  - Information contained within the specification should be nested
  - Diagrams and other notational forms should be restricted in number and consistent in use
  - Representations should be revisable

The software requirements specification

  - Is produced at the culmination of analysis task.
  - The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.
  - Format of software requirements specification:
    - Introduction
    - Information description
    - Functional description
    - Behavioral description
    - Validation criteria
    - Bibliography and appendix

6) ANALYSIS MODEL

- Analysis model operates as a link between the 'system description' and the 'design model'.
- In the analysis model, information, functions and the behaviour of the system is defined and these are translated into the architecture, interface and component level design in the 'design modeling'.

Elements of the analysis model

1. Scenario based element

- This type of element represents the system user point of view.
- Scenario based elements are use case diagram, user stories.

2. Class based elements

- The object of this type of element manipulated by the system.
- It defines the object,attributes and relationship.
- The collaboration is occurring between the classes.
- Class based elements are the class diagram, collaboration diagram.

3. Behavioral elements

- Behavioral elements represent state of the system and how it is changed by the external events.
- The behavioral elements are sequenced diagram, state diagram.

4. Flow oriented elements

- An information flows through a computer-based system it gets transformed.
- It shows how the data objects are transformed while they flow between the various system functions.
- The flow elements are data flow diagram, control flow diagram.

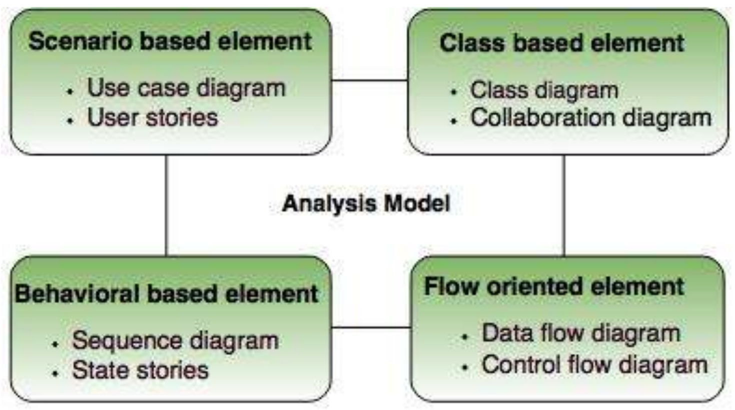


Fig. - Elements of analysis model

Analysis Rules of Thumb

The rules of thumb that must be followed while creating the analysis model.

The rules are as follows:

- The model focuses on the requirements in the business domain. The level of abstraction must be high i.e there is no need to give details.
- Every element in the model helps in understanding the software requirement and focus on the information, function and behaviour of the system.
- The consideration of infrastructure and nonfunctional model delayed in the design. **For example,** the database is required for a system, but the classes, functions and behavior of the database are not initially required. If these are initially considered then there is a delay in the designing.
- Throughout the system minimum coupling is required. The interconnections between the modules is known as 'coupling'.
- The analysis model gives value to all the people related to model.
- The model should be simple as possible. Because simple model always helps in easy understanding of the requirement.

Concepts of data modeling

- Analysis modeling starts with the data modeling.
- The software engineer defines all the data object that proceeds within the system and the relationship between data objects are identified.

Data objects

- The data object is the representation of composite information.
- The composite information means an object has a number of different properties or attribute.

  For example, Height is a single value so it is not a valid data object, but dimensions contain the height, the width and depth these are defined as an object.

Data Attributes

Each of the data object has a set of attributes.

Data object has the following characteristics:

- Name an instance of the data object.
- Describe the instance.
- Make reference to another instance in another table.

Relationship

Relationship shows the relationship between data objects and how they are related to each other.

Cardinality

Cardinality state the number of events of one object related to the number of events of another object.

The cardinality expressed as:

One to one (1:1)

One event of an object is related to one event of another object.

For example, one employee has only one ID.

One to many (1:N)

One event of an object is related to many events.

For example, One collage has many departments.

Many to many(M:N)

Many events of one object are related to many events of another object.

For example, many customer place order for many products.

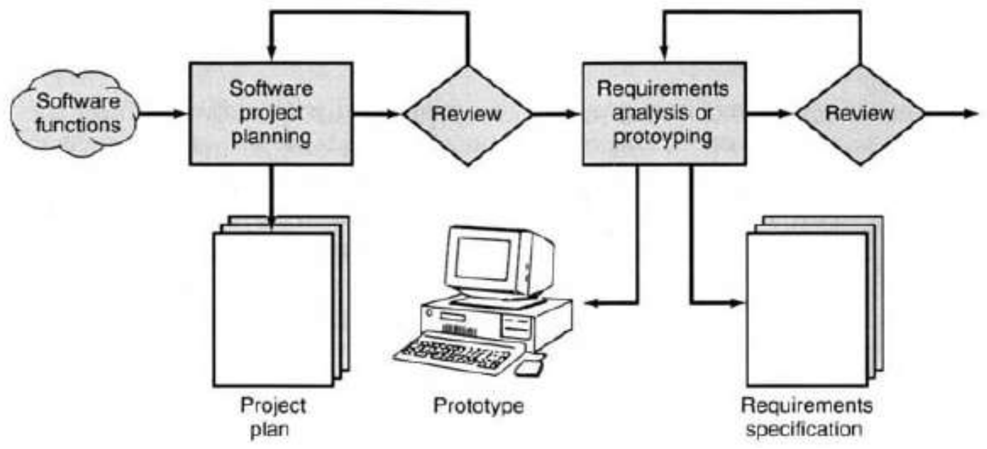
Modality If an event relationship is an optional then the modality of relationship is zero.

- If an event of relationship is compulsory then modality of relationship is one.

7)ANALYSIS PROCESS

The objective of requirements analysis then, is to create a "requirements specification document " or a " target document ", that describes in as much detail and in an unambiguous a manner as possible, exactly what the product should do. This requirements document will then form the basis of the subsequent design phase. The requirements document may well contain

Analysis then can be summed by the activities in the following diagram


Software functions
Software project planning
Review
Requirements analysis or protoyping
Review
Project plan
Prototype
Requirements specification

A Project plan including details of delivery times and cost estimates.
• A model of the software's functionality and behaviour in the form of '*data flow/ UML diagrams'* and, where appropriate, any performance and data considerations, any input/output details etc.
• The results of any prototyping so that the appearance of the software and how it should behave can be shown to the designer. This may include a *users manual*.
• Any formal Z/VDM specifications for the system requirements.
• Any verification test data that may be used to determine that the finished product conforms to the agreed specification.

UNIT - III

SOFTWARE DESIGN

UNIT - III

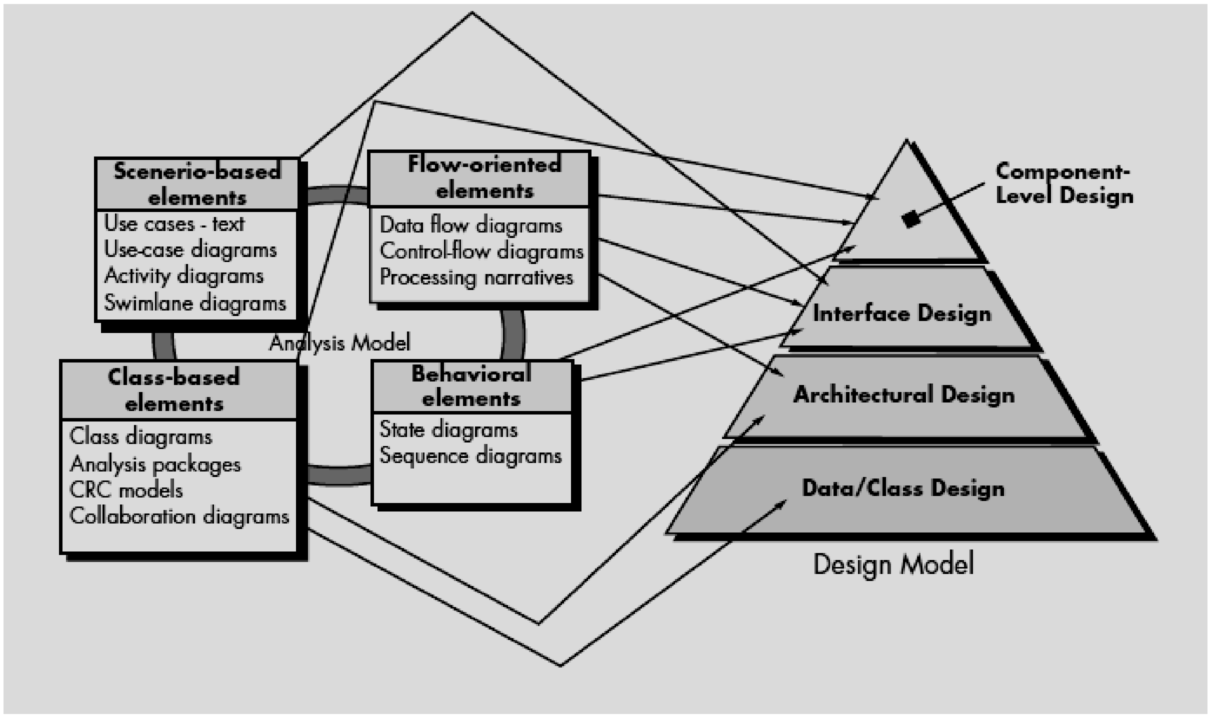
SOFTWARE DESIGN

- Software Design
- Abstraction
- Modularity
- Software Architecture
- Effective Modular Design
- Cohesion and Coupling
- Architectural Design and Procedural Design
- Data Flow Oriented Design

DESIGN CONCEPTS

- Software design encompasses the set of principles, concepts, and practices
- that lead to the development of a high-quality system or product.

1)DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING OR ELEMENTS OF DESIGN MODEL



The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task.

- The **data design** transforms the information domain model created during analysis into the data structures that will be required to implement the software.
- The **architectural design** defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements and have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied.
- The **interface design** describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior.
- The **component-level design** transforms structural elements of the software architecture into a procedural description of software components.
- **THE DESIGN PROCESS**

- Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.

- **Software Quality Guidelines and Attributes**

- • The design must implement all of the explicit requirements contained in the requirements model
- • The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- • The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

- **Quality Guidelines**
- In order to evaluate the quality of a design representation,you and other members of the software team must establish technical criteria for good design.
- A design should exhibit an architecture that
- (1) Has been created using recognizable architectural styles or patterns.
- (2) Is composed of components that exhibit good design characteristics
- (3) Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
- **Quality Attributes.**
- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- *Usability* is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- • *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability torecover from failure, and the predictability of the program.
- • *Performance* is measured by considering processing speed, response time,resource consumption, throughput, and efficiency.
- • *Supportability* combines the ability to extend the program (extensibility),adaptability, serviceability—these three attributes represent a more commonterm, *maintainability*—and in addition, testability, compatibility, configurability the ease with which a system

2. Explain the Abstraction?

- **Highest level of abstraction** : Solution is slated in broad terms using the language of the problem environment
- **Lower levels of abstraction** : More detailed description of the solution is provided
- **Procedural abstraction: :**Refers to a sequence of instructions that has a specific and limited function. Ex The word **open** of a door which implies a long sequence of procedural steps (e.g., walk to the door, grasp the knob, pull the door& step away from moving door etc)

- **Data abstraction:** Named collection of data that describe a data object (Ex: door would encompass a set of attributes like door type,weight,dimensions etc.

EXPLAIN ABOUT DESIGN CONCEPTS

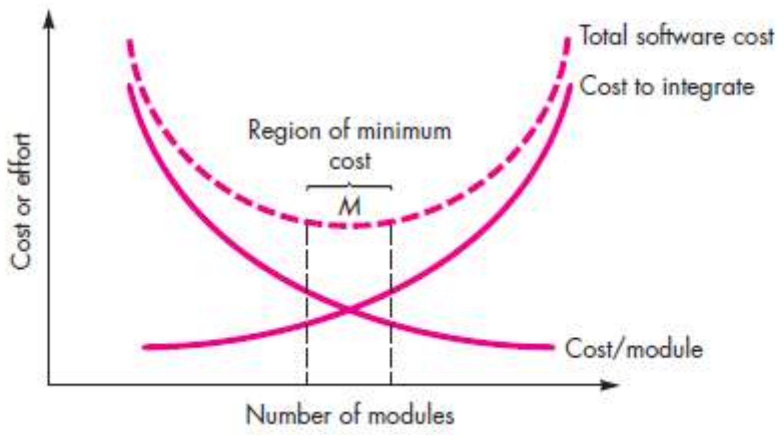
- A set of fundamental software design concepts has evolved over the history of software
- engineering. Each provides the software designer with a foundation from which more sophisticated design methods can be applied.
- *Each helps you answer the following questions:*
- • What criteria can be used to partition software into individual components?
- • How is function or data structure detail separated from a conceptual representation
- of the software?
- • What uniform criteria define the technical quality of a software design?
- **1.Abstraction**
- As different levels of abstraction are developed, you work to create both procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob andpull door, step away from moving door, etc.).
- A *data abstraction* is a named collection of data that describes a data object.
- In the context of the procedural abstraction *open,* we can define a data abstraction called **door.**
- **2.Architecture**
- *Software architecture* alludes to "the overall structure of the software and the ways
- in which that structure provides conceptual integrity for a system"
- One goal of software design is to derive an architectural rendering of a system.
- This rendering serves as a framework from which more detailed design activities are conducted.
- A set of architectural patterns enables a software engineer to solve common design problems.
- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.
- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability,security, adaptability, and other system characteristics.
- **Families of related systems.** The architectural design should draw upon repeatable
- patterns that are commonly encountered in the design of families of similar systems

- *Structural models* represent architecture as an organized collection of program components.
- *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of
- applications.
- *Dynamic models* address the behavioral aspects of the program architecture,indicating how the structure or system configuration may change as a function of external events.
- *Process models* focus on the design of the business or technical process that the system must accommodate. Finally, *functional models* can be used to represent the functional hierarchy of a system.
- **3.Patterns**

- The intent of each design pattern is to provide a description that enables a
- designer to determine
- (1) Whether the pattern is applicable to the current work
- (2) Whether the pattern can be reused (hence, saving design time),
- (3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurall different pattern.
- **4.Separation of Concerns**

- *Separation of concerns* is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently *a concern* is a feature or behavior that is specified as part of the requirements model for the software
- **5. Modularity**

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



- **6.Information Hiding**
- The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance.

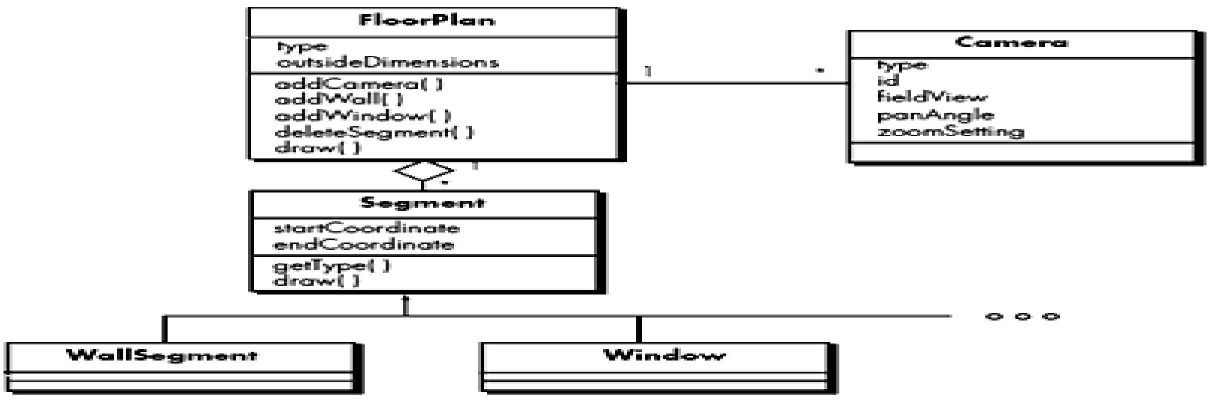
- **7.Functional Independence**

- The concept of functional independence is a direct outgrowth of separation of concerns,
- modularity, and the concepts of abstraction and information hiding.
- Functional independence is achieved by developing modules with "singleminded"
- function and an "aversion" to excessive interaction with other modules.
- Independence is assessed using two qualitative criteria: cohesion and coupling.
- *Cohesion* is an indication of the relative functional strength of a module.
- *Coupling* is an indication of the relative interdependence among modules.
- **8.Refinement**

- Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.
- **9.Refactoring**

- *Refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behaviour"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design]
- yet improves its internal structure."
- **10.Design Classes**

- *User interface classes* define all abstractions that are necessary for human computer
- interaction (HCI).
- *Business domain classes* are often refinements of the analysis classes
- P*rocess classes* implement lower-level business abstractions required to fully manage the business domain classes.
- *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software.
  System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.



Design class for FloorPlan and composite aggregation

- **WRITE ABOUT SOFTWARE ARCHITECTURE**

- The architecture is not the operational software. Rather, it is a representation that enables you to
- (1) Analyze the effectiveness of the design in meeting its stated requirements.
- (2) Consider architectural alternatives at a stage when making design changes is still relatively easy
- (3) Reduce the risks associated with the construction of the software.

- **Why Is Architecture Important?**

- • Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- • The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- • Architecture "constitutes a relatively small, intellectually graspable model of
- how the system is structured and how its components work together"
- **Architectural Descriptions**

- An architectural description is actually a set of work products that reflect different views of the system.
- **Architectural Decisions**

- Each view developed as part of an architectural description addresses a specific stakeholder concern.

Explain the Modularity?

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called **modules**, that are integrated to satisfy problem requirements. Modularity is the single attribute of software that allows a program to be intellectually manageable.

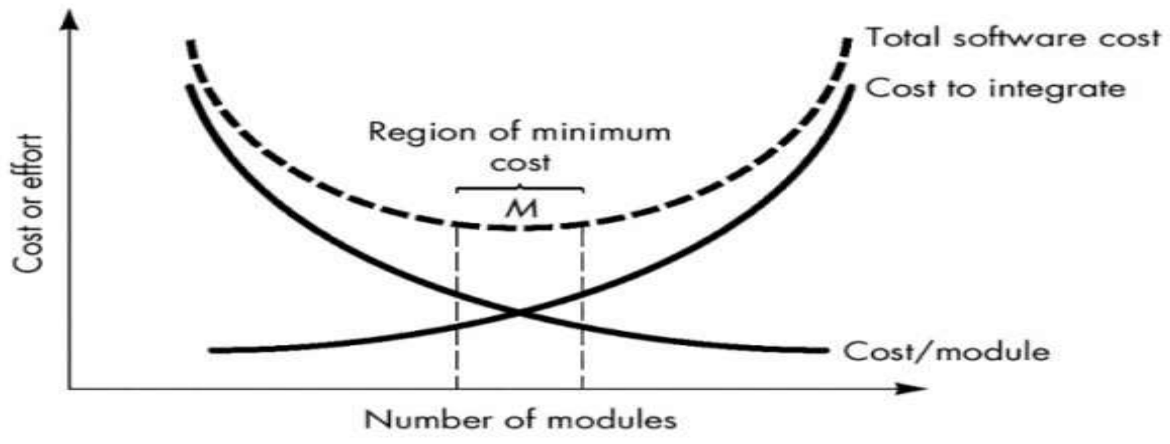
Modular decomposability : If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

Modular composability : If a design method enables existing design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Modular understandability : If a module can be understood as a standalone unit it will be easier to build and easier to change.

Modular continuity : If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.

Modular protection : If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.



WRITE ABOUT EFFECTIVE MODULAR DESIGN

A modular design reduces complexity , facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

Functional independence

- Modules have high cohesion and low coupling
- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- Software with effective modularity is easier to develop because function may be compartmentalized and interfaces are simplified.
- Independence is measured using two qualitative criteria: cohesion and coupling

COHESION AND COUPLING

Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other.

Cohesion is a measure of the function strength of a module, whereas the coupling between two modules is a measure of the degree of interaction between the two modules.

Coupling

Two modules are said to be highly coupled, in either of the following two situations arises.

- If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled
- If the interactions occur through some shared data, them also we say that they are highly coupled.

Cohesion

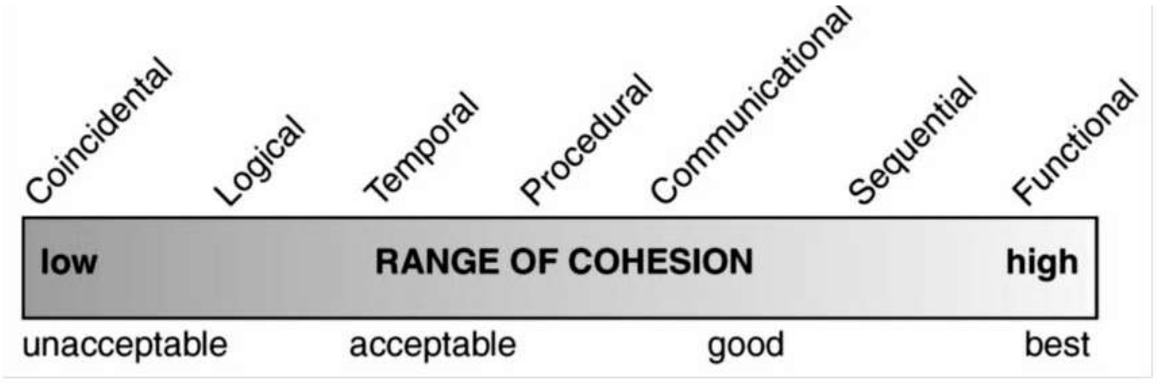
Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentence of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution. When the functions of the module cooperate with each other for performing a single objective, then the module has good cohesion. If the functions of the module do very different things and do not cooperate with each other to perform a single piece of work, then module has very poor cohesion.

Functional Independence *A module independence that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules.*

Functional independence is a key to any good design primarily due to the following advantages it offers:

Classification of Cohesiveness

Cohesiveness of a module is the degree to which the different functions of the module cooperate to work towards a single objective. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible.



Coincidental cohesion

A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all.

Functional cohesion

A module is said to possess functional cohesion, if different of the modules cooperate to complete a single task

Logical cohesion

A module is said to be logically cohesive, if all elements of the module perform similar operations such as error handling data input, data output, etc. an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheet, salary slips, annual reports, etc.

Temporal cohesion

When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion. As an example, consider the following situation.

Procedural cohesion A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.

Communicational cohesion

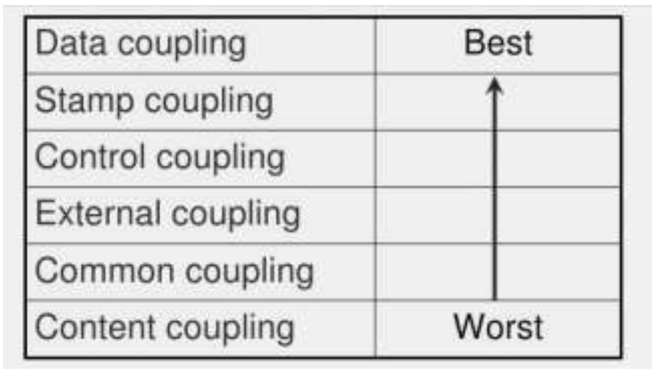
A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure.

Sequential cohesion

A module is said to possess sequential cohesion, if the different functions of the module executed in a sequence, and the output from one function is input to the next in the sequence.

Classification of Coupling

The coupling between two modules indicates the degree of interdependence between them. *The degree of coupling between two modules depends on their interface complexity*



Data coupling	Best
Stamp coupling	↑
Control coupling	
External coupling	
Common coupling	
Content coupling	Worst

Data coupling

Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem relate not used for control purposes.

Stamp coupling

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

Control coupling

Control coupling exist between two modules, if data from one module is used to direct the order of instruction execution in other.

Common coupling

Two modules are common coupled, if they share some global data items.

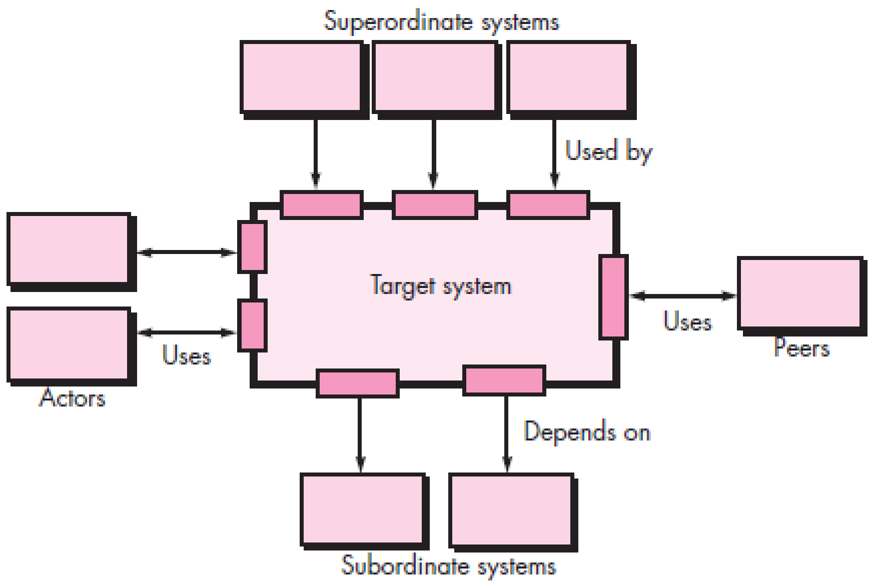
Content coupling

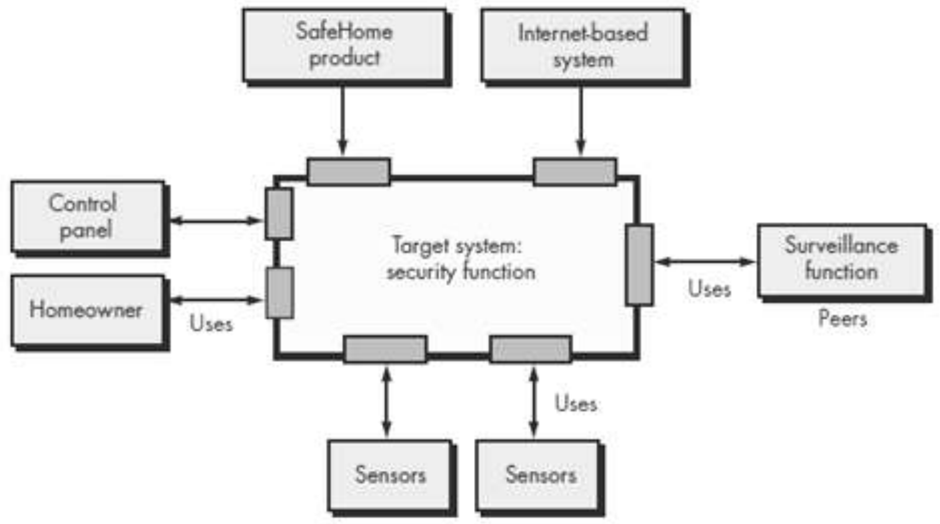
Content coupling exists between two modules, if they share code. That is a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

ARCHITECTURAL DESIGN

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction

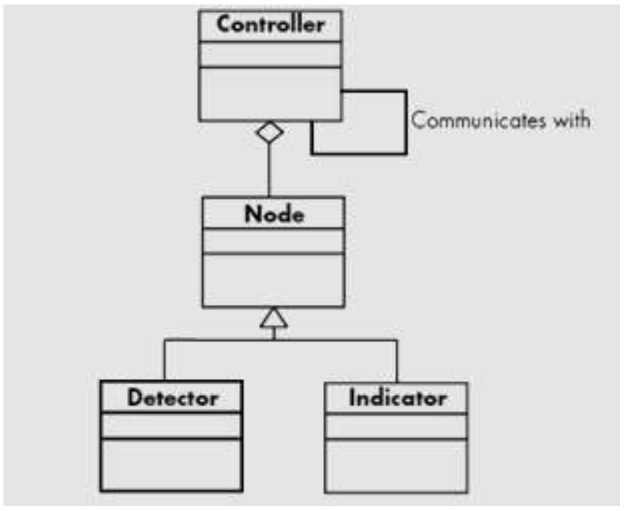
Representing the System in Context
• *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
• *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
• *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
• *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.





Defining Archetypes

An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system.



Node. Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.

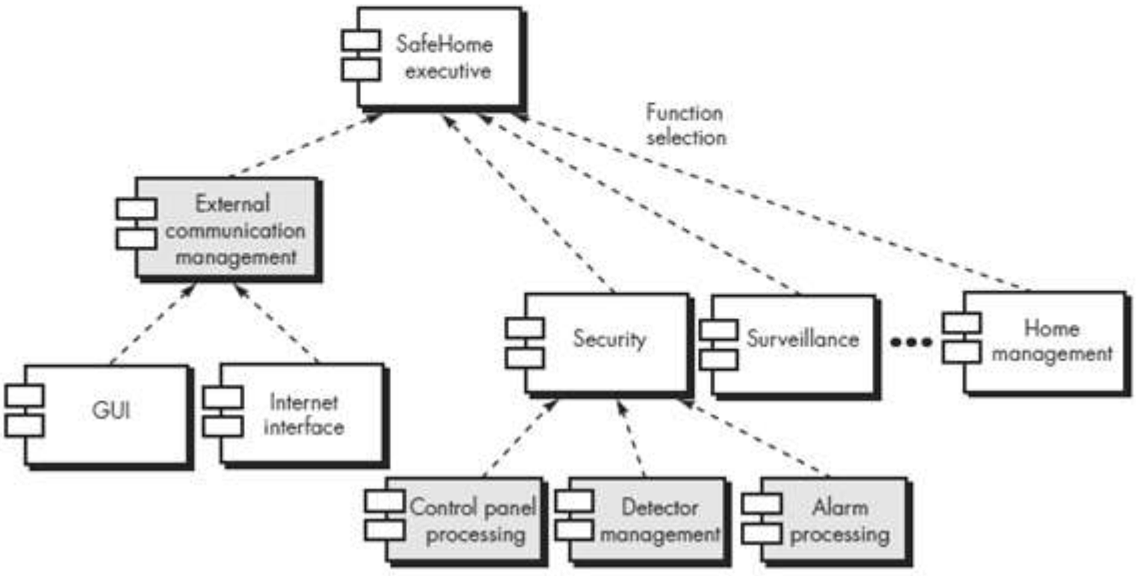
• **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.

Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

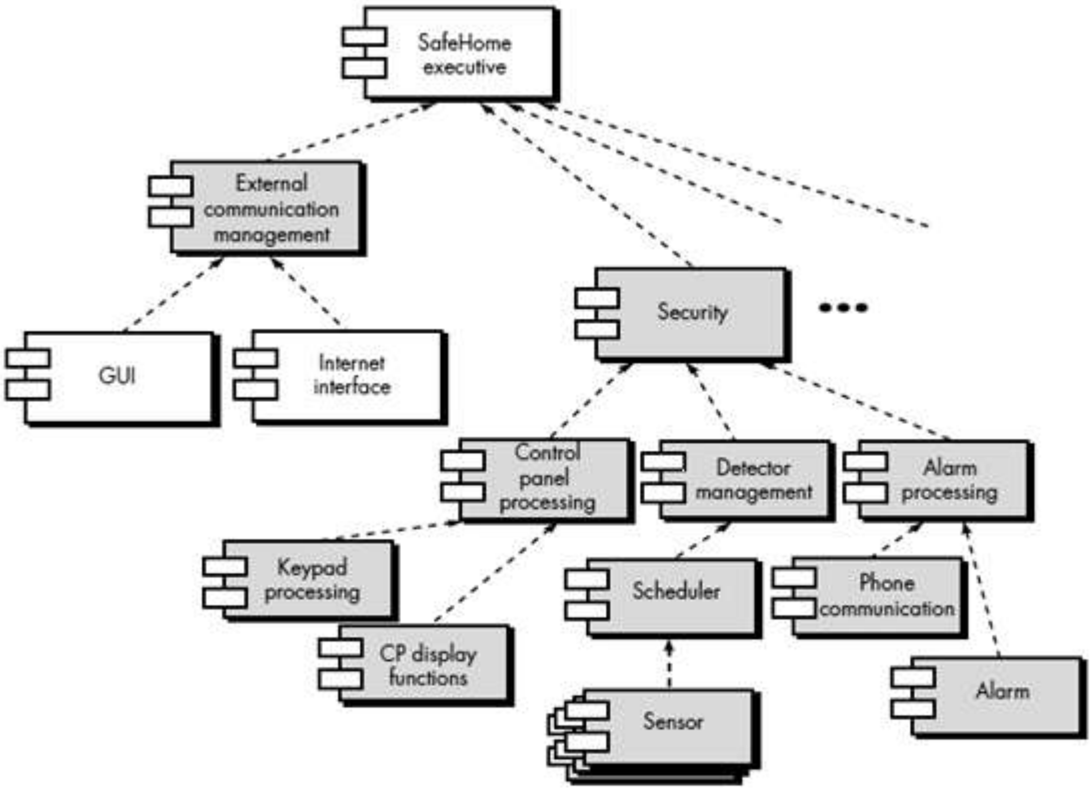
• **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

Refining the Architecture into Components

External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
• *Control panel processing*—manages all control panel functionality.
• *Detector management*—coordinates access to all detectors attached to the system.
• *Alarm processing*—verifies and acts on all alarm conditions.
Each of these top-level components would have to be elaborated iteratively



Describing Instantiations of the System
the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.
Below diagram an instantiation of the *SafeHome* architecture for the security system.



Characteristics of a Good Software Design

The desirable characteristics that a good software design should have are as follows:

• *Correctness*

• *Efficiency*

• *Understandability*

• *Maintainability*

• *Simplicity*

• *Completeness*

• *Verifiability*

• *Portability*

• *Modularity*

• *Reliability*

• *Reusability*

Design Principles

• There are certain design concepts and principles that govern the building of quality software designs.

• Some of the common concepts of software design are:

□ □*Abstraction*

□ □*information hiding*

□ □*functional decomposition*

□ □*design strategies*

□ □*Modularity*

□ □*and modular design*

VARIOUS LEVELS IN S/W DESIGN?

a) Architectural design

b) High level design

c) Detailed design

Application architectures

Application systems are designed to meet an organizational need. As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements. A **generic application architecture** is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements. application architectures can be used as a:

- Starting point for architectural design.
- Design checklist.
- Way of organizing the work of the development team.
- Means of assessing components for reuse.
- Vocabulary for talking about application types.

Examples of **application types**:

Data processing applications

Data driven applications that process data in batches without explicit user intervention during the processing.

Transaction processing applications

Data-centred applications that process user requests and update information in a system database.

Event processing systems

Applications where system actions depend on interpreting events from the system's environment.

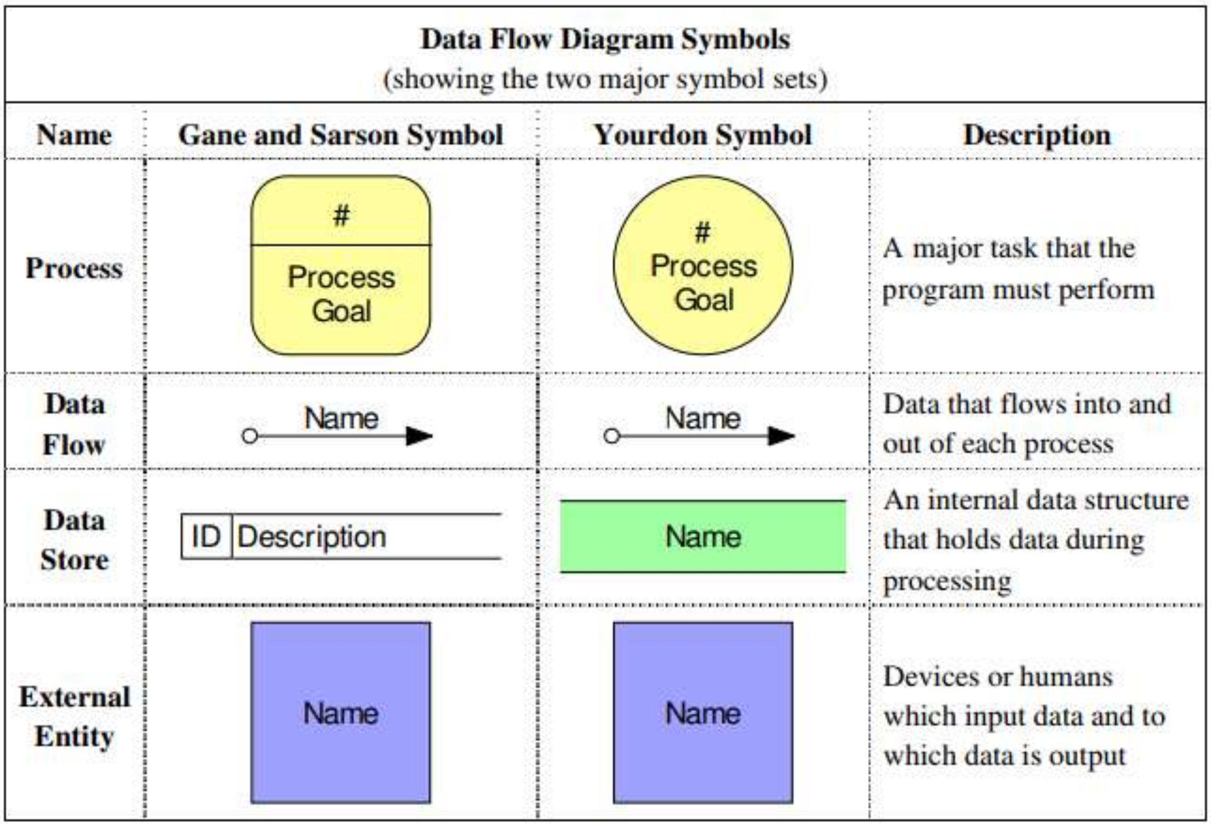
Language processing systems

Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.

PROCEDURAL DESIGN METHODOLOGY

A design methodology combines a systematic set of rules for creating a program design with diagramming tools needed to represent it. Procedural design is best used to model programs that have an obvious flow of data from input to output. It represents the architecture of a program as a set of interacting processes that pass data from one to another. Design Tools The two major diagramming tools used in procedural design are data flow diagrams and structure charts. Data Flow Diagrams A data flow diagram (or DFD) is a tool to help you discover and document the program's major processes. The following table shows the symbols used and what each

Represents



Data Flow Diagram Symbols
(showing the two major symbol sets)

Name	Gane and Sarson Symbol	Yourdon Symbol	Description
Process	# Process Goal	# Process Goal	A major task that the program must perform
Data Flow	Name	Name	Data that flows into and out of each process
Data Store	ID Description	Name	An internal data structure that holds data during processing
External Entity	Name	Name	Devices or humans which input data and to which data is output

The DFD is a conceptual model – it doesn't represent the computer program, it represents what the program must accomplish. By showing the input and output of each major task.

Data Flow Diagram

Data flow diagram is graphical representation of flow of data in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

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

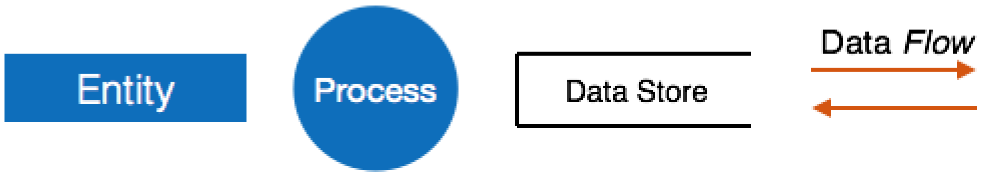
Types of DFD

Data Flow Diagrams are either Logical or Physical.

- **Logical DFD** - This type of DFD concentrates on the system process, and flow of data in the system.For example in a Banking software system, how data is moved between different entities.
- **Physical DFD** - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

DFD Components

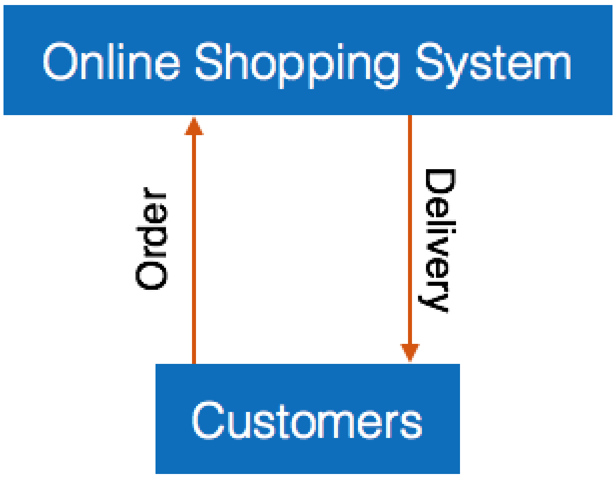
DFD can represent Source, destination, storage and flow of data using the following set of components -



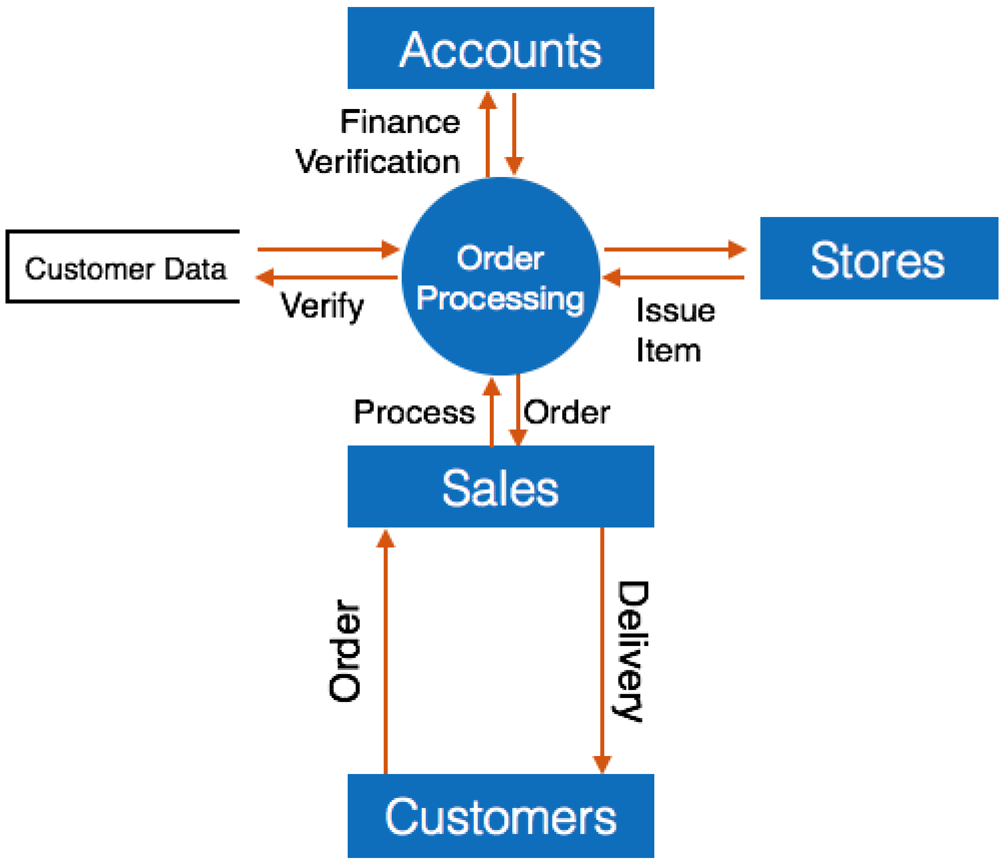
- **Entities** - Entities are source and destination of information data. Entities are represented by a rectangles with their respective names.
- **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- **Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.
- **Data Flow** - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

Levels of DFD

- **Level 0** - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



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

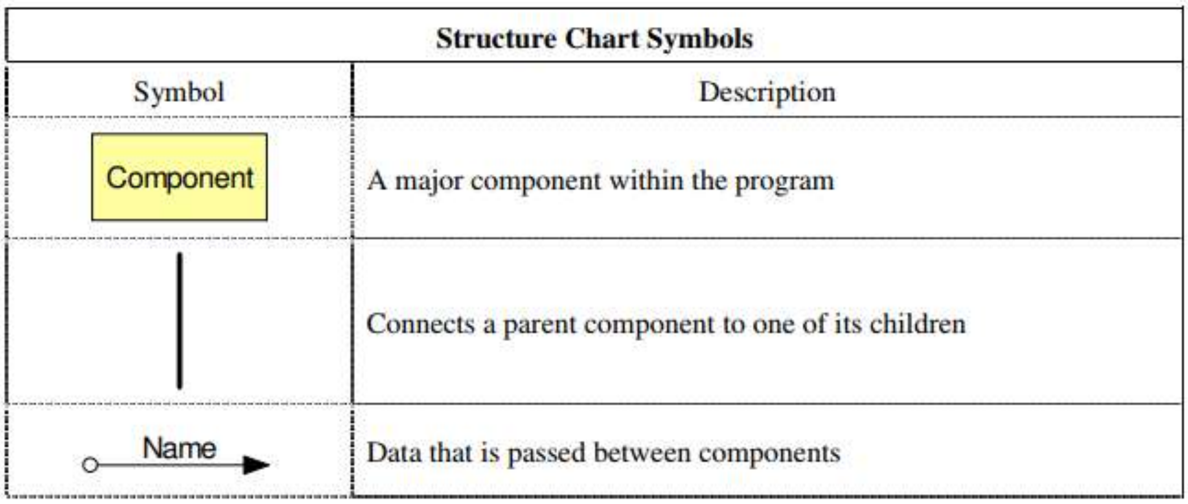


- **Level 2** - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

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

Structure Charts

A structure chart is a tool to help you derive and document the program's architecture. It is similar to an organization chart.



Structure Chart Symbols	
Symbol	Description
Component	A major component within the program
│	Connects a parent component to one of its children
○ Name ▶	Data that is passed between components

When a component is divided into separate pieces, it is called the parent and its pieces are called its children. The structure chart shows the hierarchy between a parent and its children.

The *procedural design* is often understood as a software design process that uses mainly control commands such as: sequence, condition, repetition, which are applied to the predefined data.

Sequences serve to achieve the processing steps in order that is essential in the specification of any algorithm.

Conditions provide facilities for achieving selected processing according to some logical statement.

Repetitions serve to achieve loopings during the computation process.

These three commands are implemented as ready programming language constructs.

The programming languages that provide such command constructs are called *imperative programming languages*.

The software design technique that relies on these constructs is called *procedural design*, or also *structured design*.

UNIT - IV

USER INTERFACE DESIGN & REAL TIME SYSTEMS

USER INTERFACE DESIGN & REAL TIME SYSTEMS

- User Interface Design
- Human Factors
- Human Computer Interaction
- Human - Computer Interface Design
- Interface Design
- Interface Standards

Software User Interface Design

1)What is User Interface design?or explain the human factors in user interface design?

User interface design creates an effective communication medium between a human and a computer. Set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

- **Place the user in control:** Define interactions modes in a way that does not force a user into unnecessary actions. Provided for flexible interactions
- **Support internal locus of control** - Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.
- **Reduce short-term memory load** - The limitation of human information processing in short-term memory requires the displays to be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

2) VARIOUS DESIGN ISSUES IN USER INTERFACE DESIGN?

a) Response time b) Help facilities c) Error handling d) Menu and command labeling

e) Application accessibility f) Internationalization

3) WHAT IS A USER INTERFACE DESIGN PATTERN?

Uid patterns are recurring solution that solve common design problems. Design patterns are standard reference points for the experienced user interface designer.

Patterns are available for a) page layout b) forms and input c) tables d) data manipulation e) searching f) e commerce g) page elements

4)USER INTERFACE DESIGN MODELS

- Once interface analysis has been completed, all tasks (or objects and actions)
- required by the end user have been identified in detail and the interface design
- activity commences. Interface design, like all software engineering design, is an iterative
-
- process. Each user interface design step occurs a number of times, elaborating
- and refining information developed in the preceding step.
- Although many different user interface design models have
- been proposed, all suggest some combination of the following steps:
- **1.** Using information developed during interface analysis define
- interface objects and actions (operations).

- **2.** Define events (user actions) that will cause the state of the user interface to
- change. Model this behavior.
- **3.** Depict each interface state as it will actually look to the end user.
- **4.** Indicate how the user interprets the state of the system from information provided through the interface

In some cases, you can begin with sketches of each interface state (i.e., what the user interface looks like under various circumstances) and then work backward to define objects, actions, and other important design information. Regardless of the sequence of design tasks, you should (1) always follow the golden rules discussed (2) model how the interface will be implemented, and (3) consider the environment (e.g., display technology, operating system, development tools)that will be used.

Human Factors

Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and Highsmith [Coc01a] state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process molds to the needs of the people and team,* not the other way around.2

If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

Competence. In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related

skills, and overall knowledge of the process that the team has chosen to

apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and

project issues.
Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

5)Guidelines for User Interface Design

User will interact only with the front end, look and the feel of the website or application. They don't think about the back end. The main purpose to engaged user when they visit particular website.

Following are the considerations or guidelines for a good user interface design:

- The interface design must be a **user-centered** product and should be incorporated into the final product.
- The interface design should be **simple and intuitive** so users can understand quickly and effectively without instructions.
- **Transparency** must be in user interface design. It helps user to feel like they are right through the computer.
- The interface design must be **customizable** that should be allow users to select objects from various available forms.
- The interface design must **support long-term memory retrieval** by providing user's information rather than having to recall that information and reduce user's memory load.
- The interface must **provide easy access to the common features** and frequently used actions. Do not try to put every bit of information in one main window, use second window for information.
- The interface must be **component oriented** so that the modules can easily modified and replaced without affecting the other parts of the system.
- The UI design should **integrate** smoothly with other applications such as MS-Office and notepad.
- The design must be **separated** from the logic of the system through its implementation for increasing reusability and maintainability.
- The UI design must provide some indications to the user so that they can understand and acknowledging that the action has taken place successfully.

6)What are the UI standards ?

We also realized that for our user interface standards to be useful to programmers, they needed to be set up in a way that made it easy for developers to find the relevant standard. To make it easy for developers to find the information that they need, we organized our user interface standards into four major areas:

- Navigation: We needed to develop a standard method for navigating through our applications. In the past we'd used switchboards, command buttons on forms, and drop-down menus. We decided to standardize on the use of one form of navigation, drop-down menus, which we felt would provide us the most flexibility in all of our applications.
- Forms: We needed to develop a standard method for laying out and presenting information on forms, including methods of navigating between various parts of the form, standard colors for forms, and some of the basic functionality that should be built into all forms.
- Reports: We needed to develop a standard design and layout for all reports. We believed it necessary to develop standard report design guideline (all reports include a printed date, page 1 of __, report name, standard font, and so on) and techniques for permitting users to select reports for printing that would be both easy to maintain and easy for a user to understand.
- Documentation: Last, but not least, we needed standards around the documentation we'd produce, both for systems and user documentation. Our programming standards and the use of FMS tools such as Total Access Analyzer, to a large part, addressed the system documentation requirements. However, we needed to develop a standard for user documentation including

Human–computer interaction ?

- Human–computer interaction (HCI), alternatively man–machine interaction (MMI) or computer–human interaction (CHI) is the study of interaction between people (users) and computers.
- With today's technology and tools, and our motivation to create really effective and usable interfaces and screens, why do we continue to produce systems that are inefficient and confusing or, at worst, just plain unusable? Is it because:
- We don't care?
- We don't possess common sense?
- We don't have the time?
- We still don't know what really makes good design?

- **Definition of HCI:**
- "Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them."
- **Goals:**
- 1.A basic goal of HCI is to improve the interactions between users and computers by making computers more usable and receptive to the user's needs.

- 2.A long term goal of HCI is to design systems that minimize the barrier between the human's cognitive model of what they want to accomplish and the computer's understanding of the user's task
- **Why is HCI important?**
- User-centered design is getting a crucial role!
- It is getting more important today to increase competitiveness via HCI studies.
- High-cost e-transformation investments
- Users lose time with badly designed products and services
- Users even give up using bad interface
- **Defining the User Interface:**
- User interface, design is a subset of a field of study called *human-computer interaction*
- (HCI).Human-computer interaction is the study, planning, and design of how people and computerswork together so that a person's needs are satisfied in the most effective way.
- **HCI designers must consider a variety of factors:**
- What people want and expect, physical limitations and abilities people possess,
- How information processing systems work,
- What people find enjoyable and attractive.
- Technical characteristics and limitations of the computer hardware and software
- must also be considered.
- The *user interface* is the part of a computer and its software that people can see, hear, touch, talk to, or otherwise understand or direct.
- The user interface has essentially two components: input and output.
- *Input* is how a person communicates his / her needs to the computer.
- Some common input components are the keyboard, mouse, trackball, one's finger, and one's
- voice.
- *Output* is how the computer conveys the results of its computations and requirements to the
- user.
- **The Importance of Good Design:**
- A well-designed interface and screen is terribly important to our users
- A screen's layout and appearance affect a person in a variety of ways. If they are con-fusing and inefficient, people will have greater difficulty in doing their jobs and will make more mistakes.
- Poor design may even chase some people away from a system permanently. It can also lead to aggravation, frustration, and increased stress.
-
- Three *golden rules:*
- **1.** Place the user in control.
- **2.** Reduce the user’s memory load.
- **3.** Make the interface consistent.
- These golden rules actually form the basis for a set of user interface design principles
- that guide this important aspect of software design.

- **1)Place the User in Control**
- During a requirements-gathering session for a major new information system, a key
- user was asked about the attributes of the window-oriented graphical interface.
- **Define interaction modes in a way that does not force a user into unnecessary or undesired actions.** An interaction mode is the current state of the interface.
- For example, if *spell check* is selected in a word-processor menu, the software
- moves to a spell-checking mode. There is no reason to force the user to remain in
- spell-checking mode if the user desires to make a small text edit along the way. The
- user should be able to enter and exit the mode with little or no effort.
- **Provide for flexible interaction.** Because different users have different interaction
- preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multitouch screen, or voice recognition commands. But every action is not amenable toevery interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.
- **Allow user interaction to be interruptible and undoable.** Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.
- **Streamline interaction as skill levels advance and allow the interaction to be customized.** Users often find that they perform the same sequence of interactions repeatedly.
- It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.
- **Hide technical internals from the casual user.** The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).
- **Design for direct interaction with objects that appear on the screen.** The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical

thing. For example, an application interface that allows a user to “stretch” an object (scale it in size) is an implementation of direct manipulation.

- **2)Reduce the User’s Memory Load**
- The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user’s memory. Whenever possible, the system should “remember” pertinent information and assist the user with an interaction scenario that assists recall.
- **Reduce demand on short-term memory.** When users are involved in complex
- tasks, the demand on short-term memory can be significant. The interface should be
- designed to reduce the requirement to remember past actions, inputs, and results.
- This can be accomplished by providing visual cues that enable a user to recognize
- past actions, rather than having to recall them.
- **Establish meaningful defaults.** The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a “reset” option should be available, enabling the redefinition of original default values.
- **Define shortcuts that are intuitive.** When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be
- tied to the action in a way that is easy to remember (e.g., first letter of the task to be
- invoked).
- **The visual layout of the interface should be based on a real-world**
- **metaphor.** For example, a bill payment system should use a checkbook and check
- register metaphor to guide the user through the bill paying process.
- **Disclose information in a progressive fashion.** The interface should be organized
- hierarchically. That is, information about a task, an object, or some behaviour should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick. An example, common to many word-processing applications, is the underlining function. The function itself is one of a number of functions under a *text style* menu. However, every underlining capability is not listed. The user must pick underlining; then all underlining options (e.g., single underline, double underline, dashed underline) are presented.
- **3)Make the Interface Consistent**
- The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are

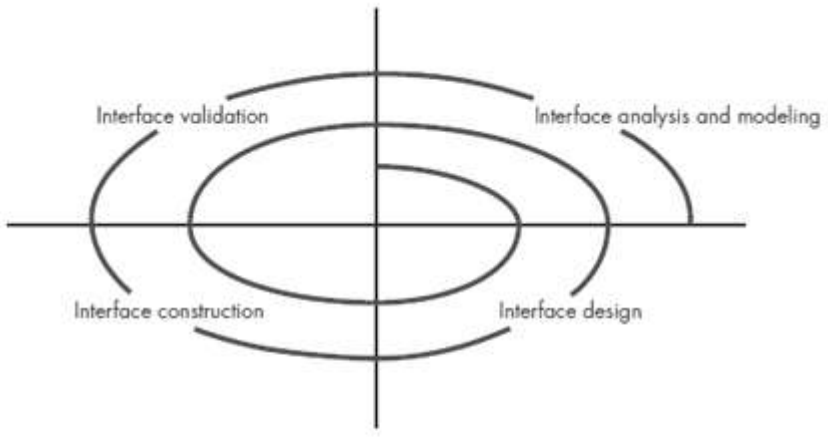
maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented.

- **Allow the user to put the current task into a meaningful context.** Many interfaces
- implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.
- **Maintain consistency across a family of applications.** A set of applications (or
- products) should all implement the same design rules so that consistency is maintained
- for all interaction.
- **If past interactive models have created user expectations, do not make**
- **changes unless there is a compelling reason to do so.** Once a particular interactive
- sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.
- **2) USER INTERFACE ANALYSIS AND DESIGN**
- The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.
-
- **Interface Analysis and Design Models**
- Four different models come into play when a user interface is to be analyzed and designed.
- A human engineer (or the software engineer) establishes a *user model,* the software engineer creates a *design model,* the end user develops a mental image that is often called the user's *mental model* or the *system perception*, and the implementers of the system create an *implementation model*. Unfortunately, each of these models may differ significantly. Your role, as an interface designer, is to reconcile these differences and derive a consistent representation of the interface.

- **NOTE:(The user model establishes the profile of end users of the system)**
- To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education,cultural or ethnic background, motivation, goals and personality".
- Users can be categorized as:
- **Novices.** No syntactic knowledge1 of the system and little semantic knowledge of the application or computer usage in general.
- **Knowledgeable, intermittent users.** Reasonable semantic knowledge of the application
- but relatively low recall of syntactic information necessary to use the interface.
- **Knowledgeable, frequent users.** Good semantic and syntactic knowledge that often
- leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.
- The user's *mental model* (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response.
- The *implementation model* combines the outward manifestation of the computerbased system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics.
- **The Process**
- The analysis and design process for user interfaces is iterative and can be represented using a spiral model,.The user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities
- (1) Interface analysis and modeling,
- (2) Interface design,
- (3) Interface construction, and
- (4) Interface validation.
- The spiral model diagram implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.
- ***Interface analysis*** focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new

system are recorded and different user categories are defined. For each user category requirements are elicited. In essence, you work to understand the system perception for each class of users.

- Once general requirements have been defined, a more detailed *task analysis* is
- conducted.



-
- Analysis of the user environment focuses on the physical work environment. Among the
- questions to be asked are
- • Where will the interface be located physically?
- • Will the user be sitting, standing, or performing other tasks unrelated to the
- interface?
- • Does the interface hardware accommodate space, light, or noise constraints?
- • Are there special human factors considerations driven by environmental factors?
- The information gathered as part of the analysis action is used to create an analysis
- model for the interface. The goal of *interface design* is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a
- manner that meets every usability goal defined for the system.
- *Interface construction* normally begins with the creation of a prototype that enables usage scenarios to be evaluated.
-
- *Interface validation* focuses on

- (1) The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements
- (2) The degree to which the interface is easy to use and easy to learn, and
- (3) The users acceptance of the interface as a useful tool in their work.
-
- **INTERFACE ANALYSIS**
- A key tenet of all software engineering process models is this: **understand the problem**
- **before you attempt to design a solution.**
- In the case of user interface design, understanding the problem means understanding
- (1) The people (end users) who will interact with the system through the interface
- (2) The tasks that end users must perform to do their work
- (3) The content that is presented as part of the interface, and
- (4) The environment in which these tasks will be conducted.
- **User Analysis**
- The phrase "user interface" is probably all the justification needed to spend some time understanding the user before worrying about technical matters.
- **User Interviews.** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.
- **Sales input.** Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.
- **Marketing input.** Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.
- **Support input.** Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.
- **The following set of questions will help you to better understand the users of a system:**
- • Are users trained professionals, technicians, clerical, or manufacturing

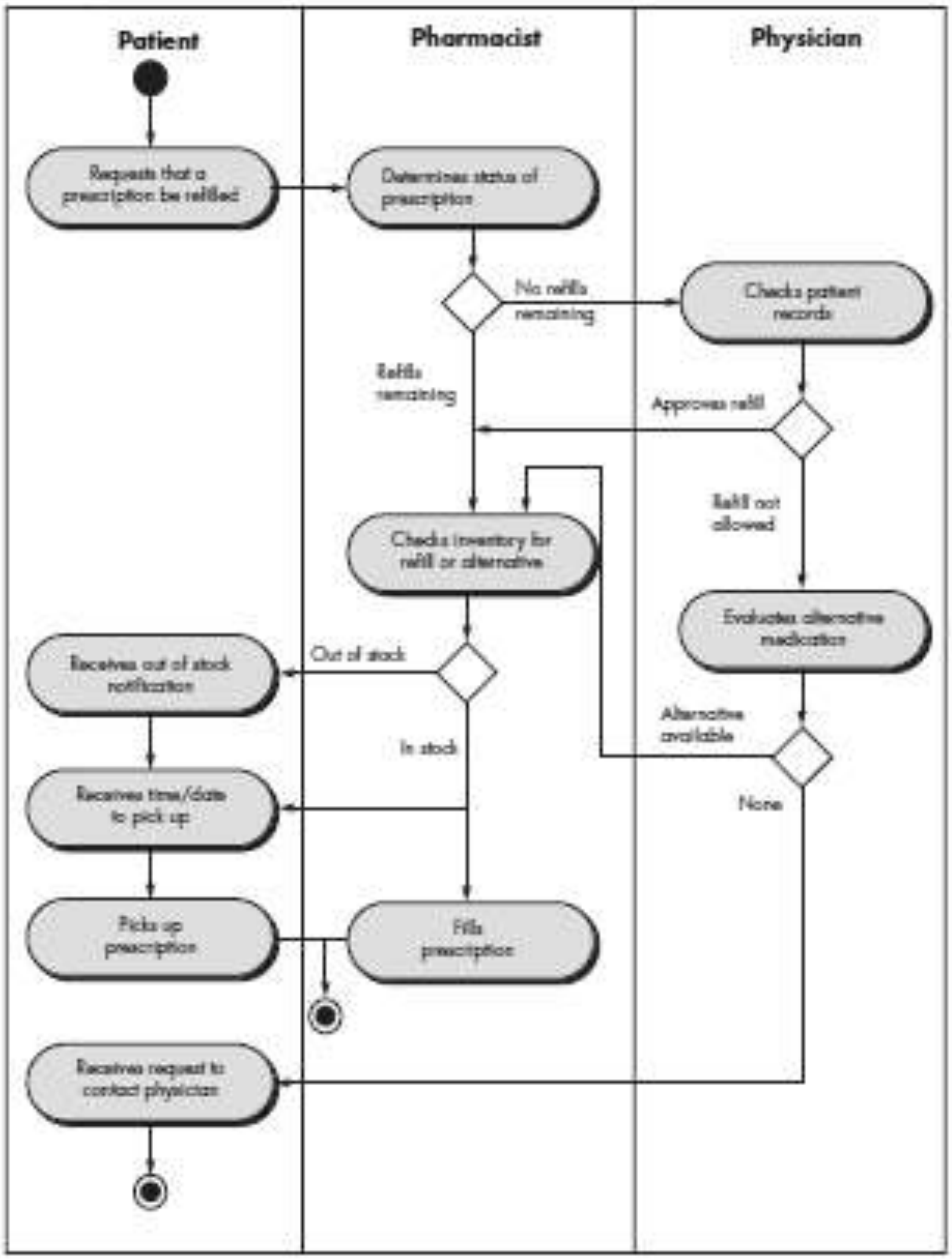
- workers?
- • What level of formal education does the average user have?
- • Are the users capable of learning from written materials or have they
- expressed a desire for classroom training?
- • Are users expert typists or keyboard phobic?
- • What is the age range of the user community?
- • Will the users be represented predominately by one gender?
- • How are users compensated for the work they perform?

- **Task Analysis and Modeling**
- The goal of task analysis is to answer the following questions
- • What work will the user perform in specific circumstances?
- • What tasks and subtasks will be performed as the user does the work?
- • What specific problem domain objects will the user manipulate as work is performed?
- • What is the sequence of work tasks

- **Use cases.**
- When used as part of task analysis, the use case is developed to show how an end user performs some specific work-related task.
- **Task elaboration.**
- Task analysis for interface design uses an elaborative approach to assist in understanding
- the human activities the user interface must accommodate.

- **Object elaboration.** Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes.
- **Workflow analysis.** When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply *workflow analysis.*
- **Hierarchical representation.** The hierarchy is derived by a stepwise elaboration of each task identified for the user.

FIGURE 11.2 Swimlane diagram for prescription refill function


Patient
Pharmacist
Physician
Requests that a prescription be refilled
Determines status of prescription
No refills remaining
Checks patient records
Refills remaining
Approves refill
Refill not allowed
Checks inventory for refill or alternative
Evaluates alternative medication
Out of stock
Receives out of stock notification
Alternative available
In stock
None
Receives time/date to pick up
Picks up prescription
Fills prescription
Receives request to contact physician

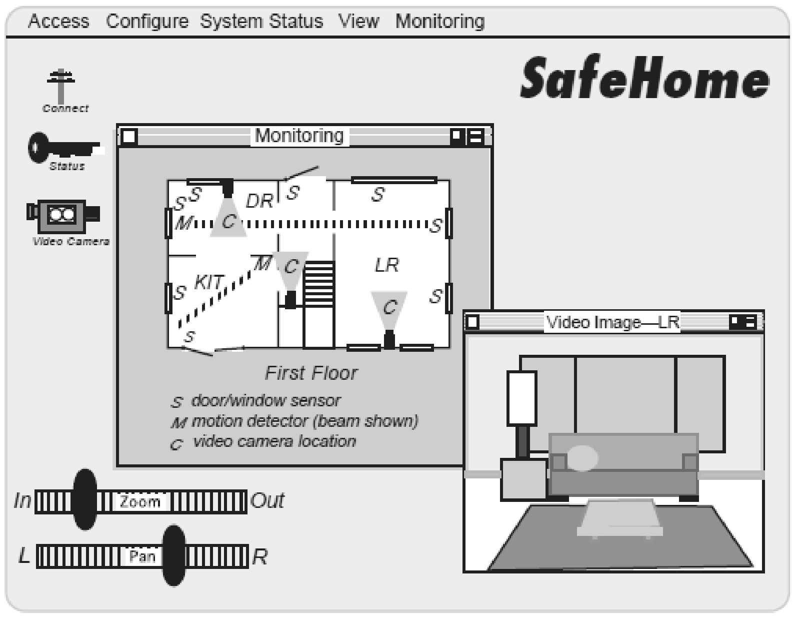
- **3)INTERFACE DESIGN STEPS**
- Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design, like all software engineering design, is an iterative process.
- Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.
- **1.** Using information developed during interface analysis (Section 11.3), define
- interface objects and actions (operations).
- **2.** Define events (user actions) that will cause the state of the user interface to
- change. Model this behavior.
- **3.** Depict each interface state as it will actually look to the end user.
- **4.** Indicate how the user interprets the state of the system from information provided
- through the interface.
- **Applying Interface Design Steps**
- The definition of interface objects and the actions that are applied to them is an important step in interface design.
- Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified.
- A *source object* (e.g., a report icon) is dragged and dropped onto a *target object* (e.g., a printer icon). The implication of this action is to create a hard-copy report.
- An *application object* represents application-specific data that are not directly manipulated
- as part of screen interaction.
- For example, a mailing list is used to store names for a mailing. The list itself might be sorted, merged, or purged (menu-based actions), but it is not dragged and dropped via user interaction.
- • *accesses* the *SafeHome* system
- • *enters* an **ID** and **password** to allow remote access
- • *checks* **system status**
- • *arms* or *disarms* *SafeHome* system
- • *displays* **floor plan** and **sensor locations**
- • *displays* **zones** on floor plan

- • *changes* **zones** on floor plan
- • *displays* **video camera locations** on floor plan
- • *selects* **video camera** for viewing
- • *views* **video images** (four frames per second)
- • *pans* or *zooms* the **video camera**

- **User Interface Design Patterns**
- Graphical user interfaces have become so common that a wide variety of user interface
- design patterns has emerged.

FIGURE 11.3

Preliminary screen layout



-

- **Design Issues**
- As the design of a user interface evolves, four common design issues
- **Response time.** System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.
- **Error handling.** Error messages and warnings are "bad news" delivered to users of interactive systems when something has gone awry.
- **Menu and command labeling.** The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type.

- **Application accessibility.** As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that
- enable easy access for those with special needs. *Accessibility* for users (and software
- engineers) who may be physically challenged is an imperative for ethical, legal, and
- business reasons.

- **Internationalization.** Software engineers and their managers invariably underestimate
- the effort and skills required to create user interfaces that accommodate the
- needs of different locales and languages.

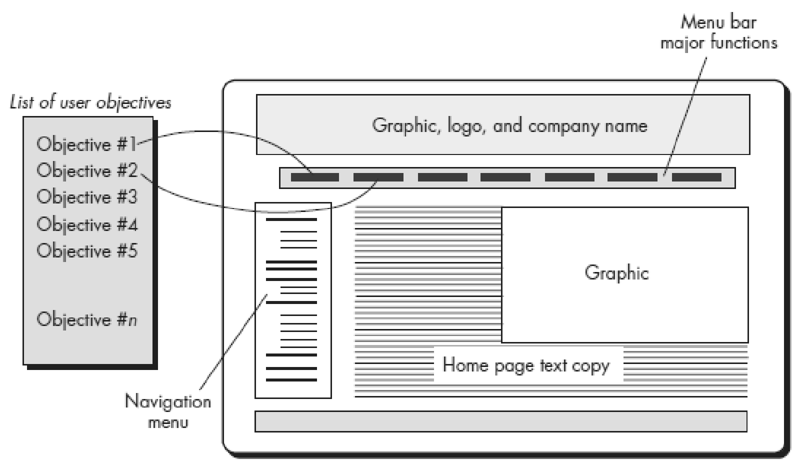
- **4)WEBAPP INTERFACE DESIGN**

- *Where am I?* The interface should (1) provide an indication of theWebApp that has been accessed and (2) inform the user of her location in the content hierarchy.
- *What can I do now?* The interface should always help the user understand his current options
- *Where have I been, where am I going?* The interface must facilitate navigation.
- **Anticipation.** *A WebApp should be designed so that it anticipates the user's next move.* For example, consider a customer support WebApp developed by a manufacturer of computer printers. A user has requested a content object that presents information about a printer driver for a newly released operating system. The designer of the WebApp should anticipate that the user might request a download of the driver and should provide navigation facilities that allow this to happen without requiring the user to search for this capability.
- **Communication.** *The interface should communicate the status of any activity initiated*
- *by the user.* Communication can be obvious (e.g., a text message) or subtle (e.g., an image of a sheet of paper moving through a printer to indicate that printing is under way).
- **Consistency.** *The use of navigation controls, menus, icons, and aesthetics (e.g., color,*
- *shape, layout) should be consistent throughout the WebApp.* For example, if underlined blue text implies a navigation link, content should never incorporate blue underlined text that does not imply a link

- **Controlled autonomy.** *The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that*
- *have been established for the application.*
- For example, navigation to secure portions of the WebApp should be controlled by userID and password, and there should be no navigation mechanism that enables a user to circumvent these controls.
- **Efficiency.** *The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the clientserver*
- **Flexibility.** *The interface should be flexible enough to enable some users to accomplish*
- *tasks directly and others to explore the WebApp in a somewhat random fashion*
- **Focus.** *The WebApp interface (and the content it presents) should stay focused on the*
- *user task(s) at hand.*
- **Fitt's law.** *"The time to acquire a target is a function of the distance to and size of the*
- *target"*
- **Human interface objects.** *A vast library of reusable human interface objects has been developed for WebApps.*
- **Latency reduction.** *Rather than making the user wait for some internal operation tocomplete (e.g., downloading a complex graphical image), the WebApp should use multitasking*
- *in a way that lets the user proceed with work as if the operation has been completed.*
- **Learnability.** *A WebApp interface should be designed to minimize learning time, and*
- *once learned, to minimize relearning required when the WebApp is revisited.*
- **Metaphors.** *An interface that uses an interaction metaphor is easier to learn and*
- *easier to use, as long as the metaphor is appropriate for the application and the user.*
- For example, an e-commerce site that implements automated bill paying for a financial institution,uses a checkbook metaphor (not surprisingly) to assist the user in specifying and
- scheduling bill payments.
- **Maintain work product integrity.** *A work product (e.g., a form completed by the*
- *user, a user-specified list) must be automatically saved so that it will not be lost if an error*

- *occurs.*
- **Readability.** *All information presented through the interface should be readable by*
- *young and old.*
- **Track state.** *When appropriate, the state of the user interaction should be tracked and*
- *stored so that a user can logoff and return later to pick up where she left off.* I
- **Visible navigation.** *A well-designed WebApp interface provides "the illusion that users*
- *are in the same place, with the work brought to them"*
-

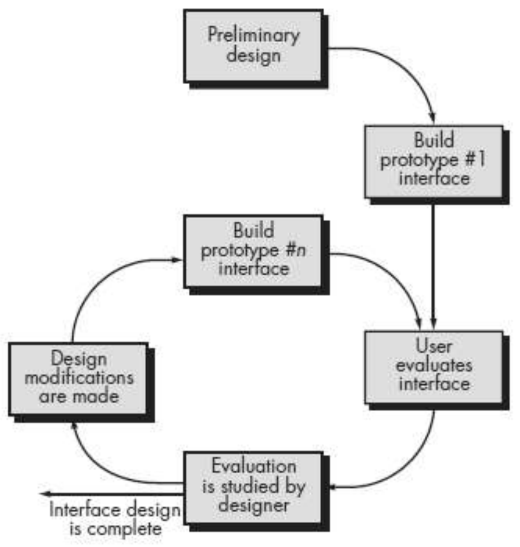
FIGURE 11.4 Mapping user objectives into interface actions



-
- **5)DESIGN EVALUATION**
- Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user.

FIGURE 11.5

The interface design evaluation cycle


Preliminary design
Build prototype #1 interface
Build prototype #n interface
User evaluates interface
Design modifications are made
Evaluation is studied by designer
Interface design is complete

-
- **1.** The length and complexity of the requirements model or written specification
- of the system and its interface provide an indication of the amount of learning
- required by users of the system.
- **2.** The number of user tasks specified and the average number of actions per
- task provide an indication of interaction time and the overall efficiency of the
- system.
- **3.** The number of actions, tasks, and system states indicated by the design model
- imply the memory load on users of the system.
- **4.** Interface style, help facilities, and error handling protocol provide a general
- indication of the complexity of the interface and the degree to which it will be
- accepted by the user.

UNIT-V TESTING & CASE

5

Acceptance Testing
Black Box Testing
White Box Testing
Unit Testing
System Testing
Integration Testing

Software testing is the process of evaluation a software item to detect differences between given input and expected output

TOPICS

Software Quality Assurance

Quality metrics

Software Reliability

Software testing

Path testing

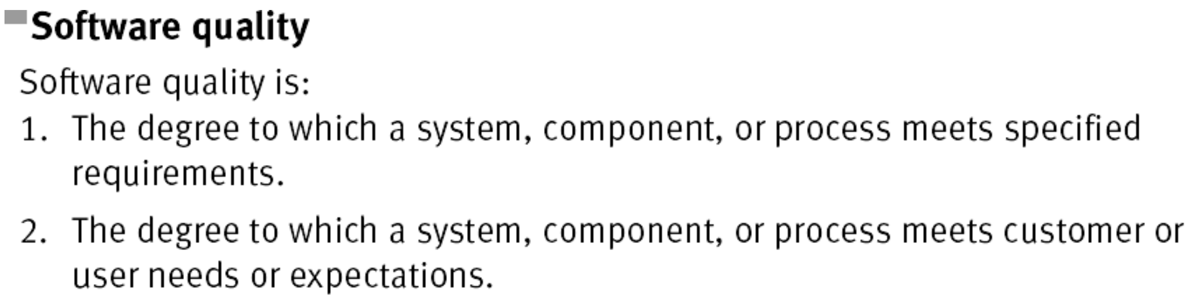
Control Structures testing

Black Box testing

Integration, Validation and System testing

Reverse Engineering and Reengineering

SOFTWARE QUALITY ASSURANCE



Software quality

Software quality is:

1. The degree to which a system, component, or process meets specified requirements.
2. The degree to which a system, component, or process meets customer or user needs or expectations.

Software quality assurance (SQA) is a process that ensures that developed **software** meets and complies with defined or standardized **quality** specifications. SQA is an ongoing process within the **software** development life cycle (SDLC) that routinely checks the developed **software** to ensure it meets desired **quality** measures.

Elements of SQA Standards:

- Reviews and Audits
- Testing
- Error/defect collection and analysis
- Change management
- Education
- Vendor management
- Security management
- Safety
- Risk management

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality.

The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

Reviews and audits. Technical reviews are a quality control activity performed by software engineers for software engineers Their intent is to uncover errors.

Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work

Testing. Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted

Error/defect collection and analysis. The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

Change management. Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion,and confusion almost always leads to poor quality.

Education. Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders.

Vendor management. Three categories of software are acquired from external software vendors

Security management. SQA ensures that appropriate process and technology are used to achieve software security.

Safety. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

Risk management. The SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

Q2)QUALITY METRICS

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality (and perform quality control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

SQA Tasks:

The Software Engineering Institute recommends a set of SQA actions that address quality assurance planning, oversight, record keeping, analysis,and reporting. These actions are performed by an independent SQA group that

Prepares an SQA plan for a project. The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance actions performed by the software engineering team and the SQA group are governed by the plan.

Participates in the development of the project's software process description. The software team selects a process for the work to be performed.

Reviews software engineering activities to verify compliance with thedefined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products identifies, documents and tracks deviations.Verifies that corrections have been made and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure.

Deviations may be encountered in the project plan

Records any noncompliance and reports to senior management.

Noncompliance items are tracked until they are resolved.

GOALS, ATTRIBUTES, AND METRICS

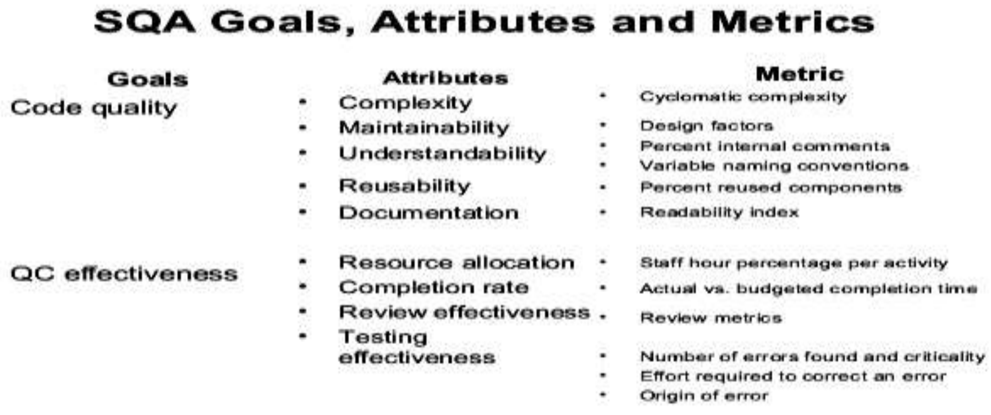
The SQA actions described in the preceding section are performed to achieve a set of pragmatic goals:

Requirements quality. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

Design quality. SQA looks for attributes of the design that are indicators of quality.

Code quality. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

Quality control effectiveness.. SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.



SQA Goals, Attributes and Metrics

Goals	Attributes	Metric
Code quality	• Complexity	• Cyclomatic complexity
	• Maintainability	• Design factors
	• Understandability	• Percent internal comments • Variable naming conventions
	• Reusability	• Percent reused components
	• Documentation	• Readability index
QC effectiveness	• Resource allocation	• Staff hour percentage per activity
	• Completion rate	• Actual vs. budgeted completion time
	• Review effectiveness	• Review metrics
	• Testing effectiveness	• Number of errors found and criticality • Effort required to correct an error • Origin of error

Q3)SOFTWARE RELIABILITY
Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer programin a specified environment for a specified time" .
Measures of Reliability and Availability
A simple measure of reliability is **meantime-between-failure** (MTBF):
MTBF = MTTF +MTTR
where MTTF = mean-time-to-failure
and MTTR = mean-time-to-failure and mean-time-to-repair

Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\%$$

Software Safety:
Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.For example, some of the hazards associated with a computer-based cruise control for an automobile might be:

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

Q4)A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted systematically.

Generic characteristics:

- To perform effective testing you should conduct effective technical reviews.By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects)an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification

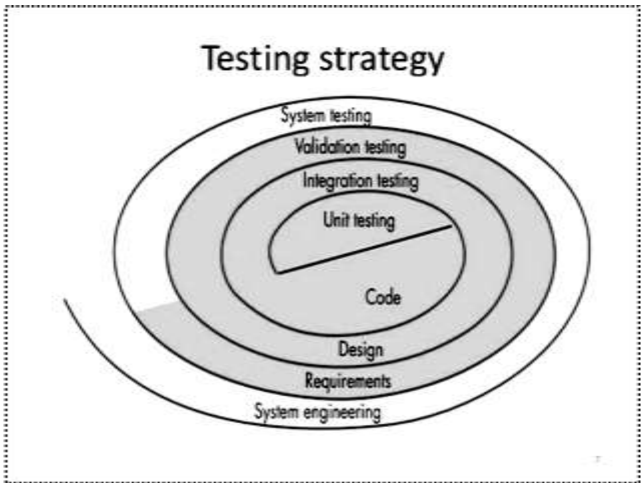
and validation (V&V).

Verification refers to the set of tasks that ensure that software correctly implements a specific function.

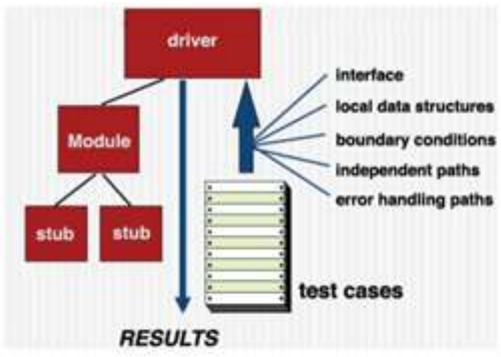
Verification: "Are we building the product right?"

Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer

Validation: "Are we building the right product?"



UNIT TESTING:A testing technique using which individual modules are tested to determine if there are any issues by the developer himself. It is concerned with functional correctness of the standalone modules.

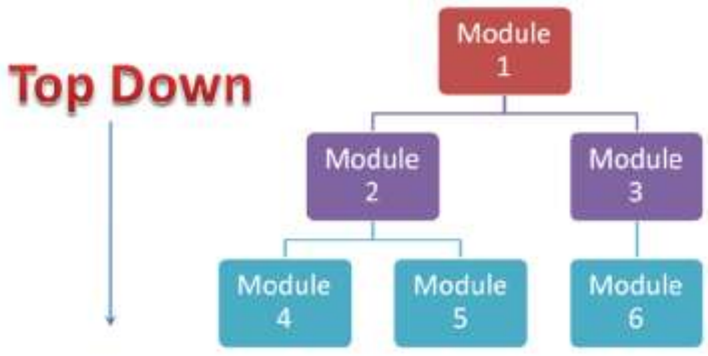


UNIT TEST

Q5)INTEGRATION TESTING

In **Integration Testing**, individual software modules are integrated logically and tested as a group. Integration testing focuses on checking data communication amongst these modules.

Top down Integration: In Top to down approach, testing takes place from top to down following the control flow of the software system.



Advantages:

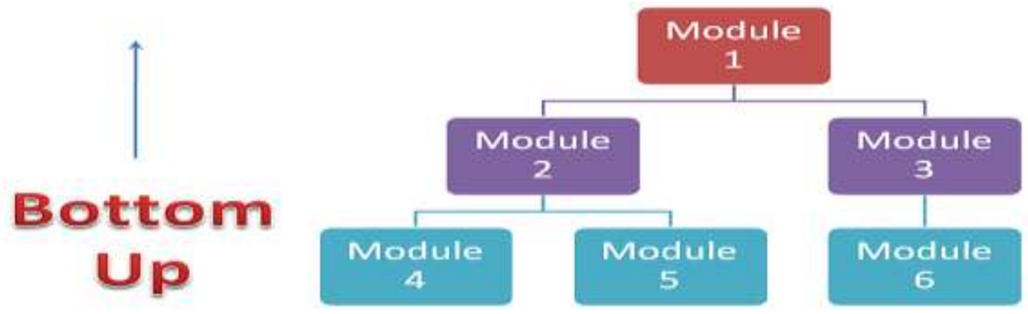
- Fault Localization is easier.
- Possibility to obtain an early prototype.
- Critical Modules are tested on priority; major design flaws could be found and fixed first.

Disadvantages:

- Needs many Stubs.
- Modules at lower level are tested inadequately.

Bottom up Integration

In the bottom up strategy, each module at lower levels is tested with higher modules until all modules are tested. It takes help of Drivers for testing



Advantages:

- Fault localization is easier.
- No time is wasted waiting for all modules to be developed unlike Big-bang approach

Disadvantages:

- Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.
- Early prototype is not possible

WHAT IS SMOKE TESTING?

Smoke testing is the initial testing process exercised to check whether the software under test is ready/stable for further testing.

WHAT IS REGRESSION TESTING?

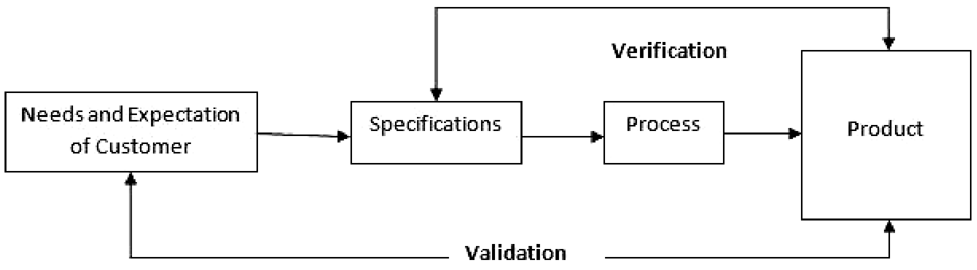
Regression testing a black box testing technique that consists of re-executing those tests that are impacted by the code changes. These tests should be executed as often as possible throughout the software development life cycle.

Q6)VALIDATION TESTING

The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfils its intended use when deployed on appropriate environment.

After each validation test case has been conducted, one of two possible conditions exists:
(1) The function or performance characteristic conforms to specification and is accepted or
(2) A deviation from specification is uncovered and a deficiency list is created.



Alpha Testing is a type of testing conducted by a team of highly skilled testers at development site

Beta Testing is done by customers or end users at their own site.

Q7)What is System Testing?

System testing is the type of testing to check the behaviour of a complete and fully mointegrated software product based on the software requirements specification (SRS) document. ***The main focus of this testing is to evaluate Business / Functional / End-user requirements.***

Types of System Tests:
Recovery Testing: *Recovery testing* is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

Security Testing: *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

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

Performance Testing: Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

Deployment testing: In many cases, software must execute on a variety of platforms and under more than one operating system environment. ***Deployment testing***, sometimes called *configuration testing*, exercises the software in each environment in which it is to operate.

Q8)SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error.

Testability. "*Software testability* is simply how easily [a computer program] can be tested." The following characteristics lead to testable software.

Operability. "The better it works, the more efficiently it can be tested."

Observability. "What you see is what you test.".

Controllability. "The better we can control the software, the more the testing can be automated and optimized."

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

Simplicity. "The less there is to test, the more quickly we can test it."

The program should exhibit *functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements);

structural simplicity (e.g., architecture is modularized to limit the propagation of faults)

code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability. "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests.

Understandability. "The more information we have, the smarter we will test."

Test Characteristics.

A good test has a high probability of finding an error.

A good test is not redundant.

A good test should be "best of breed" .

A good test should be neither too simple nor too complex.

Q9)WHITE-BOX TESTING

White-box testing, sometimes called *glass-box testing*, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.

(1) Guarantee that all independent paths within a module have been exercised at least once

(2) Exercise all logical decisions on their true and false sides

(3) Execute all loops at their boundaries and within their operational bounds

(4) Exercise internal data structures to ensure their validity.

Q10)What is Basis Path Testing?

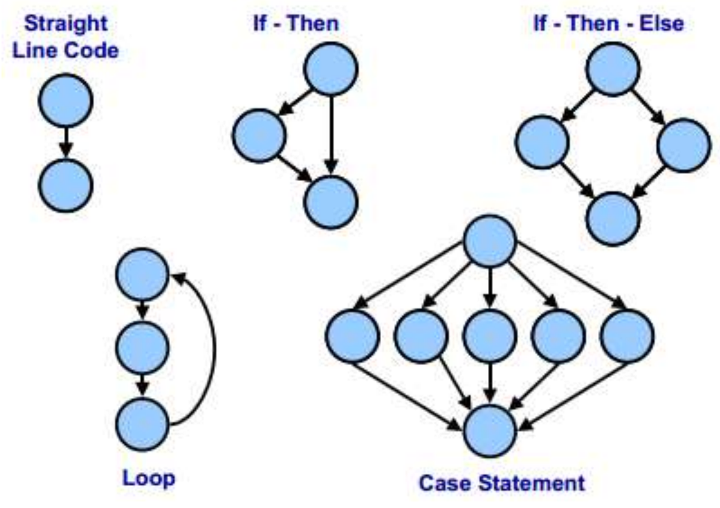
The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed

The principle behind basis path testing is that all independent paths of the program have to be tested at least once. **Below are the steps of this technique**:

- Draw a control flow graph.
- Determine Cyclomatic complexity.
- Find a basis set of paths.
- Generate test cases for each path.

Step 1: Draw a control flow graph

Basic control flow graph structures:

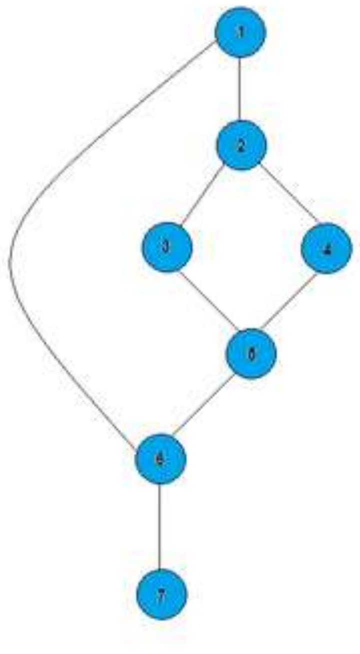


On a control flow graph, we can see that:

- Arrows or edges represent flows of control.
- Circles or nodes represent actions.
- Areas bounded by edges and nodes are called regions.
- A predicate node is a node containing a condition.

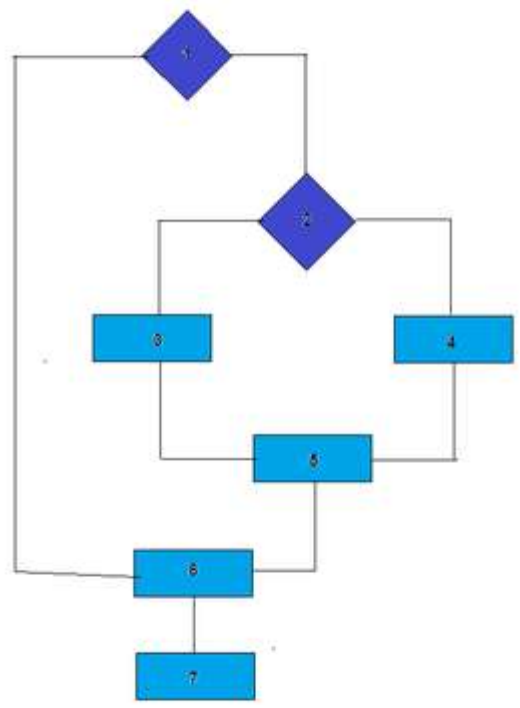
Below is an example of control flow graph:

```
1:   IF   A   =   100

2: THEN IF B > C

3:  THEN  A  =  B

4:   ELSE   A=   C

5:              ENDIF

6:              ENDIF

7: Print A
```



Cyclomatic complexity= Number of Predicate Nodes + 1

From the example in **Step 1,** we can redraw it as below to show predicate nodes clearly:



As we see, there are two predicate nodes in the graph.

So the Cyclomatic complexity is 2+1= 3.

Cyclomatic complexity V(G) for a flow graph G is defined as

$$V(G) = E - N + 2$$

where E is the number of flow graph edges and N is the number of flow graph nodes.

3. Cyclomatic complexity V(G) for a flow graph G is also defined as

$$V(G) = P + 1$$

where P is the number of predicate nodes contained in the flow graph G.

Step 3: Find a basis set of paths

The Cyclomatic complexity tells us the number of paths to evaluate for basis path testing. In the example, we have 3 paths, and our basis set of paths is:

Path 1: 1, 2, 3, 5, 6, 7.

Path 2: 1, 2, 4, 5, 6, 7.

Path 3: 1, 6, 7.

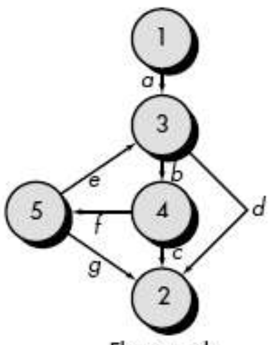
Step 4: Generate test cases for each path

After determining the basis set of path, we can generate the test case for each path. Usually we need at least one test case to cover one path. In the example, however, Path 3 is already covered by Path 1 and 2 so we only need to write 2 test cases.

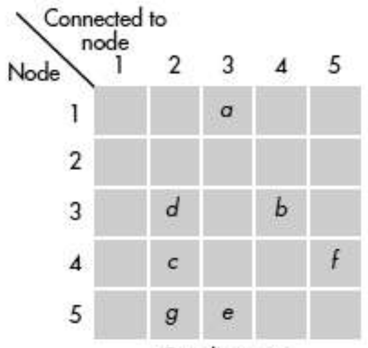
In conclusion, basis path testing helps us to reduce redundant tests. It suggests independent paths from which we write test cases needed to ensure that every statement and condition can be executed at least one time.

Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. A data structure, called a *graph matrix*, can be quite useful for developing a software tool that assists in basis path testing.



Flow graph



Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Q11)CONTROL STRUCTURE TESTING

Condition Testing

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

$$E_1 \text{<relational-operator>} E_2$$

where *E*1 and *E*2 are arithmetic expressions and <relational-operator> is one of the following: $<, \leq, =, \neq$ (nonequality), $>$, or $\geq$.

A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses.

Data Flow Testing

The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.

$\text{DEF}(S) = \{X \mid \text{statement } S \text{ contains a definition of } X\}$

$\text{USE}(S) = \{X \mid \text{statement } S \text{ contains a use of } X\}$

Loop Testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs

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

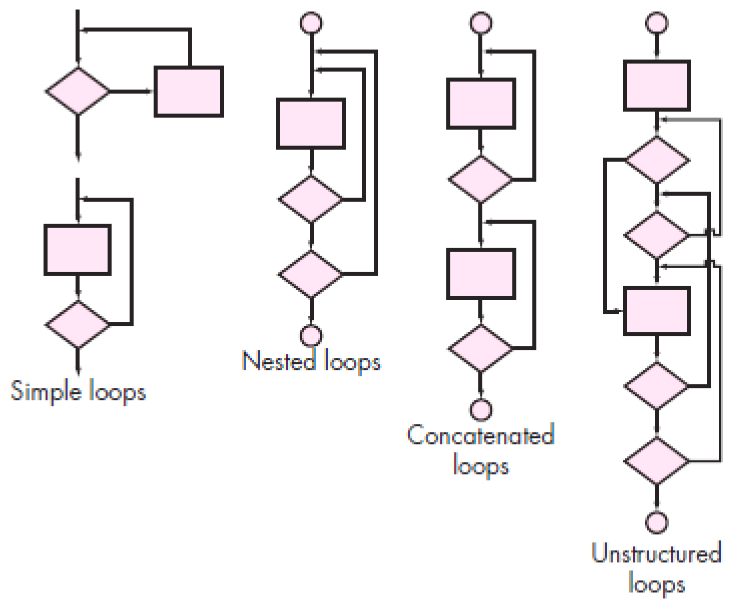
1. Skip the loop entirely.

2. Only one pass through the loop.

3. Two passes through the loop.

4. *m* passes through the loop where $m < n$.

5. *n* -1, *n*, *n* + 1 passes through the loop.



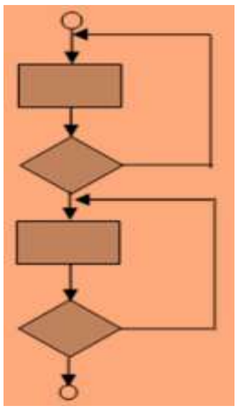
For nested loop, you need to follow the following steps.

1. Set all the other loops to minimum value and start at the innermost loop
2. For the innermost loop, perform a simple loop test and hold the outer loops at their minimum iteration parameter value

3. Perform test for the next loop and work outward.
4. Continue until the outermost loop has been tested.

Concatenated Loops:

In the concatenated loops, if two loops are independent of each other then they are tested using simple loops or else test them as nested loops.



Unstructured Loops For unstructured loops, it requires restructuring of the design to reflect the use of the structured programming constructs.

Q12)BLACK-BOX TESTING

Black-box testing, also called *behavioral testing,* focuses on the functional requirements of the software.

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

(1) Incorrect or missing functions,

(2) Interface errors

(3) Errors in data structures or external database access

(4) Behavior or performance errors

(5) Initialization and

(6)termination errors.

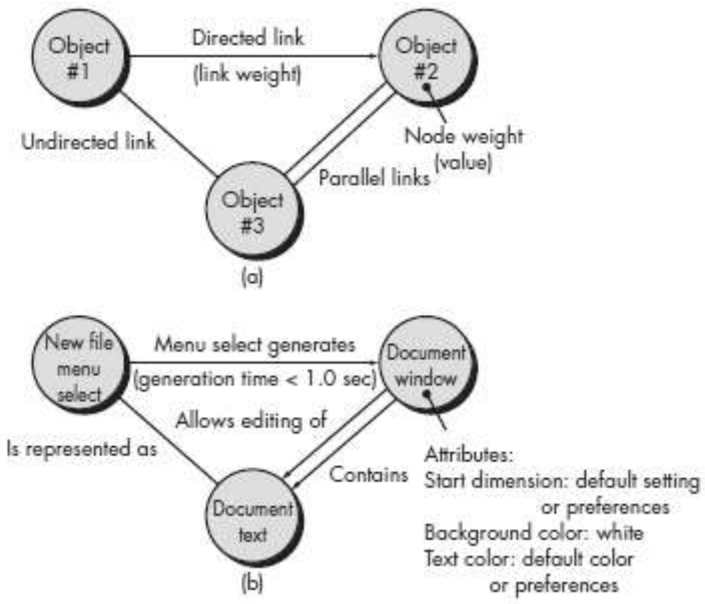
Tests are designed to answer the following questions:

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

- What effect will specific combinations of data have on system operation?

Graph-Based Testing Methods

Software testing begins by creating a graph of important objects and their relationships andthen devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.



A **directed link** (represented by an arrow) indicates that a relationship moves in only one direction.

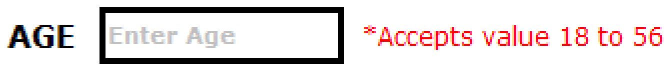
A **bidirectional link**, also called a *symmetric link*, implies that the relationship applies in both directions.

Parallel links are used when a number of different relationships are established between graph nodes.

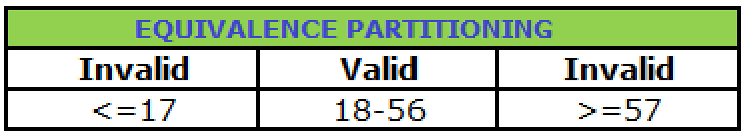
Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.

Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition.



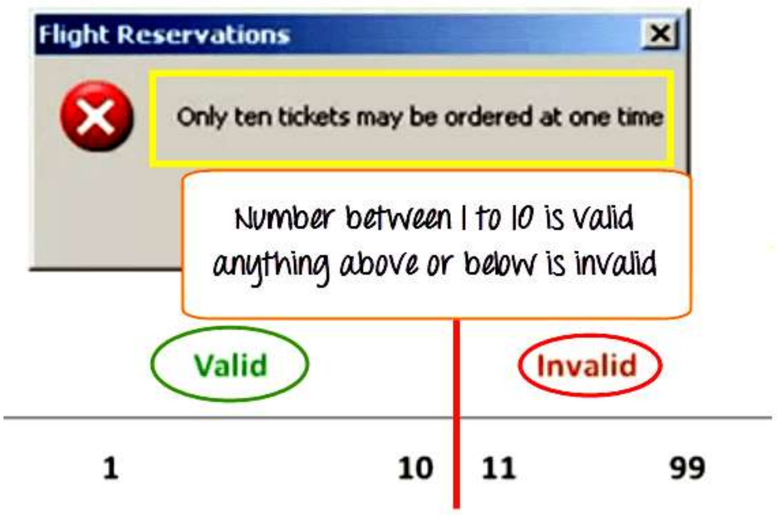
AGE Enter Age *Accepts value 18 to 56



EQUIVALENCE PARTITIONING		
Invalid	**Valid**	**Invalid**
<=17	18-56	>=57

Boundary Value Analysis

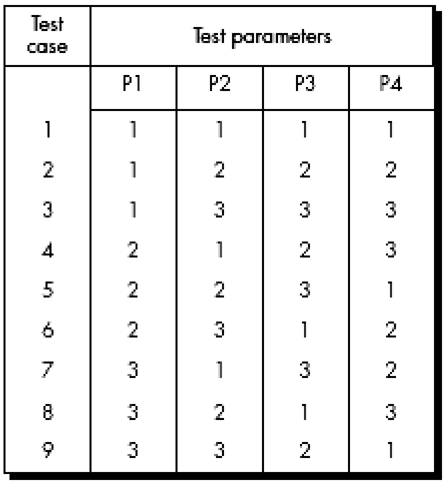
A greater number of errors occurs at the boundaries of the input domain rather than in the "canter." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test Cases that exercise bounding values.


Flight Reservations
Only ten tickets may be ordered at one time
Number between 1 to 10 is valid anything above or below is invalid
Valid
Invalid
1
10
11
99

Orthogonal array testing:

Orthogonal array testing is a black box **testing** technique that is a systematic, statistical way of software **testing**. It is used when the number of inputs to the system is relatively small, but too large to allow for exhaustive **testing** of every possible input to the systems.

The orthogonal array testing method is particularly useful in finding *region faults*—an error category associated with faulty logic within a software component.



Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Q13)SOFTWARE RE-ENGINEERING

Reorganising and modifying existing software systems to make them more maintainable

- ➢ Re-structuring or re-writing part or all of a legacy system without changing its functionality
- ➢ Applicable where some but not all sub-systems of a larger system require frequent maintenance
- ➢ Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented

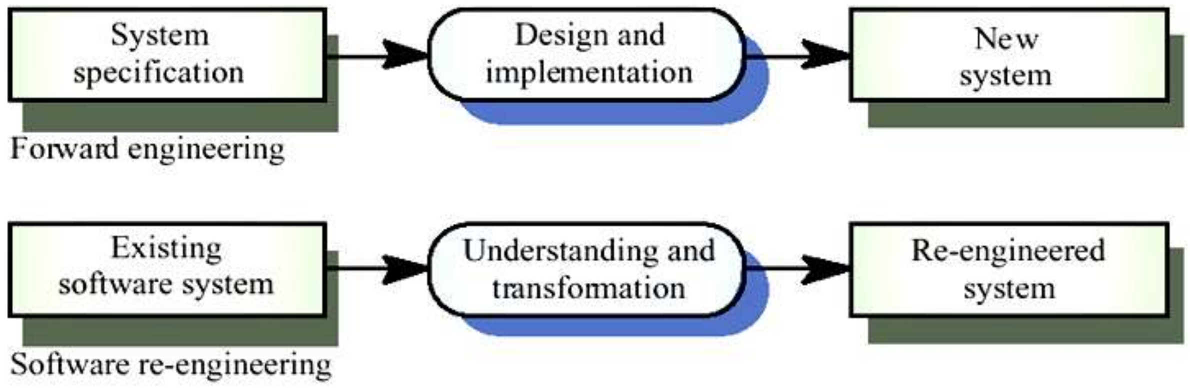
When to re-engineer

- ✓ When system changes are mostly confined to part of the system then re-engineer that part
- ✓ When hardware or software support becomes obsolete
- ✓ When tools to support re-structuring are available

Re-engineering advantages:

Reduced risk: There is a high risk in new software development. There may be development problems, staffing problems and specification problems

Reduced cost: The cost of re-engineering is often significantly less than the costs of developing new software



Re-engineering cost factors:

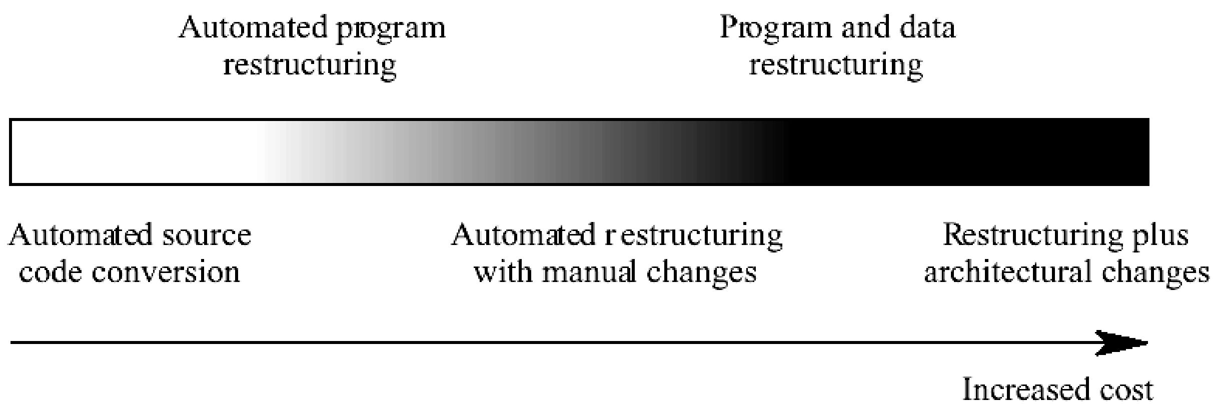
The quality of the software to be re-engineered

The tool support available for re-engineering

The extent of the data conversion which is required

The availability of expert staff for re-engineering

Re-engineering approaches:


Automated program restructuring
Program and data restructuring
Automated source code conversion
Automated restructuring with manual changes
Restructuring plus architectural changes
Increased cost

Source code translation:

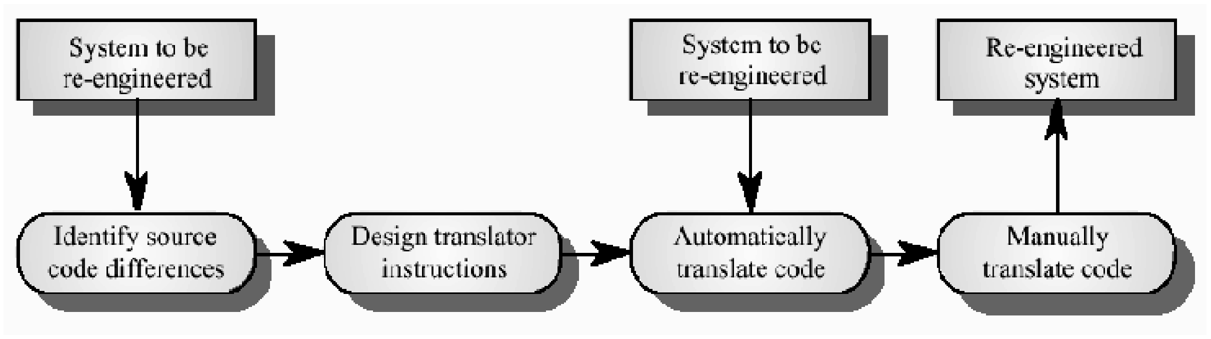
Involves converting the code from one language (or language version) to another e.g. FORTRAN to C

May be necessary because of:

- Hardware platform update
- Staff skill shortages
- Organisational policy changes

-Only realistic if an automatic translator is available

The program translation process:

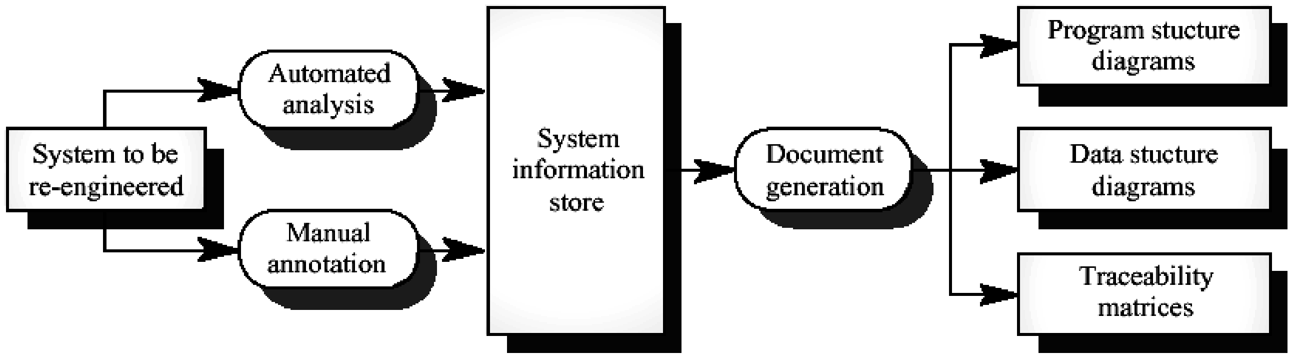

System to be re-engineered
System to be re-engineered
Re-engineered system
Identify source code differences
Design translator instructions
Automatically translate code
Manually translate code

Q14)Reverse engineering

- ✓ Reverse engineering is the process of deriving the system design and specification from its source code
- ✓ Analysing software with a view to understanding its design and specification
- ✓ May be part of a re-engineering process but may also be used to re-specify a system for re-implementation
- ✓ Builds a program data base and generates information from this

- ✓ Program understanding tools (browsers, cross-reference generators, etc.) may be used in this process

The reverse engineering process



Program structure improvement:

- ✷ The program may be automatically restructured to remove unconditional branches
- ✷ Conditions may be simplified to make them more readable

Restructuring problems:

Problems with re-structuring are:

- Loss of comments
- Loss of documentation and Heavy computational demands

- ✷ Restructuring doesn't help with poor modularisation where related components are dispersed throughout the code
- ✷ The understandability of data-driven programs may not be improved by re-structuring

Q15) CASE stands for **C**omputer **A**ided **S**oftware **E**ngineering.

It means, development and maintenance of software projects with help of various automated software tools.

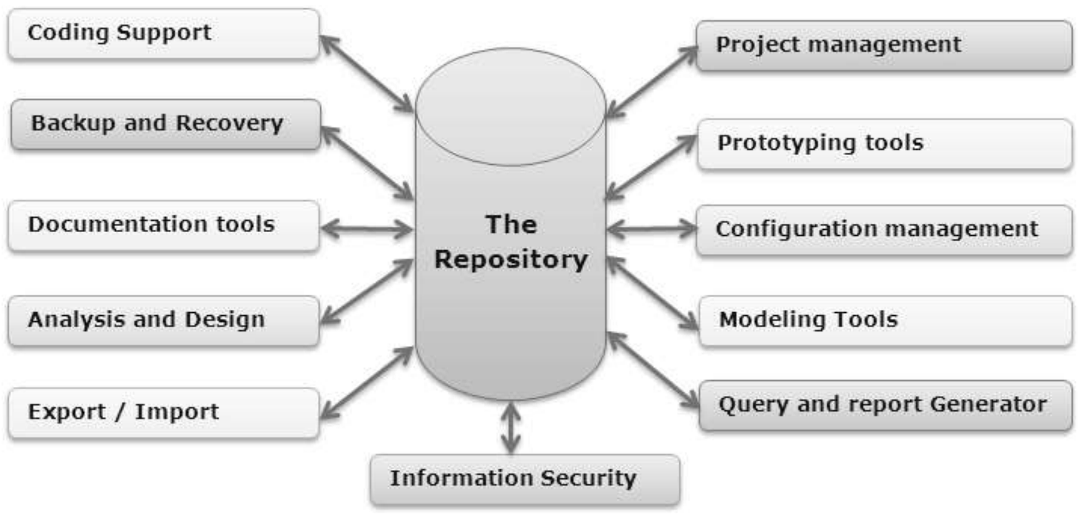
CASE Tools: CASE tools are set of software application programs, which are used to automate SDLC activities. CASE tools are used by software project managers, analysts and engineers to develop software system.

There are number of CASE tools available to simplify various stages of Software Development Life Cycle such as Analysis tools, Design tools, Project management tools, Database Management tools, Documentation tools are to name a few.

Use of CASE tools accelerates the development of project to produce desired result and helps to uncover flaws before moving ahead with next stage in software development

Components of CASE Tools

CASE tools can be broadly divided into the following parts based on their use at a particular SDLC stage:CASE stands for **C**omputer **A**ided **S**oftware **E**ngineering. It means, development and maintenance of software projects with help of various automated software tools.



CASE Tools

Components of CASE Tools

CASE tools can be broadly divided into the following parts based on their use at a particular SDLC
stage:

Central Repository - CASE tools require a central repository, which can serve as a source of common, integrated and consistent information. Central repository is a central place of storage where product specifications, requirement documents, related reports and diagrams, other useful information regarding management is stored. Central repository also serves as data dictionary.

Upper Case Tools - Upper CASE tools are used in planning, analysis and design stages of SDLC.

Lower Case Tools - Lower CASE tools are used in implementation, testing and maintenance.

Integrated Case Tools - Integrated CASE tools are helpful in all the stages of SDLC, from Requirement gathering to Testing and documentation.

CASE tools can be grouped together if they have similar functionality, process activities and capability of getting integrated with other tools.

Scope of Case Tools

The scope of CASE tools goes throughout the SDLC.

Case Tools Types

Now we briefly go through various CASE tools

Diagram tools

These tools are used to represent system components, data and control flow among various software components and system structure in a graphical form. For example, Flow Chart Maker
tool for creating state-of-the-art flowcharts.

Process Modeling Tools

Process modeling is method to create software process model, which is used to develop the software. Process modeling tools help the managers to choose a process model or modify it as per
the requirement of software product. For example, EPF Composer

Project Management Tools

These tools are used for project planning, cost and effort estimation, project scheduling and resource planning. Managers have to strictly comply project execution with every mentioned step in software project management. Project management tools help in storing and sharing project information in real-time throughout the organization.

Documentation Tools

Documentation in a software project starts prior to the software process, goes throughout all phases of SDLC and after the completion of the project.

Documentation tools generate documents for technical users and end users. Technical users are mostly in-house professionals of the development team who refer to system manual, referencemanual, training manual, installation manuals etc. The end user documents describe the functioning and how-to of the system such as user manual. For example, Doxygen, DrExplain,Adobe RoboHelp for documentation.

Analysis Tools

These tools help to gather requirements, automatically check for any inconsistency, inaccuracy in the diagrams, data redundancies or erroneous omissions. For example, Accept 360, Accompa,CaseComplete for requirement analysis, Visible Analyst for total analysis.

Design Tools

These tools help software designers to design the block structure of the software, which may further be broken down in smaller modules using refinement techniques. These tools provides detailing of each module and interconnections among modules. For example, Animated Software Design

Configuration Management Tools

An instance of software is released under one version. Configuration Management tools deal with –

Version and revision management

Baseline configuration management

Change control management

CASE tools help in this by automatic tracking, version management and release management. For example, Fossil, Git, Accu REV.

Change Control Tools

These tools are considered as a part of configuration management tools. They deal with changes made to the software after its baseline is fixed or when the software is first released. CASE tools automate change tracking, file management, code management and more. It also helps in enforcing change policy of the organization.

Programming Tools

These tools consist of programming environments like IDE **IntegratedDevelopmentEnvironment,** in-built modules library and simulation tools. These tools provide comprehensive aid in building software product and include features for simulation and testing. For example, Cscope to search code in C,Eclipse.

Prototyping Tools

Software prototype is simulated version of the intended software product. Prototype provides initial look and feel of the product and simulates few aspect of actual product.

Web Development Tools

These tools assist in designing web pages with all allied elements like forms, text, script, graphic and so on. Web tools also provide live preview of what is being developed and how will it look after completion. For example, Fontello, Adobe Edge Inspect, Foundation 3, Brackets.

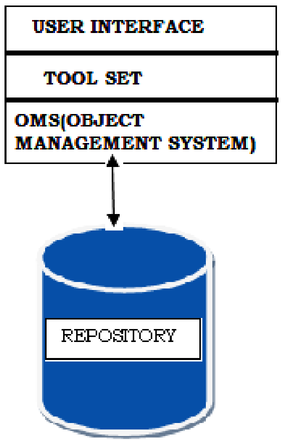
Quality Assurance Tools

Quality assurance in a software organization is monitoring the engineering process and methods adopted to develop the software product in order to ensure conformance of quality as per organization standards. QA tools consist of configuration and change control tools and software testing tools. For example, SoapTest, AppsWatch, JMeter.

Maintenance Tools

Software maintenance includes modifications in the software product after it is delivered. Automatic logging and error reporting techniques, automatic error ticket generation and root cause Analysis are few CASE tools, which help software organization in maintenance phase of SDLC.

Case Environment:



User interface

The user interface provides for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

Object management system and repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records.