UNIT I

SOFTWARE PRODUCT AND PROCESS

1.0 INTRODUCTION

SOFTWARE:

Software is a set of instruction used to perform a specific task.

ENGINEERING:

It comprises analysis,design,construction,verification and management of technical entities.

SOFTWARE ENGINEERING:

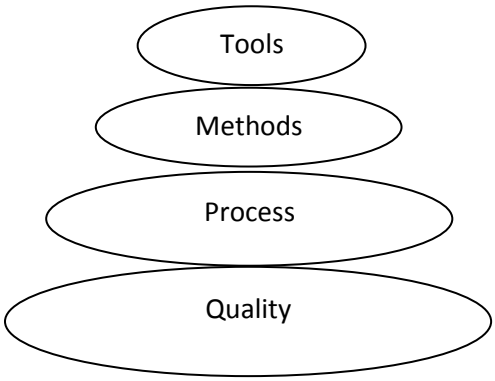
IEEE: International Electrical and Electronic Engineer

It is a systematic,disciplined,quantifiable approach for the development,operation,maintenance of software.

SOFTWARE ENGINEERING PARADIGM:

A combination of software engineering layers and generic view of software engineering is called Software Engineering Paradigm.

SOFTWARE ENGINEERING LAYERS:



SOFTWARE VIEW OF SOFTWARE ENGINEERING:

1.Analysis

2.Design

3.Implementation

4.Testing

5.Maintenance

SOFTWARE PROCESS MODELS:

- Waterfall life cycle model
- RAD model
- Prototype model
- Spiral model
- Incremental model
- Object oriented model
- Winwin spiral model

1.1 S/W Engineering Paradigm

The term "software engineering" was coined in about 1969 to mean "the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines".

This view opposed uniqueness and "magic" of programming in an effort to move the development of software from "magic" (which only a select few can do) to "art" (which the talented can do) to "science" (which supposedly anyone can do!). There have been numerous definitions given for software engineering (including that above and below).

Software Engineering is not a discipline; it is an aspiration, as yet unachieved. Many approaches have been proposed including reusable components, formal methods, structured methods and architectural studies. These approaches chiefly emphasize the engineering product; the solution rather than the problem it solves.

Software Development current situation:

- People developing systems were consistently wrong in their estimates of time, effort, and costs
- Reliability and maintainability were difficult to achieve
- Delivered systems frequently did not work
    - 1979 study of a small number of government projects showed that:
    - 2% worked
    - 3% could work after some corrections
    - 45% delivered but never successfully used
    - 20% used but extensively reworked or abandoned
    - 30% paid and undelivered
- Fixing bugs in delivered software produced more bugs
- Increase in size of software systems
    - NASA
    - StarWars Defense Initiative
    - Social Security Administration
    - financial transaction systems
- Changes in the ratio of hardware to software costs
    - early 60's - 80% hardware costs
    - middle 60's - 40-50% software costs
    - today - less than 20% hardware costs
- Increasingly important role of maintenance
    - Fixing errors, modification, adding options
    - Cost is often twice that of developing the software
- Advances in hardware (lower costs)
- Advances in software techniques (e.g., users interaction)
- Increased demands for software
    - Medicine, Manufacturing, Entertainment, Publishing
- Demand for larger and more complex software systems
    - Airplanes (crashes), NASA (aborted space shuttle launches),

- "ghost" trains, runaway missiles,
- ATM machines (have you had your card "swallowed"?), life-support systems, car systems, etc.
- US National security and day-to-day operations are highly dependent on computerized systems.

Manufacturing software can be characterized by a series of steps ranging from concept exploration to final retirement; this series of steps is generally referred to as a *software lifecycle*.

Steps or phases in a software lifecycle fall generally into these categories:

- Requirements (Relative Cost 2%)
- Specification (analysis) (Relative Cost 5%)
- Design (Relative Cost 6%)
- Implementation (Relative Cost 5%)
- Testing (Relative Cost 7%)
- Integration (Relative Cost 8%)
- Maintenance (Relative Cost 67%)
- Retirement

Software engineering employs a variety of methods, tools, and paradigms.

Paradigms refer to particular approaches or philosophies for designing, building and maintaining software. Different paradigms each have their own advantages and disadvantages which make one more appropriate in a given situation than perhaps another (!).

A method (also referred to as a technique) is heavily depended on a selected paradigm and may be seen as a procedure for producing some result. Methods generally involve some formal notation and process(es).

Tools are automated systems implementing a particular method.

Thus, the following phases are heavily affected by selected software paradigms

- Design
- Implementation
- Integration
- Maintenance

The software development cycle involves the activities in the production of a software system. Generally the software development cycle can be divided into the following phases:

- Requirements analysis and specification
- Design
  - Preliminary design
  - Detailed design
- Implementation
  - Component Implementation
  - Component Integration
  - System Documenting
- Testing
  - Unit testing
  - Integration testing
  - System testing
- Installation and Acceptance Testing
- Maintenance
  - Bug Reporting and Fixing
  - Change requirements and software upgrading

Software lifecycles that will be briefly reviewed include:

- Build and Fix model
- Waterfall and Modified Waterfall models
- Rapid Prototyping
- Boehm's spiral model

1.2 VERIFICATION VS VALIDATION

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

The V & V process

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
    - The discovery of defects in a system
    - The assessment of whether or not the system is usable in an operational situation.

Static and dynamic verification

- Software inspections Concerned with analysis of the static system representation to discover problems (static verification)
    - May be supplement by tool-based document and code analysis
- Software testing Concerned with exercising and observing product behaviour (dynamic verification)
    - The system is executed with test data and its operational behaviour is observed

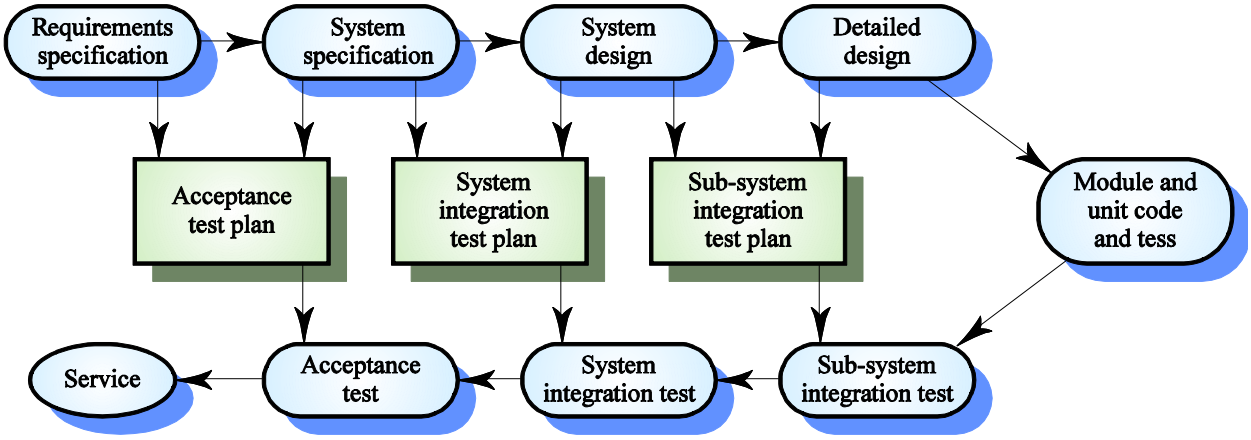
V& V goals

- Verification and validation should establish confidence that the software is fit for purpose
- This does NOT mean completely free of defects
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed

V & V planning

- Careful planning is required to get the most out of testing and inspection processes
- Planning should start early in the development process
- The plan should identify the balance between static verification and testing
- Test planning is about defining standards for the testing process rather than describing product tests

The V-model of development



Software validation

- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking and review processes and system testing.
- System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.

1.3 Life Cycle models

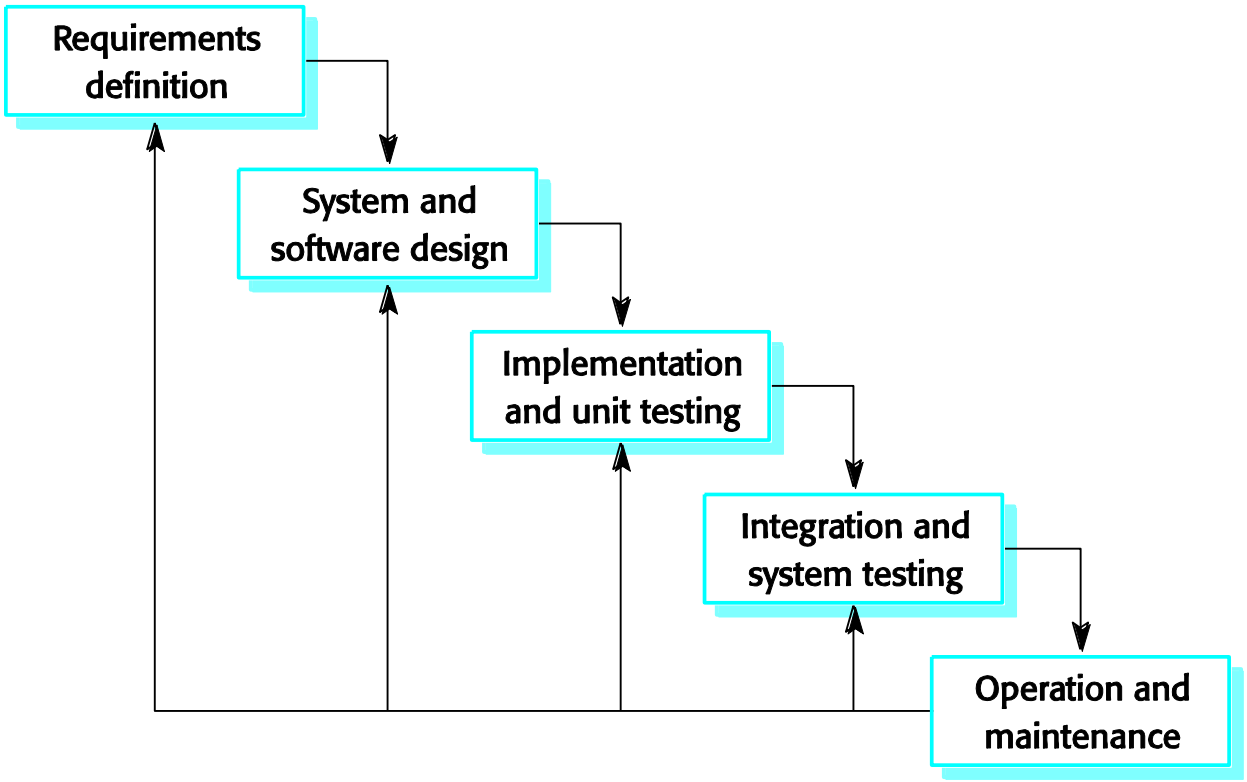
- The waterfall model
  - Separate and distinct phases of specification and development.
- Evolutionary development
  - Specification, development and validation are interleaved.
- Component-based software engineering
  - The system is assembled from existing components.
- There are many variants of these models e.g. formal development where a waterfall-like process is used but the specification is a formal specification that is refined through several stages to an implementable design.

Waterfall model phases

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance

- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. One phase has to be complete before moving onto the next phase.

Waterfall model



Waterfall model problems

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
- Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites**.**

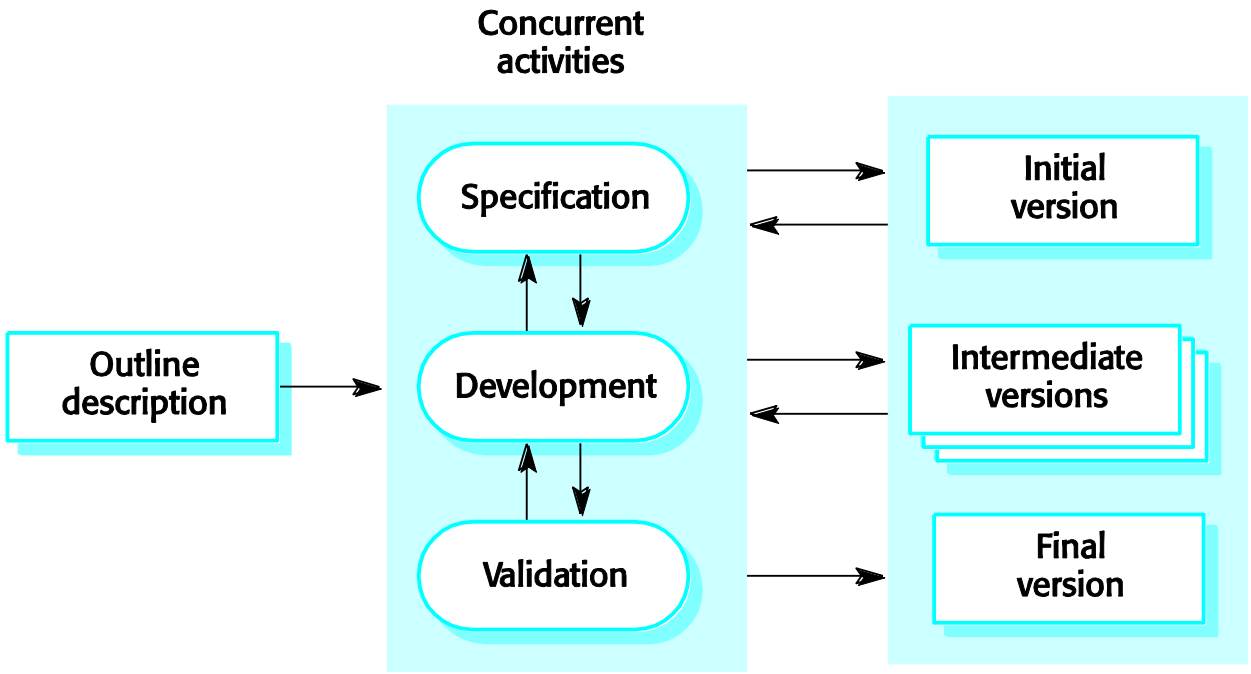
Evolutionary development

- Exploratory development

- Objective is to work with customers and to evolve a final system from an initial outline specification. Should start with well-understood requirements and add new features as proposed by the customer.

- Throw-away prototyping
  - Objective is to understand the system requirements. Should start with poorly understood requirements to clarify what is really needed.

Evolutionary development

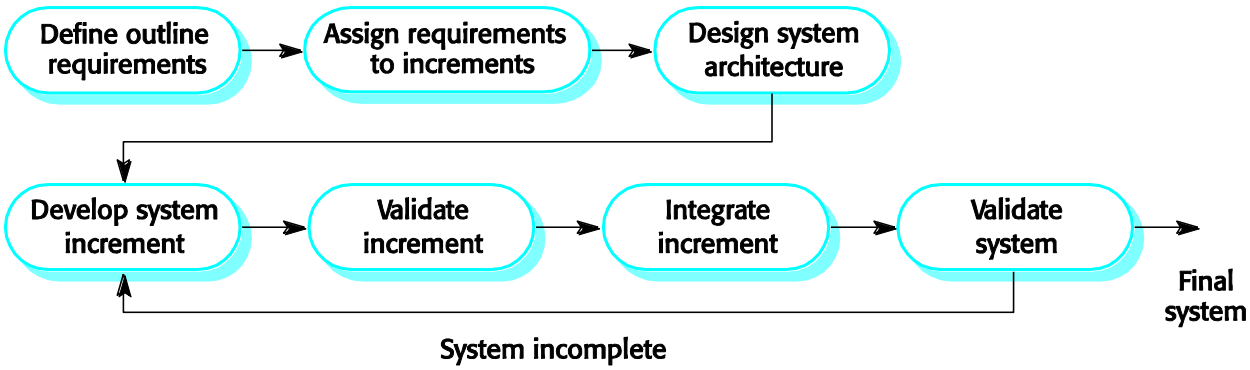


Evolutionary development

- Problems
  - Lack of process visibility;
  - Systems are often poorly structured;
  - Special skills (e.g. in languages for rapid prototyping) may be required.

- Applicability
  - For small or medium-size interactive systems;
  - For parts of large systems (e.g. the user interface);
  - For short-lifetime systems.

Incremental development



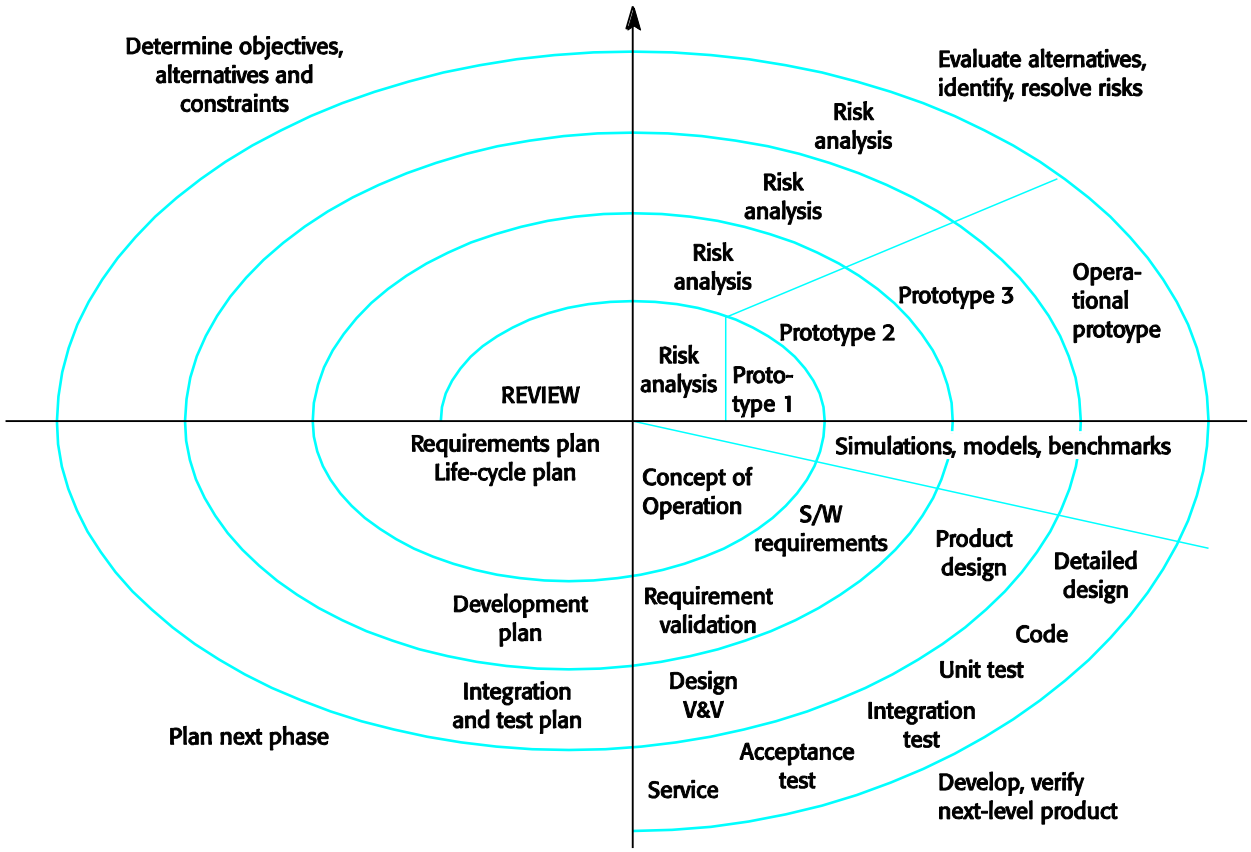
Incremental development advantages

- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.

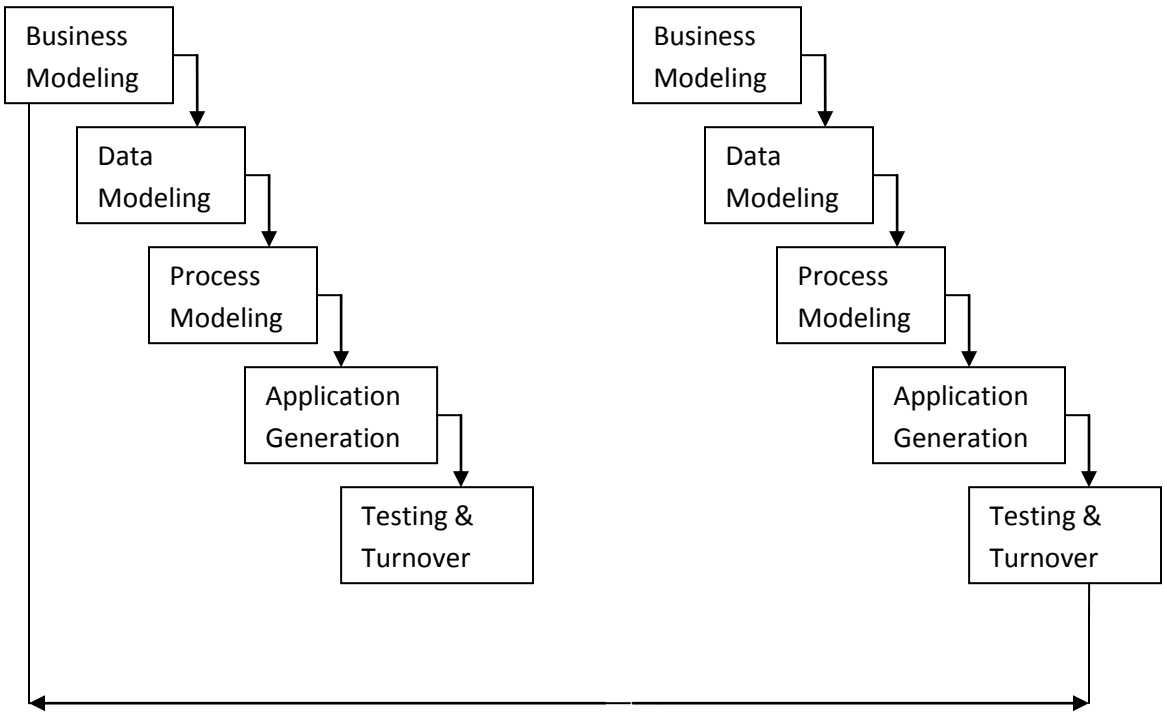
Spiral development

- Process is represented as a spiral rather than as a sequence of activities with backtracking.
- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- Risks are explicitly assessed and resolved throughout the process.

Spiral model of the software process


Determine objectives, alternatives and constraints
Evaluate alternatives, identify, resolve risks
Risk analysis
Risk analysis
Risk analysis
Prototype 3
Opera-tional protoype
Prototype 2
Risk analysis
Proto-type 1
REVIEW
Requirements plan Life-cycle plan
Simulations, models, benchmarks
Concept of Operation
S/W requirements
Product design
Detailed design
Development plan
Requirement validation
Code
Unit test
Integration and test plan
Design V&V
Integration test
Plan next phase
Acceptance test
Service
Develop, verify next-level product

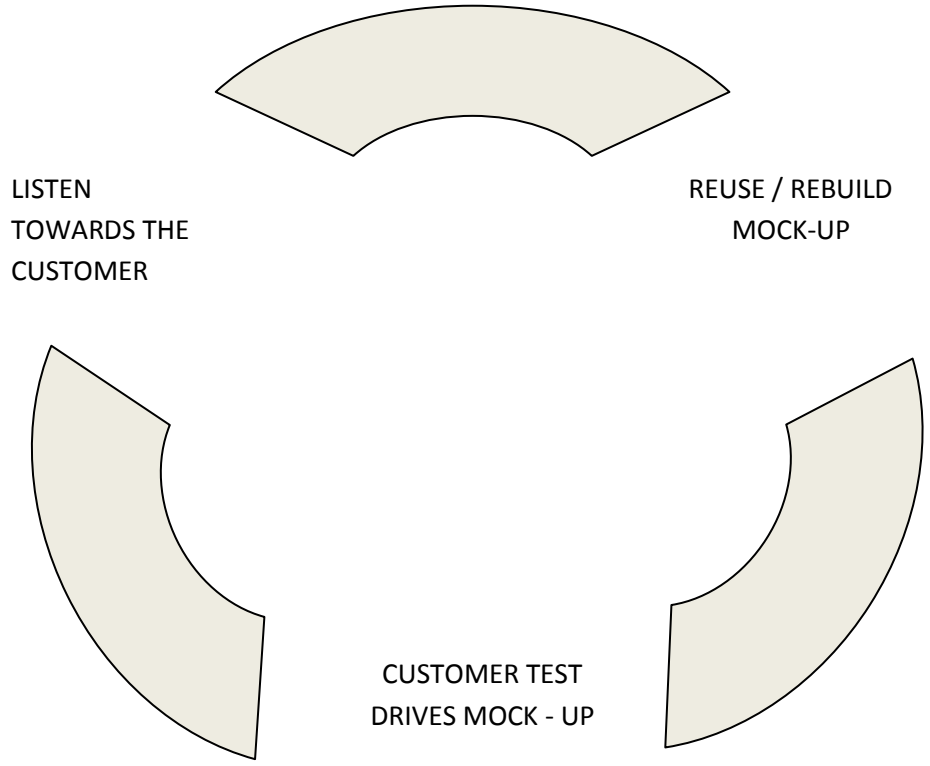
RAD MODEL:



PROTOTYPE MODEL:

It has 6 steps, They are as follows:

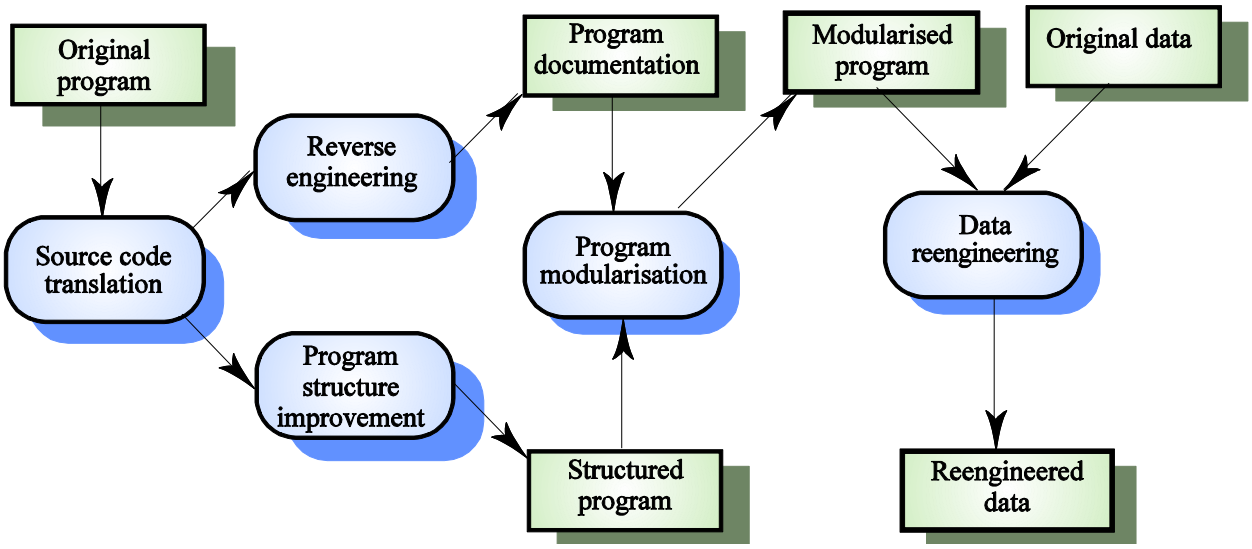
- Requirement collection
- Quick Design
- Prototype creation(or)modification
- Assessment
- Prototype refinement



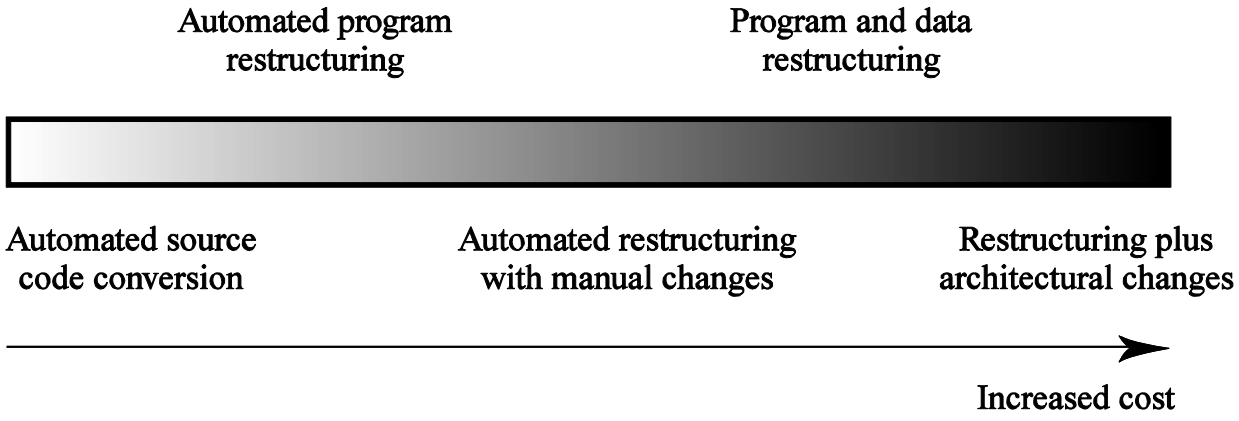
1.4 BUSINESS PROCESS ENGINEERING

- Concerned with re-designing business processes to make them more responsive and more efficient
- Often reliant on the introduction of new computer systems to support the revised processes
- May force software re-engineering as the legacy systems are designed to support existing processes

The re-engineering process



Re-engineering approaches



1.5 SYSTEM ENGINEERING

What is a system?

- A purposeful collection of inter-related components working together towards some common objective.

- A system may include software, mechanical, electrical and electronic hardware and be operated by people.
- System components are dependent on other system components
- The properties and behaviour of system components are inextricably inter-mingled

Problems of systems engineering

- Large systems are usually designed to solve 'wicked' problems Systems engineering requires a great deal of co-ordination across disciplines
- Almost infinite possibilities for design trade-offs across components
- Mutual distrust and lack of understanding across engineering disciplines
- Systems must be designed to last many years in a changing environment

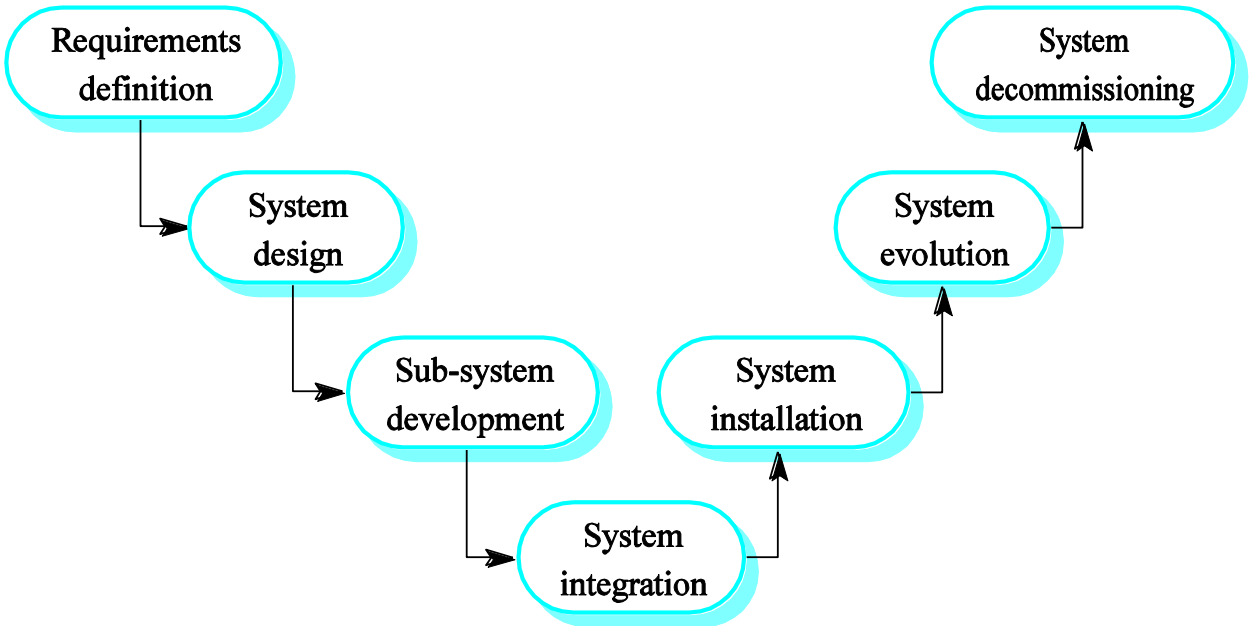
Software and systems engineering

- The proportion of software in systems is increasing. Software-driven general purpose electronics is replacing special-purpose systems
- Problems of systems engineering are similar to problems of software engineering
- Software is (unfortunately) seen as a problem in systems engineering. Many large system projects have been delayed because of software problems

The system engineering process

- Usually follows a 'waterfall' model because of the need for parallel development of different parts of the system
  - Little scope for iteration between phases because hardware changes are very expensive. Software may have to compensate for hardware problems
- Inevitably involves engineers from different disciplines who must work together
  - Much scope for misunderstanding here. Different disciplines use a different vocabulary and much negotiation is required. Engineers may have personal agendas to fulfil

The systems engineering process



1.6 COMPUTER-BASED SYSTEMS

Definition

A set or arrangement of elements that are organized to accomplish some predefined goal by processing information.

The goal may be to support some business function or to develop a product that can be sold to generate business revenue. To accomplish the goal, a computer-based system makes use of a variety of system elements:

Software. Computer programs, data structures, and related documentation that serve to effect the logical method, procedure, or control that is required.

Hardware. Electronic devices that provide computing capability, the interconnectivity devices (e.g., network switches, telecommunications devices) that enable the flow of

data, and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function.

People. Users and operators of hardware and software.

Database. A large, organized collection of information that is accessed via software.
Documentation. Descriptive information (e.g., hardcopy manuals, on-line help files, Web sites) that portrays the use and/or operation of the system.

Procedures. The steps that define the specific use of each system element or the procedural context in which the system resides.

The elements combine in a variety of ways to transform information. For example, a marketing department transforms raw sales data into a profile of the typical purchaser of a product; a robot transforms a command file containing specific instructions into a set of control signals that cause some specific physical action. Creating an information system to assist the marketing department and control software to support the robot both require system engineering.

1.7 Product Engineering Overview

Product Engineering

Product engineering is a crucial term in the sphere of software development. It is through product engineering that the future of a product is decided. The purpose of software Product Engineering is to consistently and innovatively perform a well-defined engineering process that integrates all software engineering activities to effectively and efficiently develop correct, consistent software products. Software Engineering tasks include analyzing the system requirements allocated to software, developing software architecture, designing the software, implementing the software in the code, integrating software components, and testing the software to verify whether it specifies specific requirements.

Product Conceptualization Engineering

- Write product marketing/business requirements specifications (MRS, BRS, PRD), system requirements specifications and functional specifications (SRS, FS)
- Identify and design key features
- Select architecture and design
- Provide UI prototypes

Product Architecture Consulting

- Construct the technology foundations needed to build robust products
- Consult on Enterprise Application Integration, Distributed Computing, Transaction Management

- Select architectural styles and patterns

Product Design and Implementation

- Draw a development strategy
- Integrate and customize products to meet requirements
- Train the end-user on product skills
- Reinforcing product best practices
- Testing for any technical issue

UNIT II

SOFTWARE REQUIREMENTS

2.1Requirement Engineering Process

Software Requirements Specification(SRS):

The s/w requirement and specification focuses on what the system will do, not how the system will be implemented. It is produced as the culmination of the s/w requirements analysis task in the lifecycle model. You must analyze the information domain, the function, performance and behavior and interface requirement of the system. Software requirements can be specified in the following ways.

- representation format and content should be relevant to the problem
- information contain within the specification should be nested.
- Representations should be revisable.
- s/w requirements specification produced at the culmination of the analysis task. This also states the goal and objectives of the s/w.
- information description provides a detailed description of the problem that the s/w must solve.
- Functional description is a description of each function required to the solve the problem.
- Behavioral description section of the specification examines the operation of the s/w as a consequence of external events and internally generated control characteristics.
- Validation criteria is the most important and ironically the most often neglected section of requirements specification.
- Functional vs. non functional requirements

(i) functional requirements: statements of services the systems should provide, how the system should react to particular inputs and how the systems should behave in particular situations.

(ii) Non functional requirements: constraints on the services or functions offered by the system such as timing constraints, constrains on the development process, standards etc.,

2.1.1Requirement engineering process:

Requirement engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need accessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification and managing the requirements as they are transformed into an operational system. The requirement engineering process can be described in five distinct steps.

- requirement elicitation
- requirement analysis and negotiation
- requirement specification
- system modeling
- requirement validation
- requirement management

2.2 software prototyping:

A model of s/w to be built is called a prototype. A prototype is constructed for customer and developer assessment.

(i) selecting a prototyping approach:

* the prototyping paradigm can be either close ended or open ended

* the close ended approach also called throw away prototyping. Using this a prototype serves as a rough demonstration of requirements.

* an open ended approach, called evolution of the prototyping uses the prototype as the first part of an analysis activity that will be continued into design and construction.

* Before a close ended or open ended approach can be chosen, it is necessary to determine whether the system to be built is amenable to prototyping.

* Prototyping factors are application area, application complexity, customer characteristics and project characteristics.

(ii) prototyping s/w process:

* in common with other types of s/w development, the prototyping process follows a define s/w process model.

* This model indicated the processes and tasks, which have to be performed during development of the prototype.

* Process model device for this particular approach comprise the following stages.

1. analysis requirements: This involved the developer understanding the content and nature of the customer's initial requirements.

2. prototype design: Here the developer should choose a suitable implementation approach for which to develop the prototype. Also a design is derived for the prototype based upon the results of analysis phase.

3. Prototype construction: This stage involves actual coding of the prototype

Rapid prototyping:

* An evolutionary s/w prototyping process is produced based on a requirement analysis of the customer's problem. This analysis is needed to ensure the initial version of the prototype is close enough to what the customer need to enable them to provide meaningful evaluations and criticism.

* Each succeeding version of the prototype is produced based upon an analysis of the customer's reaction to the demonstration of the previous version.

* Delivered products are delivered from the prototypes that are accepted by the customers via an optimal optimization process.

* maintenance activities are sparked by new customer's requirements, which restart the prototyping process and extent the series of prototypes until a new stable point is reached.

* The spiral model is well known because it combines the common knowledge of water fall model, incremental method and process model work into an attractive notation, even with less specific variation of the process.

* Some advantage of this approach are that prototype of different aspects of the system can be developed concurrently and independently, that each fragment is relatively small, simple and easy to change, and that different tools and environments can be used for different aspects. The last property can be important in the short term, if tools are available for solving different parts of the problem, but these tools have not been integrated together into a comprehensive prototyping environment.

2.3 SOFTWARE REQUIREMENTS

Descriptions and specifications of a system

FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

- **Functional requirements**
    - **Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.**
- **Non-functional requirements**
    - **constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.**
- **Domain requirements**
    - **Requirements that come from the application domain of the system and that reflect characteristics of that domain**

Functional requirements

- **Describe functionality or system services**
- **Depend on the type of software, expected users and the type of system where the software is used**
- **Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail**

Examples of functional requirements

- **The user shall be able to search either all of the initial set of databases or select a subset from it.**
- **The system shall provide appropriate viewers for the user to read documents in the document store.**
- **Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.**

Non-functional requirements

- **Define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.**

- **Process requirements may also be specified mandating a particular CASE system, programming language or development method**

- **Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless**

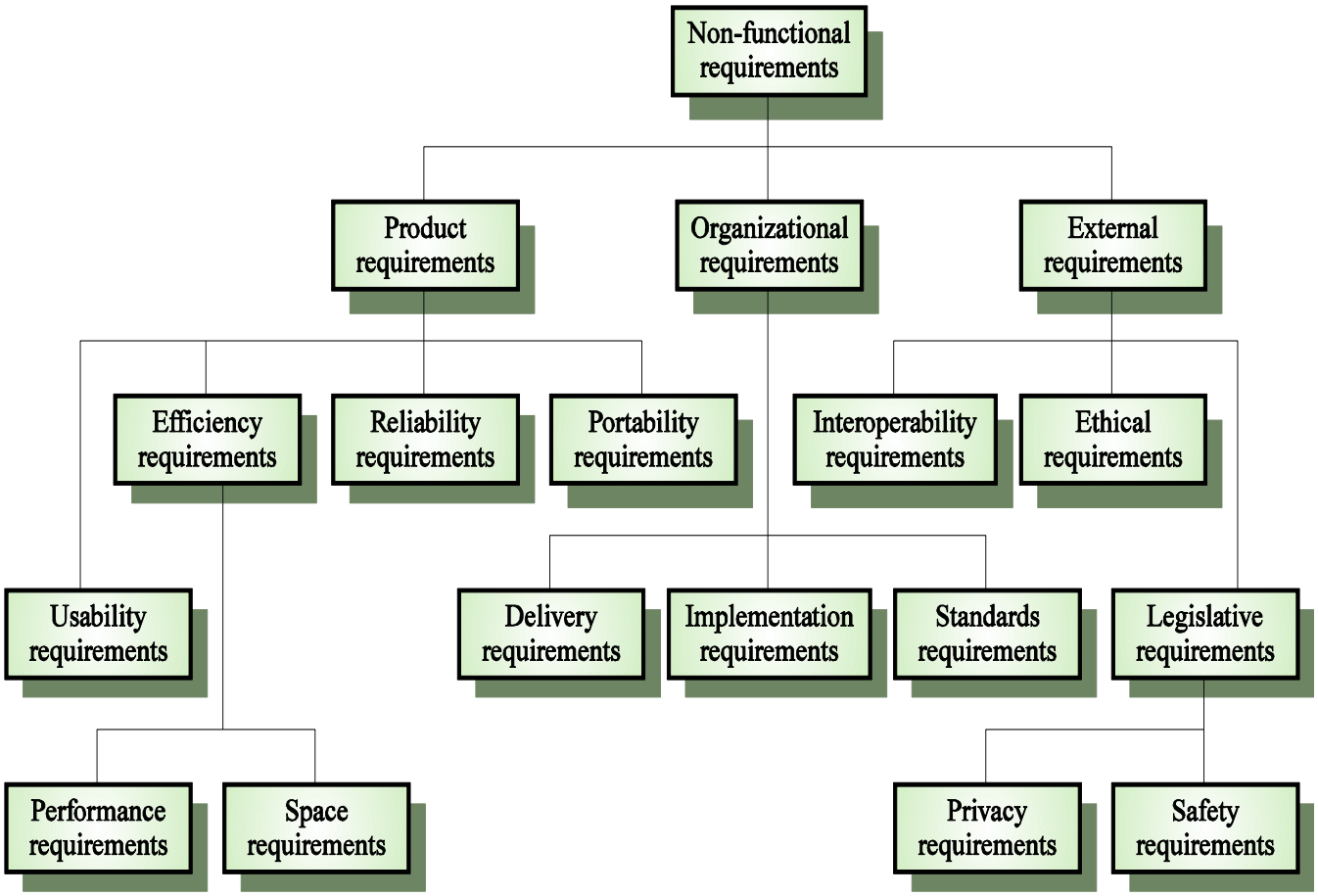
Non-functional classifications

- **Product requirements**
    - **Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.**

- **Organisational requirements**
    - **Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.**

- **External requirements**
    - **Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.**

Non-functional requirement types



Non-functional requirements examples

- **Product requirement**
  - **4.C.8 It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set**

- **Organisational requirement**
  - **9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95**

- **External requirement**
  - **7.6.5 The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system**

2.4 THE REQUIREMENTS DOCUMENT

- **The requirements document is the official statement of what is required of the system developers**
- **Should include both a definition and a specification of requirements**
- **It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it**

Document requirements

- **Specify external system behaviour**
- **Specify implementation constraints**
- **Easy to change**
- **Serve as reference tool for maintenance**
- **Record forethought about the life cycle of the system i.e. predict changes**
- **Characterise responses to unexpected events**

Document structure

- **Introduction**
- **Glossary**
- **User requirements definition**
- **System architecture**
- **System requirements specification**
- **System models**
- **System evolution**
- **Appendices**
- **Index**

2.5 FEASIBILITY STUDY

A **feasibility study** is an evaluation of a proposal designed to determine the difficulty in carrying out a designated task. Generally, a feasibility study precedes technical development and project implementation. In other words, a feasibility study is an evaluation or analysis of the potential impact of a proposed project

Common Factors

Technology and system feasibility

The assessment is based on an outline design of system requirements in terms of Input, Processes, Output, Fields, Programs, and Procedures. This can be quantified in terms of volumes of data, trends, frequency of updating, etc. in order to estimate whether the new system will perform adequately or not. Technological feasibility is carried out to determine whether the

company has the capability, in terms of software, hardware, personnel and expertise, to handle the completion of the project

Economic feasibility

Economic analysis is the most frequently used method for evaluating the effectiveness of a new system. More commonly known ascost/benefit analysis, the procedure is to determine the benefits and savings that are expected from a candidate system and compare them with costs. If benefits outweigh costs, then the decision is made to design and implement the system. An entrepreneur must accurately weigh the cost versus benefits before taking an action.

Cost Based Study: It is important to identify cost and benefit factors, which can be categorized as follows: 1. Development costs; and 2. Operating costs. This is an analysis of the costs to be incurred in the system and the benefits derivable out of the system.

Time Based Study: This is an analysis of the time required to achieve a return on investments. the benefits derived from the system. The future value of a project is also a factor.

Legal feasibility

Determines whether the proposed system conflicts with legal requirements, e.g. a data processing system must comply with the local Data Protection Acts.

Operational feasibility

Is a measure of how well a proposed system solves the problems, and takes advantage of the opportunities identified during scope definition and how it satisfies the requirements identified in the requirements analysis phase of system development.[1]

Schedule feasibility

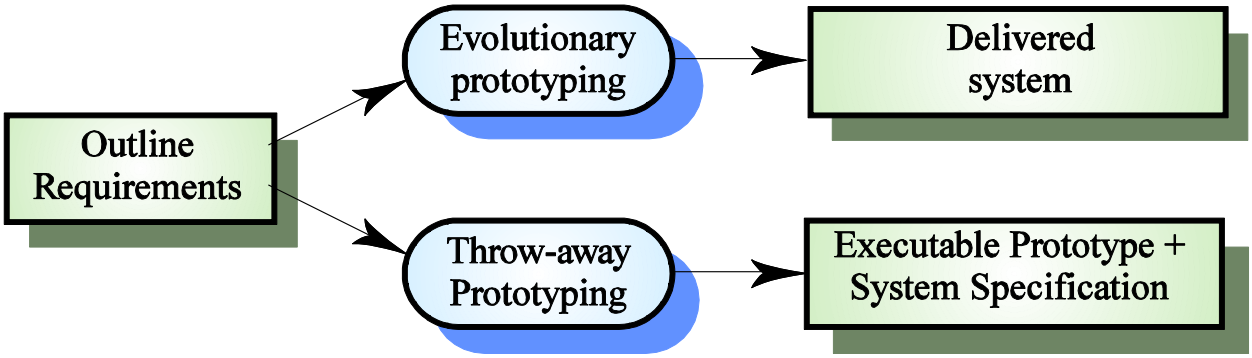
A project will fail if it takes too long to be completed before it is useful. Typically this means estimating how long the system will take to develop, and if it can be completed in a given time period using some methods like payback period. Schedule feasibility is a measure of how reasonable the project timetable is. Given our technical expertise, are the project deadlines reasonable? Some projects are initiated with specific deadlines. You need to determine whether the deadlines are mandatory or desirable...

2.6 Prototyping in the software process

- **Evolutionary prototyping**
  - **An approach to system development where an initial prototype is produced and refined through a number of stages to the final system**

- **Throw-away prototyping**
  - **A prototype which is usually a practical implementation of the system is produced to help discover requirements problems and then discarded. The system is then developed using some other development process**

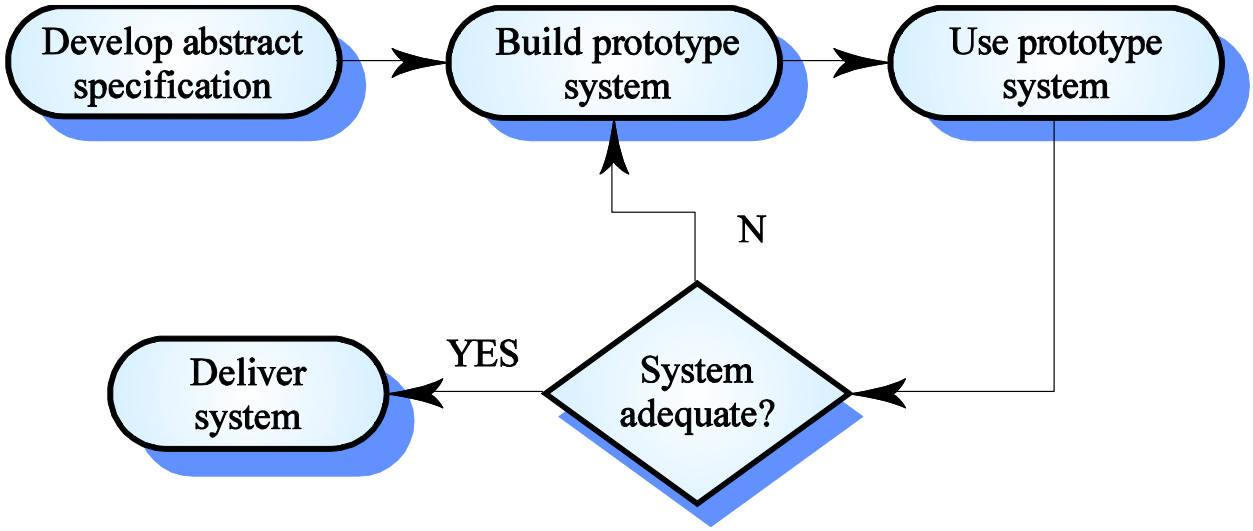
Approaches to prototyping



Evolutionary prototyping

- **Must be used for systems where the specification cannot be developed in advance e.g. AI systems and user interface systems**
- **Based on techniques which allow rapid system iterations**
- **Verification is impossible as there is no specification. Validation means demonstrating the adequacy of the system**

Evolutionary prototyping



Evolutionary prototyping advantages

- **Accelerated delivery of the system**
  - **Rapid delivery and deployment are sometimes more important than functionality or long-term software maintainability**
- **User engagement with the system**
  - **Not only is the system more likely to meet user requirements, they are more likely to commit to the use of the system**

Evolutionary prototyping

- **Specification, design and implementation are inter-twined**
- **The system is developed as a series of increments that are delivered to the customer**
- **Techniques for rapid system development are used such as CASE tools and 4GLs**
- **User interfaces are usually developed using a GUI development toolkit**

Throw-away prototyping

- **Used to reduce requirements risk**
- **The prototype is developed from an initial specification, delivered for experiment then discarded**

- **The throw-away prototype should NOT be considered as a final system**
    - **Some system characteristics may have been left out**
    - **There is no specification for long-term maintenance**
    - **The system will be poorly structured and difficult to maintain**

Rapid prototyping techniques

- **Various techniques may be used for rapid development**
    - **Dynamic high-level language development**
    - **Database programming**
    - **Component and application assembly**
- **These are not exclusive techniques - they are often used together**
- **Visual programming is an inherent part of most prototype development systems**

2.7 DATA-PROCESSING MODELS

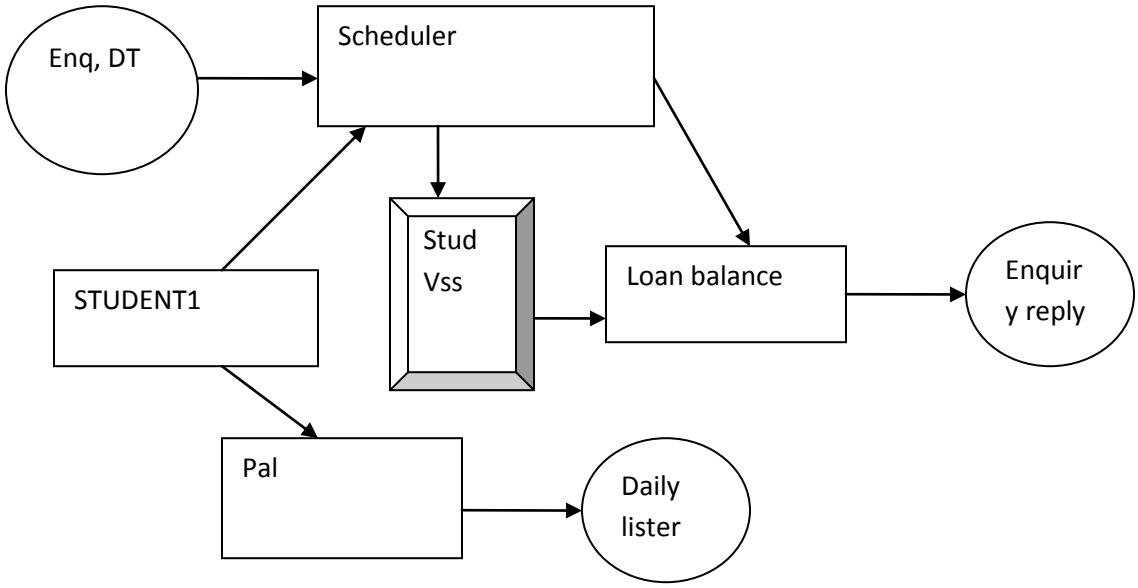
DATA MODEL

- **Data flow diagrams are used to model the system's data processing**
- **These show the processing steps as data flows through a system**
- **Intrinsic part of many analysis methods**
- **Simple and intuitive notation that customers can understand**
- **Show end-to-end processing of data**

Data flow diagrams

- **DFDs model the system from a functional perspective**
- **Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system**

- **Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment**



FUNCTIONAL MODEL

Functional modeling methods

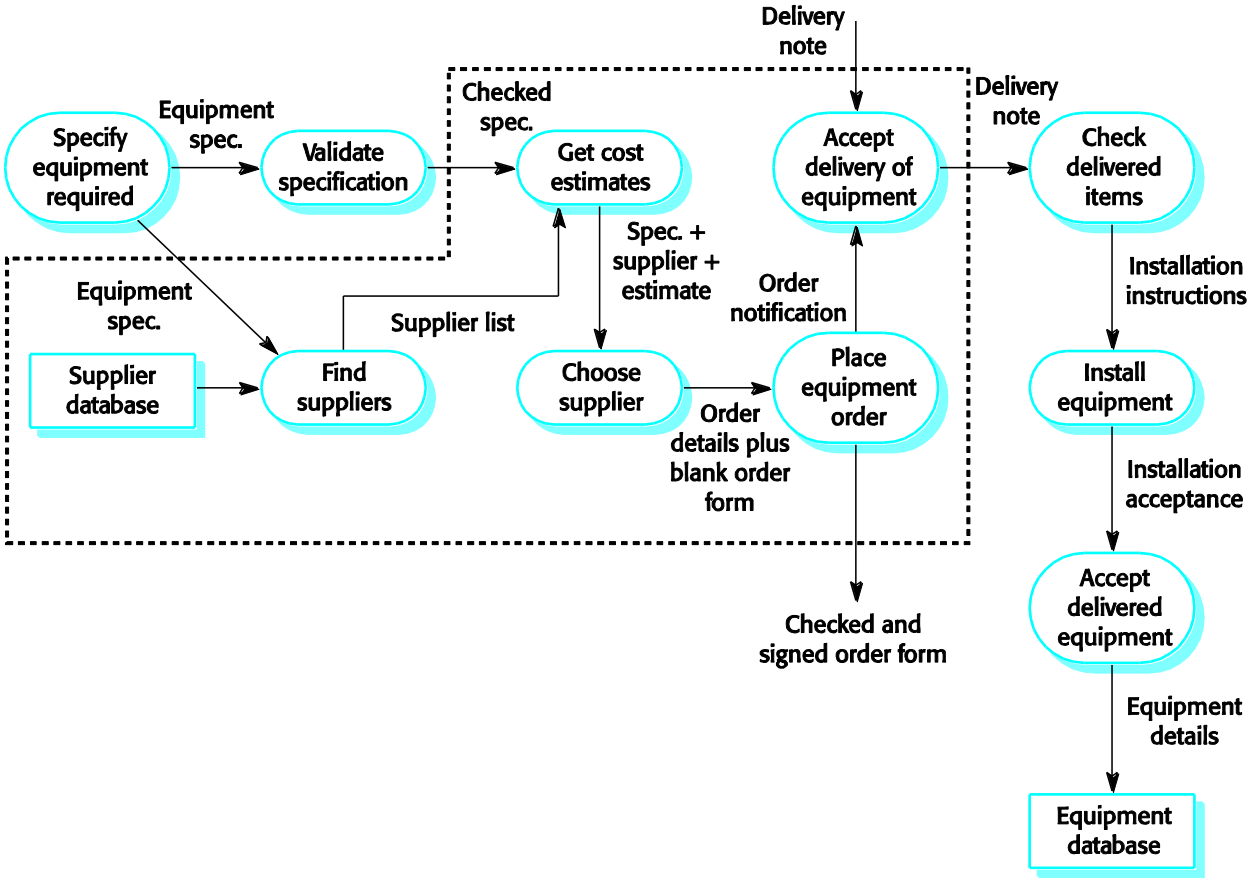
The functional approach is extended in multiple diagrammic techniques and modeling notations. This section gives an overview of the important techniques in chronological order.

Functional Flow Block

The Functional flow block diagram (FFBD) is a multi-tier, time-sequenced, step-by-step flow diagram of the system's functional flow.The diagram is developed in the 1950s and widely used in classical systems engineering. The Functional Flow Block Diagram is also referred to as Functional Flow Diagram, functional block diagram, and functional flow.

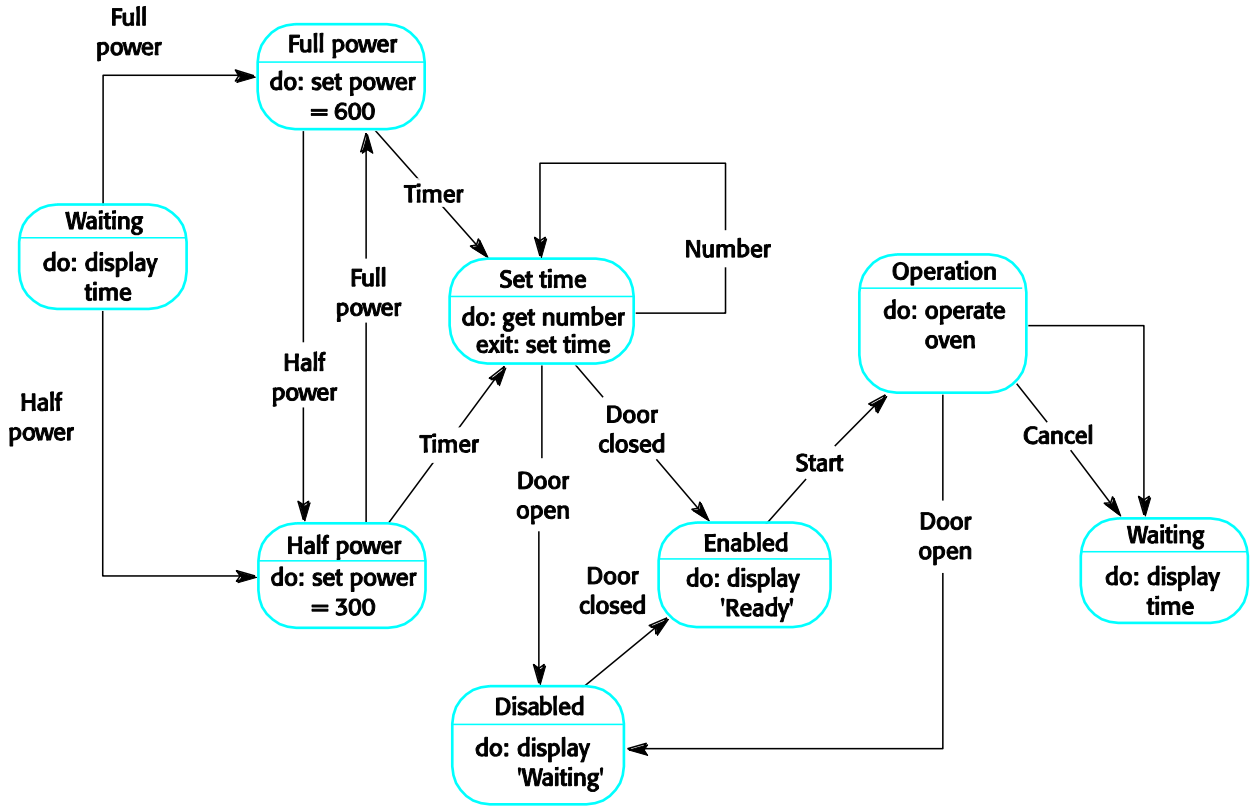
Functional Flow Block Diagrams (FFBD) usually define the detailed, step-by-step operational and support sequences for systems, but they are also used effectively to define processes in developing and producing systems. The software development processes also use FFBDs extensively. In the system context, the functional flow steps may include combinations of

hardware, software, personnel, facilities, and/or procedures. In the FFBD method, the functions are organized and depicted by their logical order of execution. Each function is shown with respect to its logical relationship to the execution and completion of other functions. A node labeled with the function name depicts each function. Arrows from left to right show the order of execution of the functions. Logic symbols represent sequential or parallel execution of functions.



2.8 Structured Analysis and Design Technique

Structured Analysis and Design Technique (SADT) is a software engineering methodology for describing systems as a hierarchy of functions, a diagrammatic notation for constructing a sketch for a software application. It offers building blocks to represent entities and activities, and a variety of arrows to relate boxes. These boxes and arrows have an associated informal semantics.[19] SADT can be used as a functional analysis tool of a given process, using successive levels of details. The SADT method allows to define user needs for IT developments, which is used in industrial Information Systems, but also to explain and to present an activity's manufacturing processes, procedures.



DATA DICTIONARIES

- **Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included**
  - **Advantages**
    - **Support name management and avoid duplication**
  - **Store of organisational knowledge linking analysis, design and implementation**
- **Many CASE workbenches support data dictionaries**

Data dictionary entries

Name	Description	Type
has-labels	1:N relation between entities of type Node or Link and entities of type Label.	Relation

Label	Holds structured or unstructured information about nodes or links. Labels are represented by an icon (which can be a transparent box) and associated text.	Entity
Link	A 1:1 relation between design entities represented as nodes. Links are typed and may be named.	Relation
name (label)	Each label has a name which identifies the type of label. The name must be unique within the set of label types used in a design.	Attribute
name (node)	Each node has a name which must be unique within a design. The name may be up to 64 characters long.	Attribute

UNIT III

ANALYSIS, DESIGN CONCEPTS AND PRINCIPLES

3.1Software Design process and concepts:

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

Design Specification Models:

Data design –created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software.

Architectural design-defines the relationships among the major structural elements of the software, it is derived from the system specification, the analysis model, the subsystem interactions defined in the analysis model (dfd).

Interface design-describes how the software elements communicate with each other, with other systems, and with human users, the data flow and control flow diagrams provide much the necessary information.

Component –Level design-created by transforming the structural elements defined by the software architecture in to procedural descriptions of software components using information obtained from the PSPEC, CSPEC, and STD

Design Guidelines:

A design should:

- Exhibit good architectural structure.
- Be modular
- Contain distinct representatives of data architecture, interfaces, and components (modules)
- Lead to data structures are appropriate for the objects to be implemented that exhibit independent functional characteristics

- Lead to interfaces reduce the complexity of connections between modules and with the external environment
- Be derived using a reputable method is driven by information obtained during software requirements

Design Principles:

The design

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

Fundamental Software Design Concepts:

- Abstraction-allows designers to focus on solving a problem without being concerned about irrelevant lower level details(procedural abstraction –named sequence of events, data abstraction-named collection of data objects)
- Refinement –process of elaboration where the designer provides successively more detail for each design component.
- Modularity – the degree to which software can be understood by examining its components independently of one another.
- Software architecture – overall structure of the software components and the ways in which structure provides conceptual integrity for a system.
- Control hierarchy or program structure-represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e,g,event sequences)
- Structural portioning – horizontal partitioning defines three partitions(input, data transformations, and output); vertical partitioning (factoring) distributes control in a top down manner(control decisions in top level modules and processing work in the lower level modules)
- Data structure –representation of the logical relationship among individual data elements (requires at least as much attention as algorithm design)
- Software procedure –precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure)

Information hiding –information (data and Procedure) contained within a module is inaccessible to modules have no need for such information.

Abstraction

Abstraction is the theory that allows one to deal with concepts apart from the particular instances of those concepts.

- Abstraction reduces complexity to a larger extent during design
- Abstraction is an important tool I software engineering in many aspects.
- There are three types of abstractions used in software design
    a) Functional abstraction.
    b) Data abstraction
    c) Control abstraction

Functional abstraction:

-This involves the usage of parameterized routines.

-The number and type of parameters to a routine can be made dynamic and this ability to use the apt parameter during the apt invocation of the sub-program is functional abstraction

Data Abstraction:

This involves specifying or creating a new data type or a date object by specifying valid operations on the object.

-Other details such as representative and manipulations of the data are not specified.

- Many languages is an important feature of OOP.

- Data abstraction is an important as ADA, C++, provide abstract data type.

-Example: Implementation of stacks, queues.

Control Abstraction:

-Control abstraction is used to specify the effect of a statement or a function without

Stating the actual mechanism of control.

MODULARITY

Modularity derives from the architecture. Modularity is a logical partitioning of the software design that allows complex software to be manageable for purpose of implementation and maintenance. The logic of partitioning may be based on related functions, implementations considerations, data links, or other criteria. Modularity does imply interface overhead related to information exchange between modules and execution of modules.

- Modularity – the degree to which software can be understood by examining its components independently of in another

COHESION:

Cohesion is an interaction within a single object of software component.

It reflects the single-purposeless of an object.

Coincidentally cohesive is cohesive that performs a set of tasks, that relate to each other object.

Logically cohesive is a cohesive process that performs tasks that are related logically each other objects.

Method cohesion, like the function cohesion, means that a method should carry only function.

Inheritance cohesion concerned with interrelation of classes, specialization of classes with attributes.

Coupling:

- Coupling is a measure of interconnection among modules in a software structure. It depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- It is a measure of the strength of association established by a connection from one object or s/w component to another.
- It is binary relationship: A is coupled with B.

Types of coupling:

Common Coupling:

It is high coupling occurs when a number of modules (object)

Reference a global data area.

The objects will access a global data space for both to read and write operations of attributes.

Content Coupling:

It is the degree of coupling. It occurs when one object or module makes use of data control information maintained within the boundary of another object or module.

It refers to attributes or methods of another object.

Control coupling:

It is characterized by passage of control between modules or objects.

It is very common in most software designs.

It involves explicit control of the processing logic of one object by another.

Stamp Coupling:

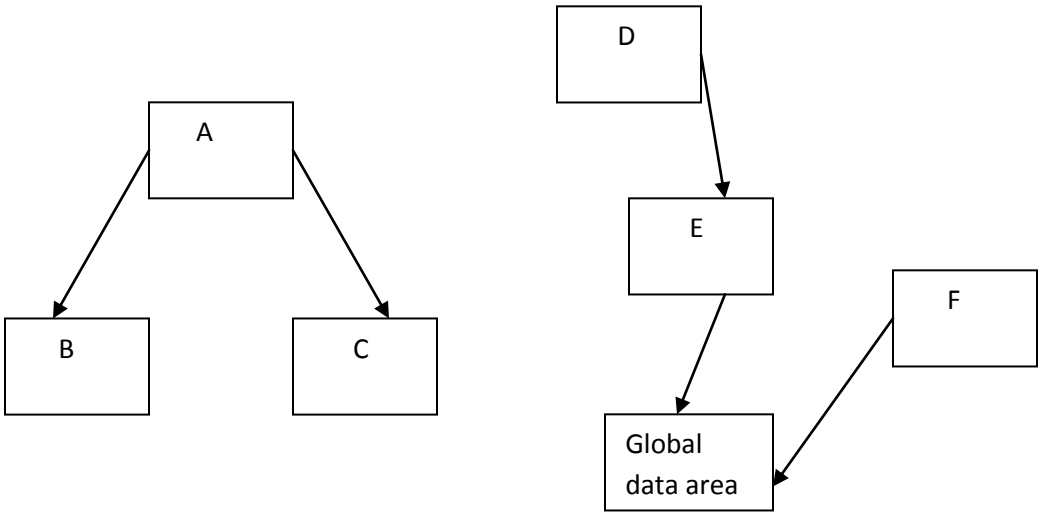
The connection involves passing an aggregate data structure to another object, which uses only a portion of the components of the data structure.

It is found when a portion of a data structure is passed via a module or object interface.

Data Coupling:

- It is low degree of coupling.
- The connection involves either simple data items or aggregate structures all of whose elements are used by the receiving object.
- This should be the goal of an architectural design.
- It is exhibited in the portion of structure.

Types of coupling:



DESIGN CONCEPTS AND NOTATIONS:

Information Hiding;

- Information hiding is an OOP concept.
- Each module in a software system hides its processing details activities and communicates with the other modules only through well-defined interfaces.
- Data abstraction is also example of information hiding.
- Information hiding can be used as a whole in the architecture design of the software system or in conjunction with other design techniques.

Concurrency:

- Concurrency systems are those systems n which there are multiple independent processes, which can be activated simultaneously.
- On a multi-processor system sharing them across the processors can do such a task.
- On a single processor system, concurrency can be achieved by process of interleaving.
- The problems encountered in a concurrent system are Deadlock, Mutual Exclusion and Process Synchronization.

Verification:

- Verification is basic concept of software design.

- A design can be accepted by customer only if it is verified & satisfies customer's requirements.
- The design is the kick-start for the project and so its verification to meet the client's needs is important.

Aesthetics:

- In any art and technology aesthetic points are to be considered.
- In software design the aesthetic considerations is an important area to be fine turned by the designer to meet the end-user taste.
- An aesthetically pleasing software product, though difficult to design, makes the overall look and feel better and hence the product is well appreciated.
- This is an important factor in the current world of DOTCOMS with website designs and user interfaces.

Structure:

- A structure is a fundamental concept of software design.
- As in modularity structure permits the breaking up or decompositions of a larger system into smaller units with well-defined relationships between the different units.
- A network is a good example of a structure.
- A network has nodes and arcs and represented as a directed graph.

Design Steps:

Step 1: Review the fundamental system model.

Step 2: Review and refine data flow diagrams for the software.

Step 3: Determine whether the DFD has transform or transaction characteristics.

Step 4: Identify the transaction center and the flow characteristics along each of the action paths.

Step 5: Map the DFD in a program structure amenable to transaction processing.

Step 6: Factor and refine the transaction structure and the structure of each action path.

Step 7: Refine the first-iteration architecture using design heuristics for improved software quality.

- Well suited to stepwise refinement.
- Can be presented and reviewed at varying levels of detail.
- Well suited to top-down implementations.
- Results in programs that are well structured, easy to implement, modify, and test.

- Structures are easy to represent on a computer screen.

3.2 Modular Design:

Isolating the details of certain activities within procedures we obtain a program that is expressed clearer than if it had all activities included. Modularity in a software system is where modules take the form of objects or units each with an internal structure independent of other objects or units. The reason for the popularity of object oriented approach is its modularity as when modifying certain parts it can be done with less affect on the rest of the program.

SOFTWARE IS DIVIDED INTO SEPERATELY NAMED AND ADDRESSABLE COMPONENTS OTEN CALLED MODULES.

Any good design requires the entire software product to be split onto several modules or smaller units.

Examples of modules: functions, procedures, data abstraction groups.

Modules contain instructions data structures

They can be stored separately

Modules can be compiled separately.

Modules are used by invoking their name and associated arguments.

They can all other modules also

Advantages of modularization:

- Hierarchy in the operations.
- Data abstractions
- Independent powerful subsystems.
- Inheritance
- Reusability
- Ease in testing debugging
- Ease in implementation.

Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

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 modular solution that does not reinvent the wheel.

Modular Understandability: If a module can be understood as a stand-alone 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.

Modularity Design:

- Module can be defined in many ways. Generally a module is a work allocation for a programmer. Fortran & ADA define module in a different manner.
- However, modularity is the concept of breaking the entire system into well-defined manageable units with well-defined interfaces between these units.
- A modular system follows the concept of abstraction also.
- Modular programming enhances clarity in design, reduces complexity and hence enables ease of implementation, testing documenting and maintenance.

Modular design Method Evaluation Criteria:

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

Effective Modular Design:

- Functional independence – modules have high cohesion and low coupling.

- Cohesion-qualitative indication of the degree to which a module focuses on just one thing.
- Coupling – qualitative indication of the degree to which a module is connected to other modules and to the outside world.

3.3 Design Heuristics for Effective Modularity:

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

3.4 SORTWARE ARCHITECTURE

Software Architecture:

While refinement is about the level of detail, architecture is about structure. The architecture of the procedural and date elements of a design represents a software solution for the real-world problem defined by the requirements analysis. It is unlikely that there will be one obvious candidate architecture.

- Software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of assemblage.
- Effective software architecture and its explicit representation and design have become dominant themes in software engineering.
- The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.
- The architecture is not the operational software.
- Rather, it is a representation that enables a software engineer to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage with making design changes is till relatively easy, and 3) reducing the risks associated with the construction of the software.
- Importance of Architecture.

Representations of software architecture are an enabler for communications between all parties interested in the development of a computer-based system.

The architecture highlights early design decisions that will have a profound impact 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"

- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is *architectural design*
- The output of this design process is a description of the *software architecture*

Architectural design

- An early stage of the system design process
- Represents the link between specification and design processes
- Often carried out in parallel with some specification activities
- It involves identifying major system components and their communications

Architectural design process

- System structuring
    - The system is decomposed into several principal sub-systems and communications between these sub-systems are identified
- Control modelling
    - A model of the control relationships between the different parts of the system is established
- Modular decomposition
    - The identified sub-systems are decomposed into modules

Sub-systems and modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system

Architectural models

- Different architectural models may be produced during the design process
- Each model presents different perspectives on the architecture
- Static structural model that shows the major system components
- Dynamic process model that shows the process structure of the system
- Interface model that defines sub-system interfaces
- Relationships model such as a data-flow model

Architectural styles

- The architectural model of a system may conform to a generic architectural model or style
- An awareness of these styles can simplify the problem of defining system architectures
- However, most large systems are heterogeneous and do not follow a single architectural style

Architecture attributes

- Performance
    - Localise operations to minimise sub-system communication
- Security
    - Use a layered architecture with critical assets in inner layers
- Safety
    - Isolate safety-critical components
- Availability
    - Include redundant components in the architecture
- Maintainability
    - Use fine-grain, self-contained components

System structuring

- Concerned with decomposing the system into interacting sub-systems
- The architectural design is normally expressed as a block diagram presenting an overview of the system structure
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed

The repository model

- Sub-systems must exchange data. This may be done in two ways:
    - Shared data is held in a central database or repository and may be accessed by all sub-systems
    - Each sub-system maintains its own database and passes data explicitly to other sub-systems
- When large amounts of data are to be shared, the repository model of sharing is most commonly used

Repository model characteristics

- Advantages
    - Efficient way to share large amounts of data
    - Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
    - Sharing model is published as the repository schema
- Disadvantages
    - Sub-systems must agree on a repository data model. Inevitably a compromise
    - Data evolution is difficult and expensive
    - No scope for specific management policies
    - Difficult to distribute efficiently

Client-server architecture

- Distributed system model which shows how data and processing is distributed across a range of components
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services
- Network which allows clients to access servers

Client-server characteristics

- Advantages
    - Distribution of data is straightforward
    - Makes effective use of networked systems. May require cheaper hardware
    - Easy to add new servers or upgrade existing servers
- Disadvantages
    - No shared data model so sub-systems use different data organisation. data interchange may be inefficient
    - Redundant management in each server
    - No central register of names and services - it may be hard to find out what servers and services are available

3.5 REAL TIME AND DISTRIBUTED SYSTEM DESIGN:

There are many popular methodologies such as top-down, structured design and son supporting concepts such as inheritance, data hiding, abstraction, and modularity. Real time systems require these concepts too but with a higher degree of precision and clarity.

A distributed system consists of more processors operating in parallel with shared memory and also their own memory and communicates through a network. Real time systems and distributed

systems are used in process-control and other mission critical applications. So they require a lot of other trade-offs and considerations than the normal software systems.

In a real-time system, timing constraints must be met for the applications to be correct. A computing system is real-time to the degree that time constraints must be met for the applications to be correct.

This is a consequence of the system interacting with its physical environment. The environment produces stimuli, which must be accepted by the real-time system within the time constraints. For instance, in air traffic control system the environment consists of aircraft that must be monitored.

The environment stimuli are received by the system through sensors such as radars. The environment further requires control outputs, which must e produced within time constraint. In the air traffic control example, signals to the aircraft and displays to the human operators have time constraints that must be met. Time-constrained behaviour can obviously be critical to not just mission success, but even to the safety of property and human life.

In a distribute real-time systems, many of these time constraints are end-end and often require the scheduling of different resources (e.g. processors on each node and the communication facilities between them)

One of the things that make real-time resources management so much more difficult than non-real-time resource management is that the real-time performances requirements of acceptable predictability of timeliness must be met along with other requirements such as synchronized and resource utilization.

Other issues like scheduling, safe recovery due to loss of network link failure of a processing node have to be considered in the design of distributed systems.

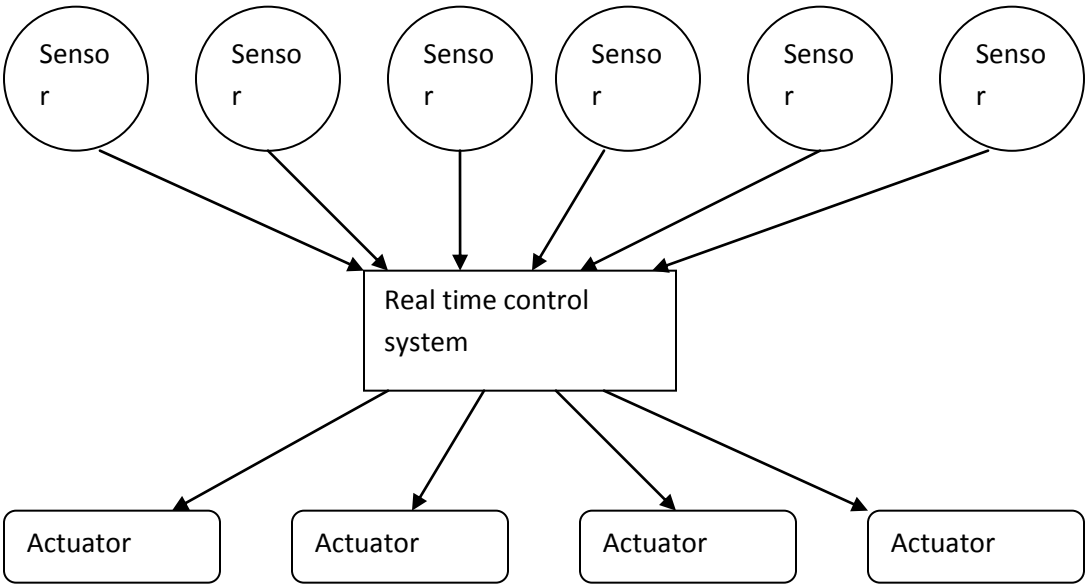
A fine of a real-time system requirements aiding tool is the SREM system. Features of SREM:

Flow=oriented approach of stimulus-response.

Associate performance characteristics with specific points in the processing sequence.

Provision of a simulation package for evaluation of systems design and choosing design alternatives.

A real-time system model:



System Elements:

Sensors control processes:

Collect information from sensors. May buffer information collected in response to a sensor stimuli.

Data processor:

Carries out processing of collected information and computes the system response.

Actuator control:

Generates control signals for the actuator.

System design:

- Design both the hardware and the software associated with system. Partition functions to either hardware or software.
- Design decisions should be made on the basis on non-functional system requirements.
- Hardware delivers better performance but potentially longer development and less scope for change.

Real time Executives:

Components of real time executives:

- A real=tome clock. This provides information to schedule process periodically.
- An interrupt handler. This manages a periodically requests for service.
- A scheduler. This component is responsible for examining the processes, which can be executed, and choosing one of these for execution.
- A resource manager. Given a process, which is scheduled for execution, the resource manager allocates appropriate memory and processor resources.

3.7 Monitoring and control systems:

- Monitoring systems are system which takes action when some exceptional sensor value is detected.
- Control systems are systems, which continuously control hardware actuators depending in the value of associated sensors.
- These systems obviously have a great deal in common and differ only in the way in which system actuators are initiated.
- Important class of real-time systems.
- Control systems take sensor values and control hardware actuators.

Example:

Burglar alarm system:

A system is required to monitor sensors on doors and windows to detect the presence of intruders in a building.

When a sensor indicates a break-in, the system switches on lights around that area and calls police automatically.

The system should include provision for operation without mains power supply.

Burglar alarm system:

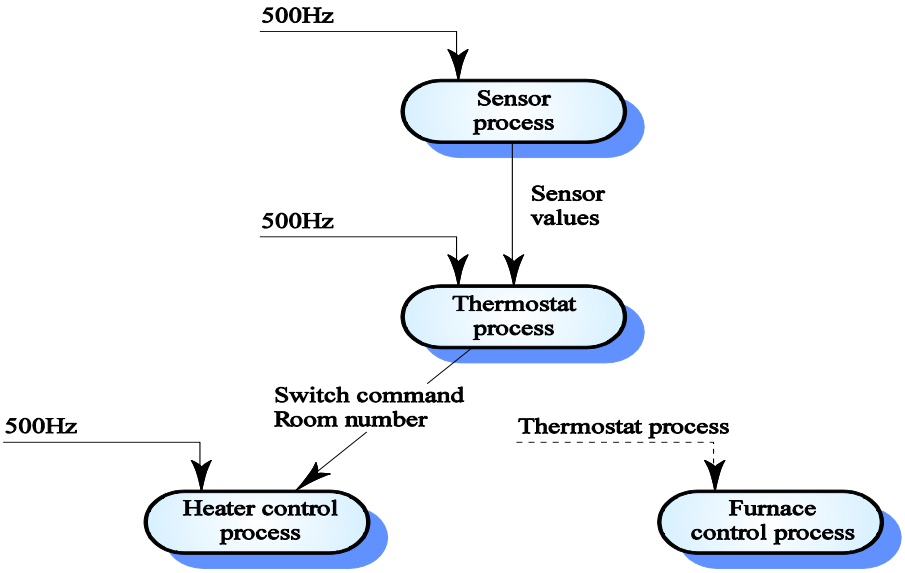
Sensors:

- Movement detectors window sensors, door sensors.
- 50 window sensors, 30 door sensors and 200 movement detectors
- Voltage drop sensor.

Actions:

- When a intruder is detected, police are called automatically.
- Lights are switched on in rooms with active sensors.
- The system switches automatically to backup power when a voltage drop is detected.

A temperature control system



3.6 WHAT IS A SYSTEM?

- A purposeful collection of inter-related components working together towards some common objective.
- A system may include software, mechanical, electrical and electronic hardware and be operated by people.
- System components are dependent on other system components
- The properties and behaviour of system components are inextricably inter-mingled

PROBLEMS OF SYSTEMS ENGINEERING

- Large systems are usually designed to solve 'wicked' problems
- Systems engineering requires a great deal of co-ordination across disciplines
    - Almost infinite possibilities for design trade-offs across components
    - Mutual distrust and lack of understanding across engineering disciplines
- Systems must be designed to last many years in a changing environment

SOFTWARE AND SYSTEMS ENGINEERING

- The proportion of software in systems is increasing. Software-driven general purpose electronics is replacing special-purpose systems
- Problems of systems engineering are similar to problems of software engineering
- Software is (unfortunately) seen as a problem in systems engineering. Many large system projects have been delayed because of software problems

Emergent properties

- Properties of the system as a whole rather than properties that can be derived from the properties of components of a system
- Emergent properties are a consequence of the relationships between system components
- They can therefore only be assessed and measured once the components have been integrated into a system

Examples of emergent properties

- *The overall weight of the system*
    - This is an example of an emergent property that can be computed from individual component properties.
- *The reliability of the system*
    - This depends on the reliability of system components and the relationships between the components.
- *The usability of a system*
    - This is a complex property which is not simply dependent on the system hardware and software but also depends on the system operators and the environment where it is used.

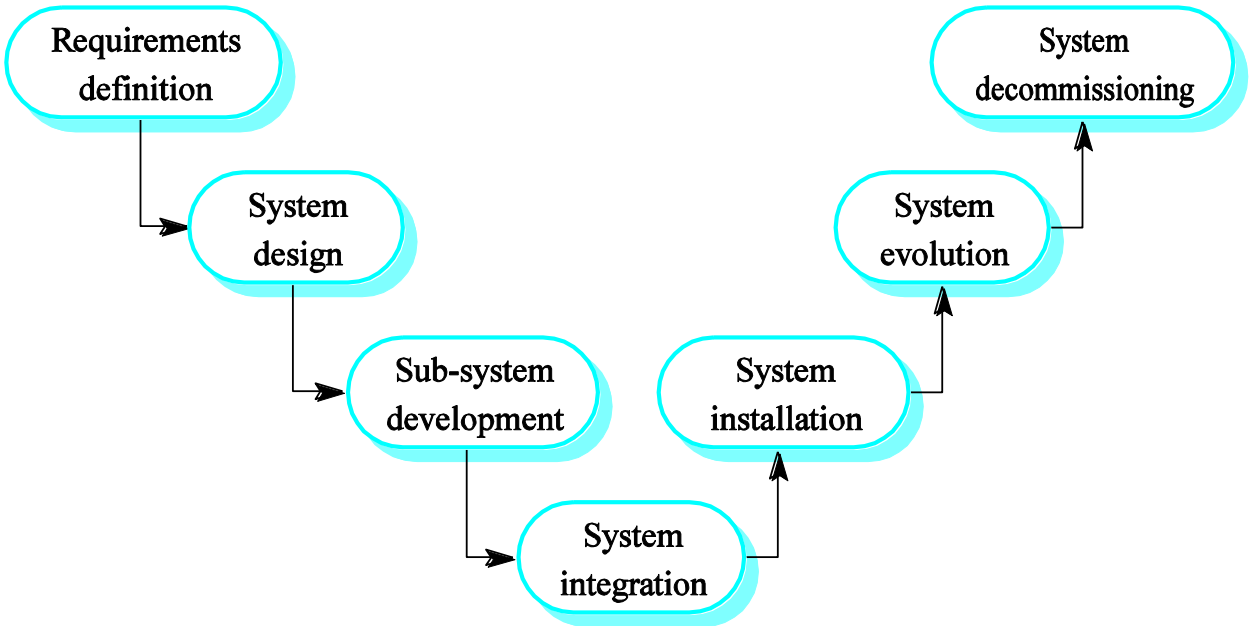
Types of emergent property

- Functional properties

- These appear when all the parts of a system work together to achieve some objective. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.

- Non-functional emergent properties
    - Examples are reliability, performance, safety, and security. These relate to the behaviour of the system in its operational environment. They are often critical for computer-based systems as failure to achieve some minimal defined level in these properties may make the system unusable.

THE SYSTEM ENGINEERING PROCESS



The system design process

- Partition requirements
    - Organise requirements into related groups
- Identify sub-systems
    - Identify a set of sub-systems which collectively can meet the system requirements
- Assign requirements to sub-systems
    - Causes particular problems when COTS are integrated

- Specify sub-system functionality
- Define sub-system interfaces
    - Critical activity for parallel sub-system development

3.7 THE USER INTERFACE

- System users often judge a system by its interface rather than its functionality
- A poorly designed interface can cause a user to make catastrophic errors
- Poor user interface design is the reason why so many software systems are never used

Graphical user interfaces

- Most users of business systems interact with these systems through graphical interfaces although, in some cases, legacy text-based interfaces are still used

GUI advantages

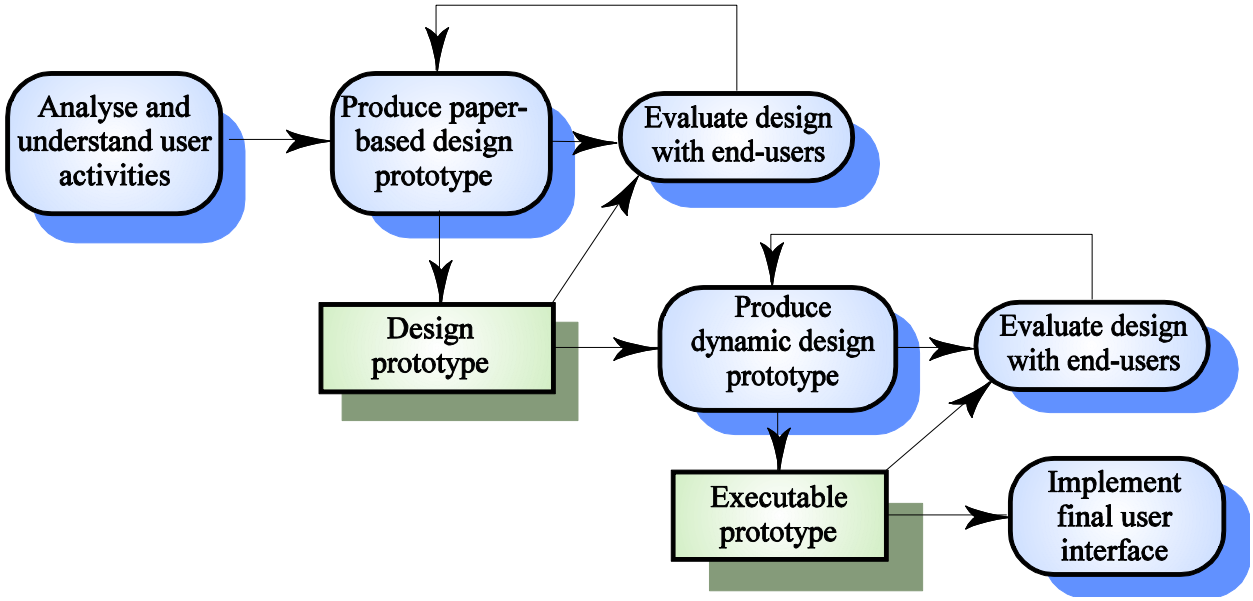
- They are easy to learn and use.
    - Users without experience can learn to use the system quickly.
- The user may switch quickly from one task to another and can interact with several different applications.
    - Information remains visible in its own window when attention is switched.
- Fast, full-screen interaction is possible with immediate access to anywhere on the screen

User-centred design

- The aim of this chapter is to sensitise software engineers to key issues underlying the design rather than the implementation of user interfaces
- User-centred design is an approach to UI design where the needs of the user are paramount and where the user is involved in the design process

- UI design always involves the development of prototype interfaces

USER INTERFACE DESIGN PROCESS



UI design principles

- UI design must take account of the needs, experience and capabilities of the system users
- Designers should be aware of people's physical and mental limitations (e.g. limited short-term memory) and should recognise that people make mistakes
- UI design principles underlie interface designs although not all principles are applicable to all designs

UI Design Principles

Principle	Description
User familiarity	The interface should use terms and concepts which are drawn from the experience of the people who will make most use of the system.
Consistency	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.

Minimal surprise	Users should never be surprised by the behaviour of a system.
Recoverability	The interface should include mechanisms to allow users to recover from errors.
User guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	The interface should provide appropriate interaction facilities for different types of system user.

Design principles

- User familiarity
    - The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.
- Consistency
    - The system should display an appropriate level of consistency. Commands and menus should have the same format, command punctuation should be similar, etc.
- Minimal surprise
    - If a command operates in a known way, the user should be able to predict the operation of comparable commands
- Recoverability
    - The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions, 'soft' deletes, etc.
- User guidance
    - Some user guidance such as help systems, on-line manuals, etc. should be supplied

- User diversity
    - Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available

User-system interaction

- Two problems must be addressed in interactive systems design
    - How should information from the user be provided to the computer system?
    - How should information from the computer system be presented to the user?
- User interaction and information presentation may be integrated through a coherent framework such as a user interface metaphor

Interaction styles

- Direct manipulation
- Menu selection
- Form fill-in
- Command language
- Natural language

Menu systems

- Users make a selection from a list of possibilities presented to them by the system
- The selection may be made by pointing and clicking with a mouse, using cursor keys or by typing the name of the selection
- May make use of simple-to-use terminals such as touchscreens

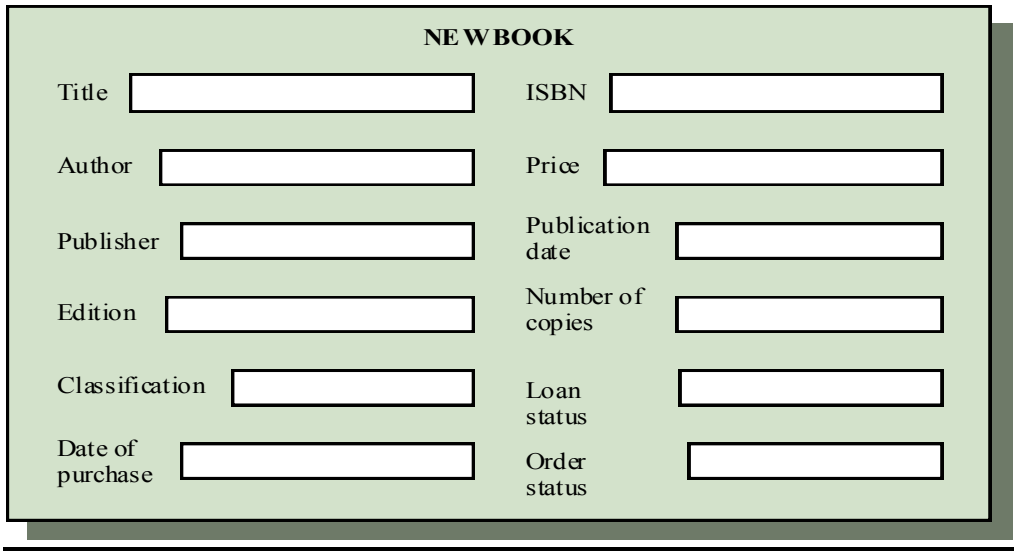
Advantages of menu systems

- Users need not remember command names as they are always presented with a list of valid commands
- Typing effort is minimal
- User errors are trapped by the interface
- Context-dependent help can be provided. The user's context is indicated by the current menu selection

Problems with menu systems

- Actions which involve logical conjunction (and) or disjunction (or) are awkward to represent
- Menu systems are best suited to presenting a small number of choices. If there are many choices, some menu structuring facility must be used
- Experienced users find menus slower than command language

Form-based interface



Command interfaces

- User types commands to give instructions to the system e.g. UNIX

- May be implemented using cheap terminals.
- Easy to process using compiler techniques
- Commands of arbitrary complexity can be created by command combination
- Concise interfaces requiring minimal typing can be created

Problems with command interfaces

- Users have to learn and remember a command language. Command interfaces are therefore unsuitable for occasional users
- Users make errors in command. An error detection and recovery system is required
- System interaction is through a keyboard so typing ability is required

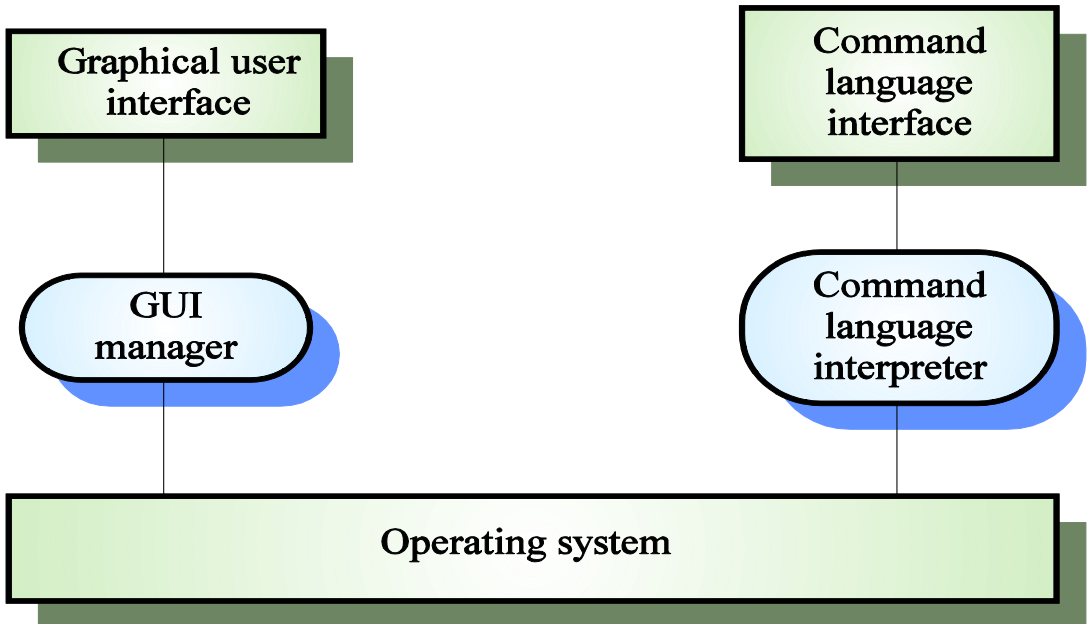
Command languages

- Often preferred by experienced users because they allow for faster interaction with the system
- Not suitable for casual or inexperienced users
- May be provided as an alternative to menu commands (keyboard shortcuts). In some cases, a command language interface and a menu-based interface are supported at the same time

Natural language interfaces

- The user types a command in a natural language. Generally, the vocabulary is limited and these systems are confined to specific application domains (e.g. timetable enquiries)
- NL processing technology is now good enough to make these interfaces effective for casual users but experienced users find that they require too much typing

Multiple user interfaces



Information presentation

- Static information
    - Initialised at the beginning of a session. It does not change during the session
    - May be either numeric or textual
- Dynamic information
    - Changes during a session and the changes must be communicated to the system user
    - May be either numeric or textual

Information display factors

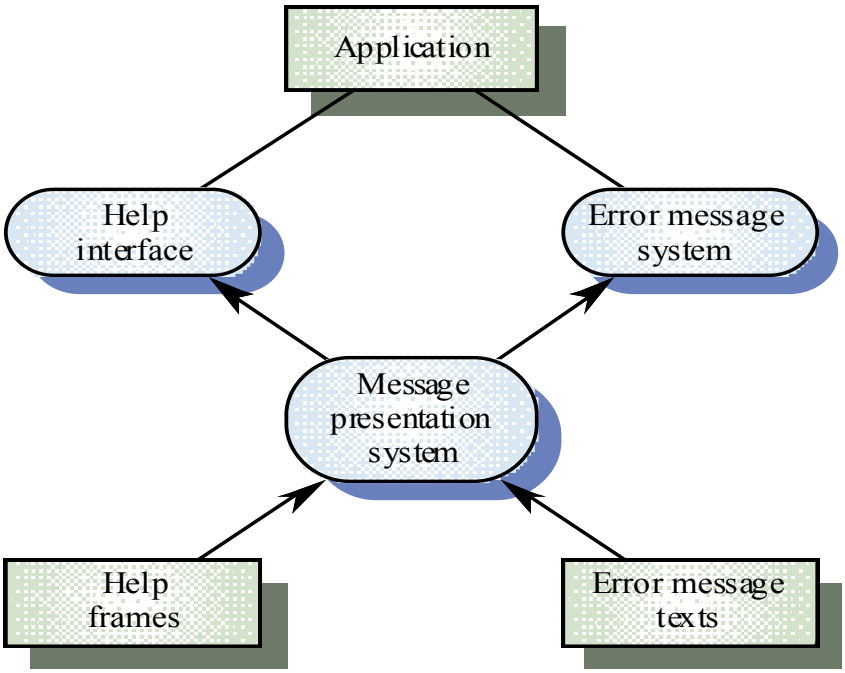
- Is the user interested in precise information or data relationships?
- How quickly do information values change? Must the change be indicated immediately?
- Must the user take some action in response to a change?
- Is there a direct manipulation interface?

- Is the information textual or numeric? Are relative values important?

Analogue vs. digital presentation

- Digital presentation
    - Compact - takes up little screen space
    - Precise values can be communicated
- Analogue presentation
    - Easier to get an 'at a glance' impression of a value
    - Possible to show relative values
    - Easier to see exceptional data values

Help and message system



Error messages

- Error message design is critically important. Poor error messages can mean that a user rejects rather than accepts a system
- Messages should be polite, concise, consistent and constructive
- The background and experience of users should be the determining factor in message design

Help system design

- *Help?* means ‘help I want information”
- *Help!* means “HELP. I'm in trouble”
- Both of these requirements have to be taken into account in help system design
- Different facilities in the help system may be required

Help information

- Should not simply be an on-line manual

- Screens or windows don't map well onto paper pages.
- The dynamic characteristics of the display can improve information presentation.
- People are not so good at reading screen as they are text.

Help system use

- Multiple entry points should be provided so that the user can get into the help system from different places.
- Some indication of where the user is positioned in the help system is valuable.
- Facilities should be provided to allow the user to navigate and traverse the help system.

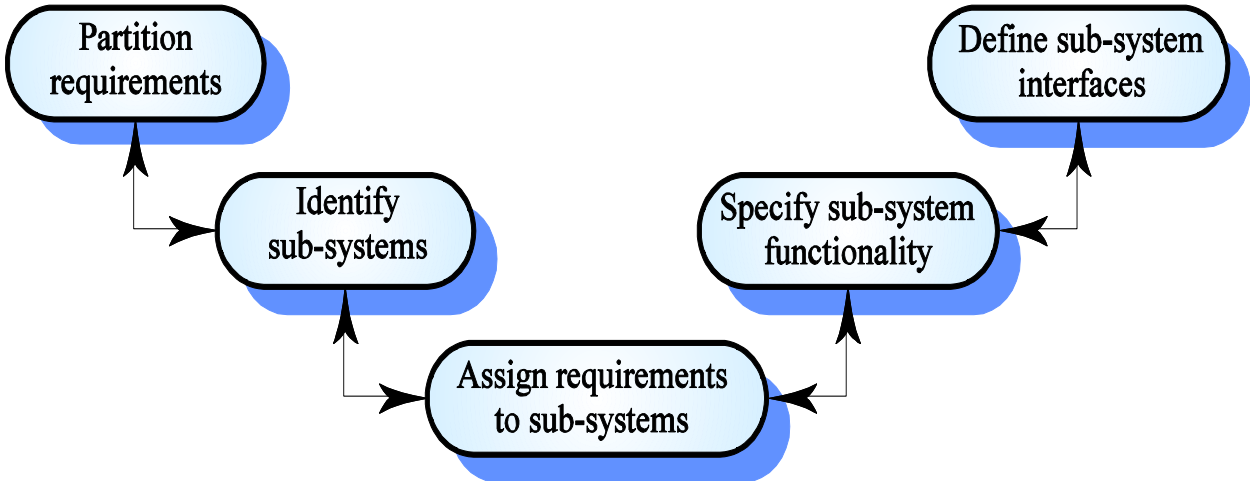
User documentation

- As well as on-line information, paper documentation should be supplied with a system
- Documentation should be designed for a range of users from inexperienced to experienced
- As well as manuals, other easy-to-use documentation such as a quick reference card may be provided

User interface evaluation

- Some evaluation of a user interface design should be carried out to assess its suitability
- Full scale evaluation is very expensive and impractical for most systems
- Ideally, an interface should be evaluated against a usability specification. However, it is rare for such specifications to be produced

THE SYSTEM DESIGN PROCESS



System design problems

- Requirements partitioning to hardware, software and human components may involve a lot of negotiation
- Difficult design problems are often assumed to be readily solved using software
- Hardware platforms may be inappropriate for software requirements so software must compensate for this

Sub-system development

- Typically parallel projects developing the hardware, software and communications
- May involve some COTS (Commercial Off-the-Shelf) systems procurement
- Lack of communication across implementation teams
- Bureaucratic and slow mechanism for proposing system changes means that the development schedule may be extended because of the need for rework

System integration

- The process of putting hardware, software and people together to make a system
- Should be tackled incrementally so that sub-systems are integrated one at a time
- Interface problems between sub-systems are usually found at this stage
- May be problems with uncoordinated deliveries of system components

3.8 REAL-TIME EXECUTIVES

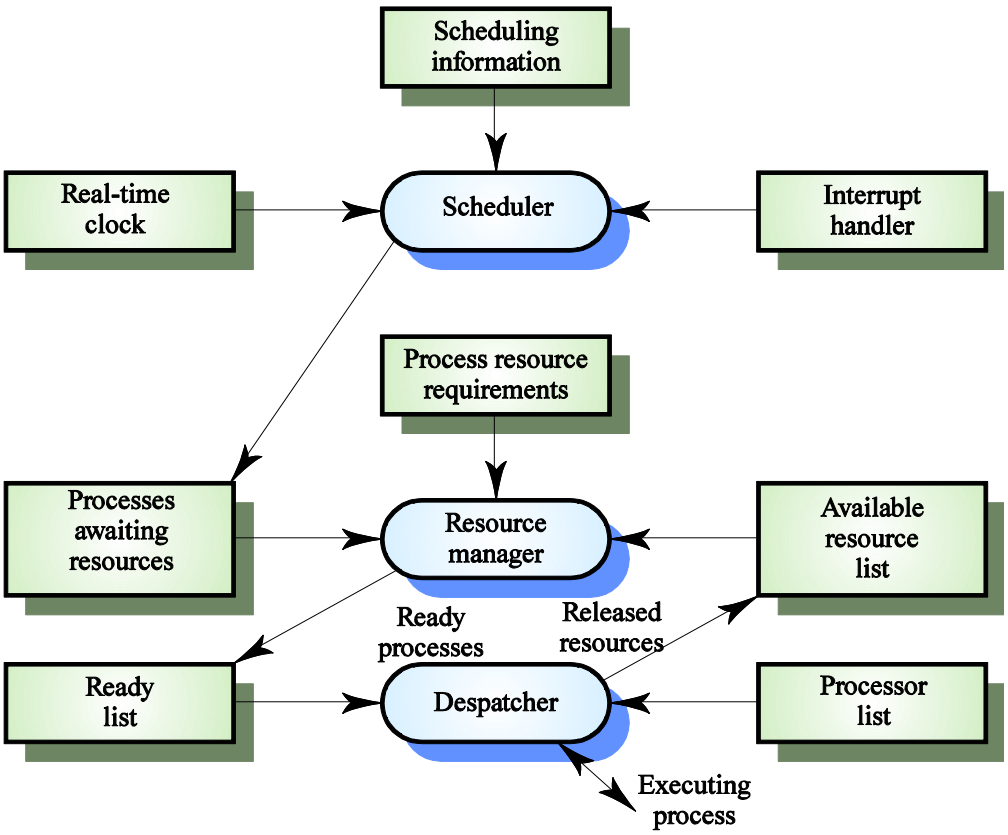
- Real-time executives are specialised operating systems which manage the processes in the RTS
- Responsible for process management and resource (processor and memory) allocation
- May be based on a standard RTE kernel which is used unchanged or modified for a particular application
- Does not include facilities such as file management

Executive components

- Real-time clock
    - Provides information for process scheduling.
- Interrupt handler
    - Manages aperiodic requests for service.
- Scheduler
    - Chooses the next process to be run.
- Resource manager
    - Allocates memory and processor resources.
- Despatcher

- Starts process execution.

Real-time executive components



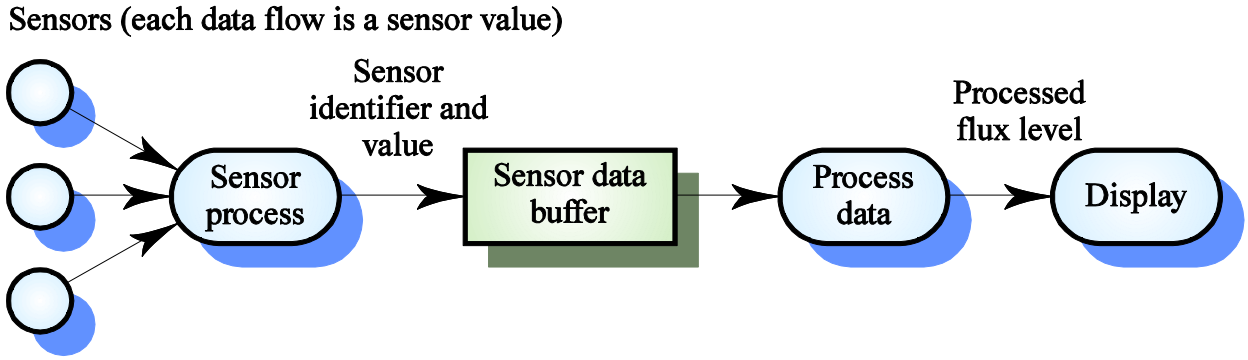
3.9 DATA ACQUISITION SYSTEMS

- Collect data from sensors for subsequent processing and analysis.
- Data collection processes and processing processes may have different periods and deadlines.
- Data collection may be faster than processing e.g. collecting information about an explosion.
- Circular or ring buffers are a mechanism for smoothing speed differences.

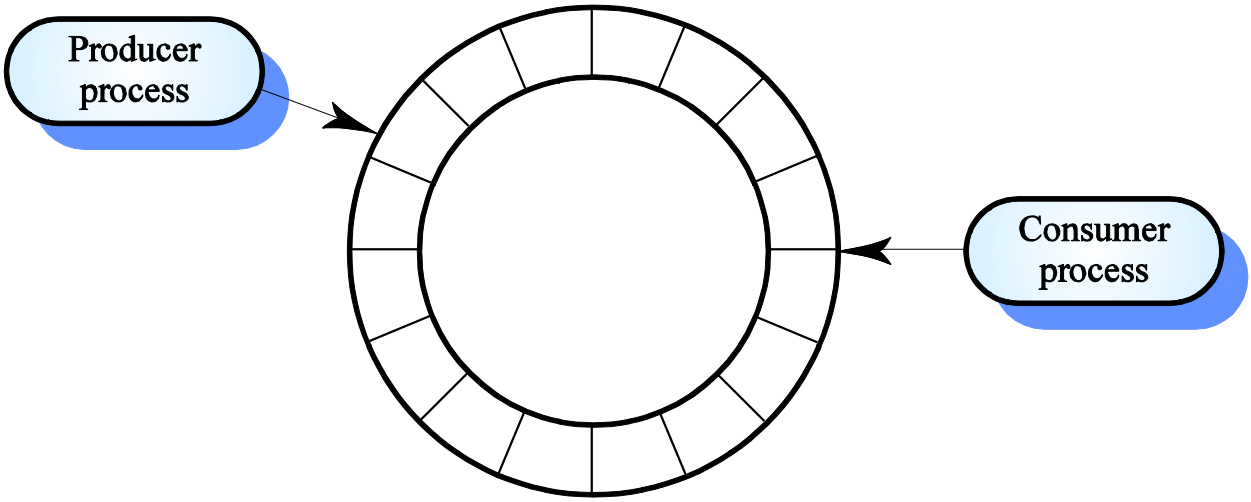
Reactor data collection

- A system collects data from a set of sensors monitoring the neutron flux from a nuclear reactor.
- Flux data is placed in a ring buffer for later processing.
- The ring buffer is itself implemented as a concurrent process so that the collection and processing processes may be synchronized

Reactor flux monitoring



A ring buffer



Mutual exclusion

- Producer processes collect data and add it to the buffer. Consumer processes take data from the buffer and make elements available
- Producer and consumer processes must be mutually excluded from accessing the same element.

DATA DESIGN

3.10 DATA FLOW DIAGRAMS (DFD)

DFD are directed graphs in which the nodes specify processing activities and the arcs specify data items transmitted between processing nodes.

Data flow diagram (DFD) serves two purposes

To provide an indication of how data are transformed as the move through the system and

To depict the functions that transforms that data flow.

Data flow diagram (DFD) –provides an indication of how data are transformed as they move through the system; also depicts functions that transform the data flow (a function is represented in a DFD using a process specification or PSPEC).

Functional Modeling and Information Flow (DFD):

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

Creating Data Flow Diagram:

- Level 0 data flow diagram should depict the system as a single bubble.
- Primary input and output should be carefully noted.
- Refinement should begin by consolidating candidate processes, data objects, and stored to be represented at the next level.
- Label all arrows with meaningful names
- Information flow must be maintained from one level to level
- Refine one bubble at a time
- Write a PSPEC (a "mini-spec" written using English or another natural language or a program design language) for each bubble in the final DFD.

UNIT IV

TESTING

4.1TAXONOMY OF SOFTWARE TESTING:

Software testing is a critical element of software quality assurance and represents the ultimate review of specification design and code generation.

Testing involves exercising the program using data like the real data processed by unexpected system outputs.

Testing may be carried out during the implementation phase to verify that the software behaves as intended by its designer and after the implementation is complete. This later phase checks conformance with requirements is complete.

Different kinds of testing use different types of data:

- statistical testing may be used to test the programs performance and reliability
- defect testing is intended to find the areas where the program does not conform to its specification

Testing Vs Debugging:

Defect testing a debugging are sometimes considered to be part of the same process. Infact they are quite different. Testing establishes the existence of defects. Debugging usually follows testing but they differs as to goals, methods and psychology

1. Testing starts with unknown conditions, uses predefined procedures, and has predictable outcomes only whether or not the program passes the test id unpredictable.

2. Testing can and should be planned designed and scheduled the procedures for and duration of debugging cannot be so constrained.

3. Testing is a demonstration of error or apparent correctness

4. Testing proves a programmers failure. Debugging is the programmer's vindication.

5. Testing as executed hold strives to predictable, dull, constrained, rigid and inhuman.
6. Much of the testing can be done without design knowledge.

7. Testing can often be done by an outsider. Debugging must be done by an insider.

8. Much of test execution and design can be automated. Automated debugging is still a dream.

Testing Objectives:

1. Testing is a process of executing a program with the intend of finding on error.

2. A good test case is one that high probability of finding an as yet undiscovered error.

3. A successful test is one that uncovers an as yet undiscovered error.

Testing Principles:

The various testing principle a listed below:

1. All tests should be traceable to customer requirements. The most serve defects are those that cause the program fail to meet its requirements.

2. Test should be planned long before testing begins. All tests can be planned and designed before any code has been generated.

3. Testing should begin “in the small” and progress towards testing “in the large”. The first tests planned and executed generally focus on the individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

4. Exhaustive testing is not possible.

5. To be more effective testing should be conducted by a third party.

Attributes of a good testing:

1. A good testing has a high probability of finding an error.

2. A good test is not redundant.

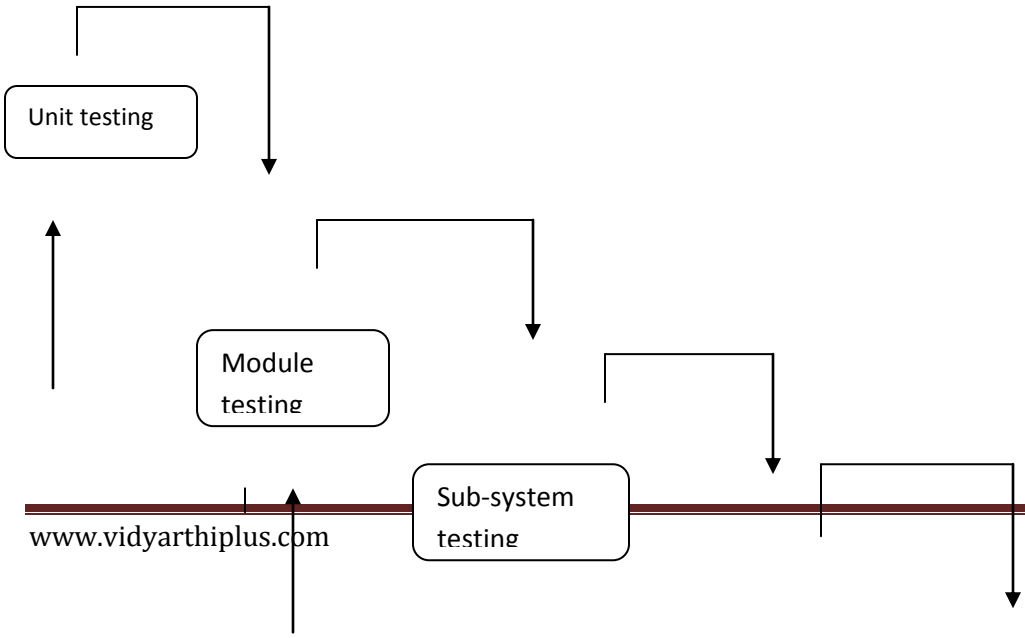
3. In a group of tests that have a similar intent, time and resource, the test that has the highest likelihood of uncovering a whole class of errors should be used.

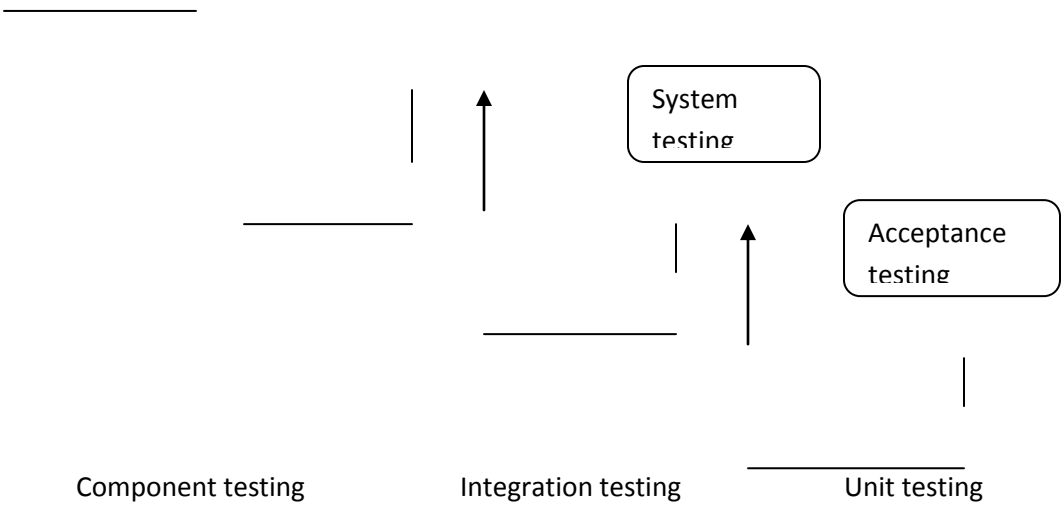
4. A good test should be neither too simple nor too complex. Each test should be executed separately.

The Testing Process

Systems should be tested as a single, monolithic unit only for small programs. Large systems are built out of sub-systems which are build out of modules which are compose of producers and functions. The testing process should therefore proceed in stages where testing is carried out incrementally in connection with system implementation.

The most widely used testing process consists of five stages





4.2 TYPES OF SOFTWARE TESTING

1. unit testing
2. module testing
3. sub-system testing
4. system testing
5. acceptance testing

1.) Unit Testing:

Here individual components are tested to ensure that they operate correctly. Each component is tested separately.

2.) Module Testing:

A module is collection of dependent components such as an object class, an abstract data type or some looser collection of procedures and functions. A module encapsulates related components so can be tested without other system modules.

3.) **Sub-System Testing:**

Here his phase involves testing collection of modules which have been integrated into sub-systems. Sub-systems be independently designed and implemented. The most common problems which arise in large s/w systems are sub-systems interface mismatches.

4.) System Testing:

The sub-systems are integrated to the entire system. The testing process is concerned with finding errors which results from anticipated interactions between sub-systems and system components. It is also concerned with validating that the system meets its functional and non-functional requirements.

5.) Acceptance testing:

This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system producer rather than stimulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the systems facilities do not really meet the user's needs or the system performance is unacceptable.

(i) Alpha testing:

Acceptance testing is sometimes called alpha testing. The alpha testing process continues until the system developer and the client agree with the deliver system is an acceptable implementation of the system requirements.

(ii) Beta testing:

When a system is to be marketed as a software product, a testing process called beta testing is often used. Beta testing involves delivering a system to a number of potential customers to agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors which may not have been anticipated by the system builders. After this feedback, the system is modified and either released or further beta testing or for general sale.

4.3 INTEGRATION TESTING

TOP - DOWN TESTING:

Top – down testing test the high level s of a system before testing its detailed components. The program is represented as a single abstract component with sub-components represented by stubs. Stubs have the same interface as the component but very limited functionality.

After the top level components have been tested, its sub-components are implemented and tested in the same way. This process continues recursively until the bottom level components are implemented. The whole system may then be completely tested.

Advantages of top down testing:

1. Unnoticed design errors may be detected at a early stage in the testing process. As these errors are mainly structural errors ,early detection means that can be corrected without undue costs

2. A limited working system is available at an early stage in the development. It demonstrates the feasibility of the system to management.

Disadvantage of top-down testing:

1. Program stubs simulating lower levels of system should be produced. If the component is a complex one, it may be impractical to produce a program stub which simulates it accurately.

2. Test output may be difficult to observe. In many systems, the higher levels of that system do not generate output but, to test these levels, they must be do so. The tester must create an artificial environment to generate the test results.

II. BOTTOM-UP TESTING

Bottom-up testing is the converses of the top-down testing. It involves testing the modules at the lower levels in the hierarchy, and then working up the hierarchy of modules until the final module is tested.

When using bottom-up testing, test drivers must be written to exercise the lower-level components. These test drivers must be written to exercise the lower-level components. These test drivers simulate the components environment and are valuable components in their own right.

If the components being tested are reusable components, The test drivers and test data should be distributed with the component.

Potential re-users can then run these tests to satisfy themselves that the component behaves as expected in their environment.

The advantages of bottom-up testing are the disadvantage of top-down testing and vice-versa.

Bottom-up testing is appropriate for object-oriented systems in that individual objects may be tested using their own drivers.

4.4Unit testing

It begins at the vortex of the spiral and concentrates on each unit of the s/w as implemented in source code.

Testing progresses by moving outward along the spiral to integration testing. Here the focus is on design and the construction of the software architecture.

Taking another turn outward on the spiral validation testing is encountered. Here requirements established as part of s/w requirements analysis are validated against the software that has been constructed.

Finally system testing is conducted. In this the software and other system elements are tested as a whole.

1. **Unit Testing:**

Unit testing focuses verification effort of the smallest unit of software design the software component or module.

Unit Testing considerations:

- The module interfaces is tested to ensure that information properly flows into and out of the program until under test.
- The local data structures is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- Finally, all error handling paths are tested.

4.5 Regression testing:

This testing is the re execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

Regression testing is the activity that helps to ensure that changes do not introduce unintended behavior or additional errors.

The regression test suite contains three different classes of test cases:

- A representation sample of tests that will exercise all software functions.
- Additional test that focus on software functions that are likely to be affected by the change.
- Tests that focus on the s/w components that have been changed.

4.6Validation testing:

At the end of integration testing, s/w is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests.

Validation succeeds when s/w functions in a manner that can be reasonably expected by the customer.

After each validation has been conducted, one of the two possible conditions exist:

1. The information or performance characteristic conform to specification and are accepted.
2. A derivation from specifications is uncovered and a deficiency list is created.

It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

Configuration review:

This is an important element of the validation process. The intent of the review is to ensure that all elements of the s/w configuration have been properly developed.

Alpha and Beta testing:

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.

The alpha and beta tests have been discussed previously.

4.7SYSTEM TESTING AND DEBUGGING:

S/w is incorporated with other system elements like hardware, people, information and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by s/w engineers.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

Types of system testing:

1. Recovery testing
2. Security testing
3. Stress testing
4. Performance testing

Recovery testing:

Many computer based systems must recover from faults and resume processing within a prespecified time.

Recovery testing is a system test that forces the s/w to fail in a variety of ways and verifies that recovery is properly performed.

If recovery requires human intervention the mean time to repair is evaluated to determine whether it is within acceptable limits.

Security testing:

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

During security testing, the tester plays the role of the individual who desires to penetrate the system.

Stress testing:

This executes a system in a manner that demands resources in abnormal quantity, frequency or volume.

Essentially, the tester attempts to break the program.

A variation of stress testing is a techniques called sensitivity testing. In some situations, a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.

Sensitivity testing attempts to uncover data combinations within valid input classes that may be cause instability or improper processing.

Performance testing

For real time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of s/w within the context of an integrated system.

Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed.

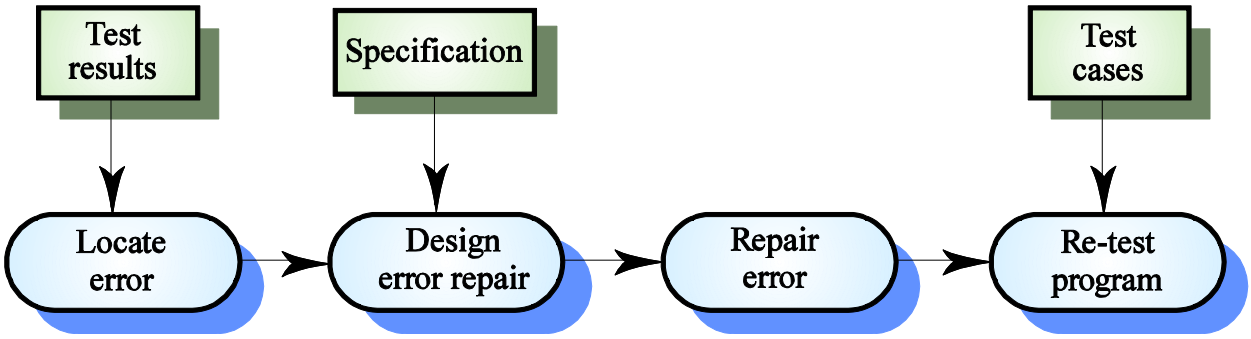
Performance tests are often couple with stress testing and usually require both hardware and s/w instrumentation.

TESTING AND DEBUGGING

- Defect testing and debugging are distinct processes
- Verification and validation is concerned with establishing the existence of defects in a program
- Debugging is concerned with locating and repairing these errors

- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error

THE DEBUGGING PROCESS



4.8 TEST COVERAGE BASED ON DATA FLOW MECHANISM:

White box testing is called as glass box testing. It is a test case design method that uses the control structure of the procedural design to the derive test cases.

Benefits of white box testing:

- **Focused testing**: The programmer can test the program in pieces. It's much easier to give an individual suspect module a through workout in glass box testing than in black box testing.

- **Testing coverage**: The programmer can also find out which parts of the program are exercised by any test. It is possible to find out which lines of code, which branches, or which paths haven't yet been tested.

- **Control flow**: The programmer knows what the program is supported to do next, as a function of its current state.

- **Data integrity**: The programmer knows which parts of the program modify any item of data. By tracking a data item through the system.

- **Internal boundaries**: The programmer can see internal boundaries in the code that are completely invisible to the outside tester.

- **Algorithmic specific**: The programmer can apply standard numerical analysis techniques to predict the results.

Various white box testing techniques:

1. 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 and use this measure as a guide for defining a basis set of execution paths.

Flow graph notation:

Flow graph is a simple notation for the representation of control flow. Each structured construct has a corresponding flow graph symbol.

Flow graph node: Represents one or more procedural statements.

Edges or links: Represent flow control.

Regions: These are areas bounded by edges and nodes.

Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

Cyclomatic complexity:

Cyclomatic complexity is a software metric that provide a quantitative measure of the logical complexity of a program.

The value computed for cyclomatic defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Thee ways of computing cyclomatic complexity:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.

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

   V (G) = E-N+2

   E is the number of the flow graph edges; N is the number of flow graph nodes.

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

   V (G) = P+1

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

Deriving test cases:

The basis path testing method can be applied to procedural design or to source code.

Steps to derive the basis test:

1. Using the design or code as a foundation draw a corresponding flow graph. A flow graph is created using the symbols and construction rules.

2. Determine the cyclomatic complexity of the resultant flow graph. V (G) is determined by applying the above algorithms.

3. Determine a basis set of linearly independent paths. The value of V (G) provides the number of linearly independent paths through the program control structure.

II. CONDITION TESTING

Condition testing is a test case design method that exercises the logical conditions contained in a program module.

The condition testing method focuses on testing each condition in the program.

Advantage of condition testing:

1. Measurement of the test coverage of a conditional is simple.

2. The test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

- ***Branch testing***: This is the simplest condition testing strategy. For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once.

- ***Domain testing***: This requires three or four tests to be derived for a relational for a relational expression.

- ***BRO*** (branch and relational operator) testing: This technique guarantees the detection of branch and relational operator errors in a condition provided that all Boolean variable and relational operators in condition occur only once.

III. DATA FLOW TESTING:

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

For a statement with S as its statement number,

DEF(S) ={X} statement S contains a definition of {X}

USE(S) ={X} statement S contains a use of {X}

If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S.

A definition use (DU) chain of variable X is of the form {X, S, S'] where S and S' are statement numbers, X is in DEF(S) and USE(S') and the definition of X in statements S is live at statement S'.

One simple data flow testing strategy is to require that every DU chain be covered at least once.

Data flow testing strategies are useful for selecting test paths for a program containing nested if and loop statements.

Since the statements in a program are related to each other according to the definitions and uses of variable the data flow testing approach is effective for error detection.

Problem:

Measuring test coverage and selecting test paths for data flow testing are more difficult.

IV. LOOP TESTING:

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

Different classes of loops:

1. Simple loops
2. Nested loops
3. Concatenated loops
4. unstructured loops

Simple loops:

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.

Nested loops: The number of possible tests would grow geometrically as the level of nesting increases.

Methods to reduce the number of tests:

1. Start at the innermost loop.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values.

3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum value and other nested loops to typical values.

Concatenated loops:

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop2.

Unstructured loops:

Whenever possible, this class of loops would be redesigned to reflect the use of the structured programming constructs.

4.9BLACK BOX TESTING:

Black box testing is also called as behavioral testing. This focuses on the functional requirements of the s/w. Black box testing enables the s/w engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

Errors found by black box testing:

1. incorrect or missing functions
2. interface errors
3. errors in data structures or external data base access.

4. behavior or performance errors.

5. initialization and termination errors.

Various black box testing method:

1. Equivalent partitioning
2. boundary value analysis
3. comparison testing
4. orthogonal array testing

1. EQUIVALENCE PARTITIONING:

It is a black box testing method that divides the inputs 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.

The input data to a program usually fall into number of different classes. These classes have common characteristics, for example positive numbers, negative numbers strings without blanks

and so on. Programs normally behave in a comparable way for all members of a class. Because of this equivalent behavior, these classes are sometimes called equivalent partitions or domains.

A systematic approach to defect testing is based on identifying a set of equivalence partitions which must be handled by a program.

Guidelines for defining equivalence classes:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set one valid and one invalid equivalence class are defined.

4. If an input condition is Boolean, one valid and one invalid class are defined.

Test cases for each input domain data item can be developed and executed by applying the guidelines for the derivation of equivalence classes.

III. COMPARISON TESTING:

When reliability of software is absolutely critical, redundant hardware and s/w are often used to minimize the possibility of error. In such situations, each version can be tested with the same test data to ensure that all provide identical output. Those independent versions from the basis of a black box testing technique called comparison testing.

If the output from the each version is the same, it is assumed that all implementations are correct. If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference.

Problem in comparison testing:

1. Comparison testing is not foolproof. If the specification from which all versions have been developed is in error, all versions will likely reflect the error.

2. If each of the independent versions produces identical but incorrect results, condition testing will fail to detect the error.

IV. ORTHOGONAL ARRAY TESTING:

Orthogonal testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding errors associated with regions faults an error category associated with faulty logic within a software component.

When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. The L9 orthogonal array has a balancing property. That is test cases are dispersed uniformly throughout the test domain.

The orthogonal array testing approach enables us to provide good test coverage with fewer test case than the exhaustive strategy.

4.10 BOUNDARY VALUE ANALYSIS:

A great number of errors tend to occur at the boundaries of the input domain rather than in the center. So boundary value analysis (BVA) derives test cases from the output domain as well.

Guidelines for boundary value analysis:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers.

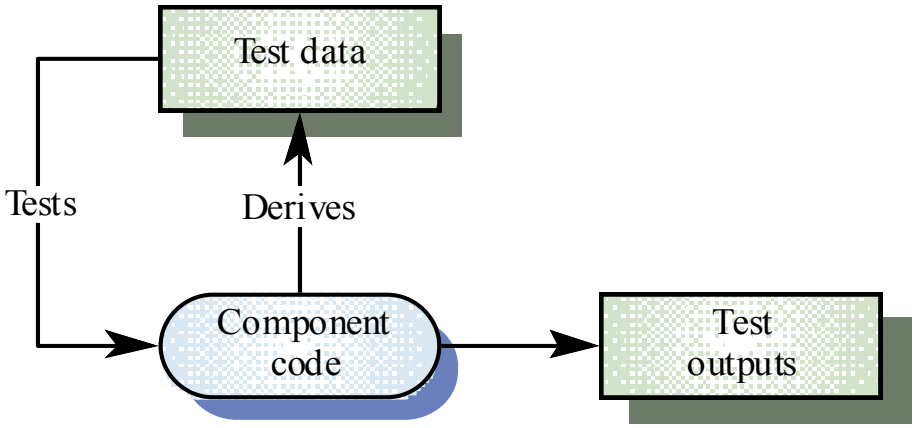
3. Apply guidelines 1 and 2 to output conditions.

4. If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary.

4.11 STRUCTURAL TESTING

- Sometime called white-box testing
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases

- Objective is to exercise all program statements
  (not all path combinations)



4.12 SOFTWARE IMPLEMENTATION TECHNIQUES

Testing tool categories:

S/w quality engineering defines the following testing categories:

- Data acquisition- tools that acquire data to be used during testing.
- Static measurement- tools that analyze source code without executing test cases.
- Dynamic measurement- tools that analyze source code during execution.
- Simulation – tools that simulate function of hardware or other externals.

- Test management – tools that assist in the planning, development, and control of testing.

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

UNIT V

SOFTWARE PROJECT MANAGEMENT

5.1 SOFTWARE COST ESTIMATION:

Predicting the resources required for a software development process.

SOFTWARE COST COMPONENTS:

*Hardware and software costs

*Travel and training costs

*Efforts costs

*Salaries of engineers involved in the project

*Social and insurance costs

*Effort costs must take overheads into account

*Costs of shared facilities

COSTING AND PRICING:

*Estimates are made to discover the cost, to the developer of producing a software system.

*There is not a simple relationship between the development ost and the price charged to the customer.

PROGRAMMER PRODUCTIVITY:

*A measure of the rate at which individual engineers involved in software development produce software and associated documentation.

* Not quality oriented although quality assurance is a factor in productivity assessment

PRODUCTIVITY MEASURES:

*Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions etc.,

* Function related measures based on an estimate of the functionality of the delivered software.

MEASUREMENT PROBLEM:

*Estimating the size of the measure.

*Estimating the total number of programmer months which have elapsed.

LINES OF CODE:

*What's a line of code?

The measures was first proposed when programs were typed on cards with one line per card.

	Analysis	Design	Coding	Testing	Documentation
Assembly code High – level Language	3 weeks 3 weeks	5 weeks 5 weeks	8 weeks 8 weeks	10 weeks 6 weeks	2 weeks 2 weeks
	Size	Effort	Productivity		
Assembly High – level language	5000 lines 1500 lines	28 weeks 20 weeks	714 lines / months 300 lines / months		

*How does this correspond to statements as in java which can span several lines or where there can be several statements.

*What programs should be counted as part of the system?

PRODUCTIVITY COMPARISONS:

*The lower level the language, the more productive the programmer.

*The same functionality takes more code to implement in a lower level language than in a high level language.

*The more verbose the programmer, the higher the productivity.

5.2 FUNCTION POINTS:

FUNCTION POINTS:

*Based on a combination of program characteristics

*External inputs and outputs

*User interactions

*External interfaces

*Files used by the system

*Function point count modified by the complexity of the project

*FP's can be used to estimate LOC depending on the average number of LOC per FP for a given language.

*LOC = AVC * number of function points

*Automatic function point counting is impossible.

5.3 The COCOMO model:

- An empirical model based on project experience
- Well documented independent model which is not tied to a specific software vendor.

Boehm introduces a hierarchy of software estimation models bearing the generic name COCOMO for constructive cost model.

*The basic COCOMO model is static single valued model that computes software development effort as a function of program size expressed in estimated lines of code.

*The intermediate COCOMO model computes software development efforts as a function of program size and a set of cost drivers that includes subjective assessments of the product, hardware, personnel and project attributes.

COCOMO 2 LEVELS:

*COCOMO 2 model is a 3 level model that allows increasingly detailed estimates to be prepared as development progresses.

*Early prototyping level.

*Estimates based on object points and a simple formula is used for effort estimation.

*Early design level.

MODEL 3:

The advanced COCOMO model incorporates all characteristics of the intermediate version with an assessment of the driver's impact on each step of the software engineering process.

The COCOMO models are defined for the three classes of software projects: Using Boehm's terminology these are:

1. Organic mode: Relatively small, simple software projects in which small team with good application experience work to set of less than rigid requirements.

2. Semi- detached mode: An intermediate s/w projects in which teams with mixed experience levels must meet a blend of rigid and less than rigid requirements

3. Embedded mode: a s/w projects must be developed within a set of tight h/w, s/w.

(i) product attributes

* required s/w reliability

* size of application data

* complexity of product.

(ii) Hardware attributes:

- Run-time performance constraints.
- Memory constraints
- Required turn around time

(iii) personal attributes

- analyst capability
- software engineer capability
- application experience
- virtual machine experience

(iv)project attributes:

- use of software tools
- application of software engineering methods
- required developed schedule

Each of the fifteen attributes is rated on a six point scale that ranges from very low to extra high. Based on this rating an effort multiplier is determined from the tables.

Early prototyping level:

- supports prototyping projects and projects where there is extensive reuse
- based on standard estimates of developer productivity
- takes case tool use into account

Object point productivity:

Early design level:

- estimates can be made a after the requirements have been agreed

- based on standard formula for algorithmic models
- PM=A?Size B ? M + PMm where
  M=PERS ? RCPX ? RUSE ? PDIF ? PREX ? FCIL ? SCED

Multipliers:

Multipliers reflect the capability of the developers, he non-functional requirements, the familiar with the development platform, etc.

- RCPF – producer reliability and complexity
- RUSE – the reuse required
- PDIF – platform difficulty
- PREX – personnel experience
- PERS – personnel capability
- SCED – required schedule
- FCIL – the team support facilities
- PM reflects the amount of automatically generated code

Post-architectural level:

- Uses same facilities as early design estimates
- Estimates of size is adjusted to take into account
- Requirements volatility. rework required to support change
- Extend of possible reuse. Reuse is non-linear and has associated costs so this is not a simple reduction in LOC
- ESLOC = ASLOC ? (AA + SU +0.4DM + 0.3CM + 0.31M)/100
- ESLOC is equivalent number of lines of new code ASLOC is the number of lines of reusable code which must be modified, DM is the % of design modified , CM is the % of the code that is modified.

The Exponent Term:

- This depends on five scale factors. Their sum/100 is added to 1.01
- Example
- Precedent ness – new project four
- Development flexibility – no client involvement – very high one
- Architecture / risk resolution – no risk analysis
- Team cohesion – new team nominal three
- Process maturity – some control – nominal three

Multipliers:

- product attributes concerned with required characteristics of the software products being developed.
- Computer attributes constraints imposed on the software by the hardware platform.
- Project attributes concerned with the particular characteristics of the software development projects.

Projects Planning:

- Algorithmic cost models provide a basis for projects planning as they allow alternative strategies to be compared.
- Embedded space craft system

  must be reliable

  must minimize weight

  multipliers on reliability and computer constraints.

- Cost components
- Target hardware
- Development platform
- Effort required

Project duration and staffing:

- As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required
- Calendar time can be estimated using COCOMO-2 formula
- PM is effort computation and B is the exponent computed as discussed above.
- The time required is independent of the number of people working o the project.

Staffing requirements:

- Staff required cannot be computed by dividing the development time by required schedule
- The number of people working on a project varies depending on the phase of the project
- The more people who work on project more total effort is usually required

Key Points:

- Factors affecting productivity include individual aptitude domain experience the development project, the project size, tool support
- Software may be priced to gain a contract and functionality
- Algorithmic cost estimation is difficult because of the need to estimate attributes
- The time to complete a project is not proportional to the number of people working on the project

Required level of reliability:

- Software reliability can be defined as the probability that a program will perform a required function.

Level of technology:

- The better level of technology and higher productivity so lower cost because the time taken to complete the project would be less and the lesser number of resources will be used.

5.4 Delphi cost estimation:

- This technique was developed at Rand Corp. in 1948 to gain expert consensus without introducing the adverse side affects of group meetings.
- The Delphi technique can be adopted to software cost estimation in the following manner:
    1. A coordinator provides each estimator with the system definition document and a form for recording their estimate.
    2. The estimator study and complete their estimation anonymously. They ask questions to the coordinator but do not discuss with one another.
    3. The coordinator makes summary and includes any unused rationales notes by the estimators.
    4. Estimators complete another estimation, again anonymously, using the results of the previous estimates. The estimators whose estimates differ sharply from the group may be asked justify their answer, anonymously.
    5. The process is iterated for as many rounds as required. No group discussion is allowed during the entire process.
    - ➢ It is possible that after several rounds of estimates will not lead to a consensus estimate. In this case, the coordinator must discuss the issues involved with each estimator to determine the reasons for the differences.
    - ➢ The coordinator may have to gather additional information and present it to the estimators in order to resolve the differences in viewpoint.

Work Breakdown Structures:

- ✓ This is a bottom-up estimation tool. A work breakdown structure is a hierarchical chart that accounts for the individual parts of a system. A WBS chart can indicate either product hierarchy or process hierarchy.
- ✓ This product hierarchy helps in identifying the manner in which the components are interconnected.

The advantages of WBS technique are in identifying and accounting for various process and product factors, in making explicit exactly which cost are included in the estimate.

5.5 SOFTWARE PROJECT SCHEDULING:

Project scheduling and tracking:

- The scheduling is the process of building and monitoring schedules for software development systems, many engineering tasks need to occur in parallel with one another to complete the project on time. The output from one task often determines when another may begin. It is difficult to ensure that a team is working on the most appropriate tasks without building a detailed schedule and sticking to it.

Software Project scheduling Principle:

- Compartmentalization:
- Interdependency
- Time allocation
- Effort validation
- Define responsibilities
- Defined outcomes
- Defined milestones

Relationship between people and effort:

*adding people to the project after it is behind schedule often causes the schedule to slip further.

* Relationship between number of peoples on a project and overall productivity is not linear.

* The main reason for using more than one person on a project are to get the job done more rapidly and to improve s/w quality.

Project effort distribution:

Generally accepted guidelines are

1. 2-3 % planning.
2. 10-25% requirement analysis
3. 22- 25% designing
4. 15 – 25% coding
5. 30-40% Testing and debugging.

Software project types:

*Concept development- to explore new business concept

*New application development- New product requested by the customer

*Application enhancement: Modifications to function, performance are interfaces

*Application maintenance: Correcting adopting, or extending existing s/w.

*Re-engineering: Rebuilding all of a legacy system.

Software process degree of Rigor:

*Casual: All framework activities applied, only minimum task set required.

*Structured: All framework and umbrella activities applied.(SQA,SCM)

*Strict: Full process and umbrella activities.

*Quick reaction: Emergency situation process frame work required.

Rigor adaptation criteria:

- Size of project
- Number of potential uses
- Mission criticality

- Application longevity
- Requirements stability
- Customer/developer communication
- Maturity of applicable technology
- Performance constraints
- Project staffing
- Reengineering factors

Concept development task:

- Concept scooping- determine overall project scope.
- Preliminary concept planning- establishes development team's ability to undertake the proposed work.
- Technology risk assessment- Evaluate the risk assessed with the technology
- Proof of concept : Demonstrate the feasibility of the technology
- Concept implementation : Concept represented in the form that can be sell to a customer.
- Customer reaction to the concept: Solicits feedback on new technology from customer.

Scheduling:

*Scheduling tools should be used to schedule any non trivial project.

* PERT and CPM – quantitative techniques that allow s/w planners to identify the chain of dependent task in the project work breakdown structure that determines the project duration.

* Time line or GANTT chart : enables s/w planners to determine what task will be needed to conducted at a given point in the given time.

*Time boxing: Practice of deciding a prior, the fixed amount of time that can be spend on each task.

5.6 Error tracking:

- Allows comparison of current works to the past projects and provides a quantitative indication on the works so far completed.
- The more quantitative approach to project tracking and control, the more likely problems can be anticipated and dealt.

5.7 MEASURES AND MEASUREMENTS

Productivity measures

- Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.
- Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure

Measurement problems

- Estimating the size of the measure
- Estimating the total number of programmer months which have elapsed
- Estimating contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate

Lines of code

- What's a line of code?
    - The measure was first proposed when programs were typed on cards with one line per card
    - How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line
- What programs should be counted as part of the system?
- Assumes linear relationship between system size and volume of documentation

Function points

- Based on a combination of program characteristics
    - external inputs and outputs
    - user interactions
    - external interfaces
    - files used by the system
- A weight is associated with each of these

- The function point count is computed by multiplying each raw count by the weight and summing all values

Object points

- Object points are an alternative function-related measure to function points when 4Gls or similar languages are used for development
- Object points are NOT the same as object classes
- The number of object points in a program is a weighted estimate of
    - The number of separate screens that are displayed
    - The number of reports that are produced by the system
    - The number of 3GL modules that must be developed to supplement the 4GL code

Productivity estimates

- Real-time embedded systems, 40-160 LOC/P-month
- Systems programs , 150-400 LOC/P-month
- Commercial applications, 200-800 LOC/P-month
- In object points, productivity has been measured between 4 and 50 object points/month depending on tool support and developer capability

5.8 ZIPF's LAW

Zipf's law, an empirical law formulated using mathematical statistics, refers to the fact that many types of data studied in the physical and social sciences can be approximated with a Zipfian distribution.

Zipf's law is most easily observed by plotting the data on a log-log graph, with the axes being log(rank order) and log(frequency). For example, the word "the" (as described above) would appear at $x = \log(1)$, $y = \log(69971)$. The data conform to Zipf's law to the extent that the plot is linear.

Formally, let:

- N be the number of elements;
- k be their rank;
- s be the value of the exponent characterizing the distribution.

Zipf's law then predicts that out of a population of *N* elements, the frequency of elements of rank *k*, *f*(*k*;*s*,*N*), is:

$$f(k;s,N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)}.$$

Zipf's law holds if the number of occurrences of each element are independent and identically distributed random variables with power law distribution $p(f) = \alpha f^{-1-1/s}$.

In the example of the frequency of words in the English language, *N* is the number of words in the English language and, if we use the classic version of Zipf's law, the exponent *s* is

1. *f*(*k*; *s*,*N*) will then be the fraction of the time the *k*th most common word occurs.

The law may also be written:

$$f(k;s,N) = \frac{1}{k^s H_{N,s}}$$

where $H_{N,s}$ is the *N*th generalized harmonic number.

The simplest case of Zipf's law is a "$^1/_f$ function". Given a set of Zipfian distributed frequencies, sorted from most common to least common, the second most common frequency will occur ½ as often as the first. The third most common frequency will occur ⅓ as often as the first. The n^{th} most common frequency will occur $^1/_n$ as often as the first. However, this cannot hold exactly, because items must occur an integer number of times; there cannot be 2.5 occurrences of a word. Nevertheless, over fairly wide ranges, and to a fairly good approximation, many natural phenomena obey Zipf's law.

Mathematically, it is impossible for the classic version of Zipf's law to hold exactly if there are infinitely many words in a language, since the sum of all relative frequencies in the denominator above is equal to the harmonic series and therefore:

$$\sum_{n=1}^{\infty} \frac{1}{n} = \infty.$$

In English, the word frequencies have a very heavy-tailed distribution and can therefore be modeled reasonably well by a Zipf distribution with an *s* close to 1.

As long as the exponent *s* exceeds 1, it is possible for such a law to hold with infinitely many words, since if $s > 1$ then

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} < \infty.$$

where ζ is Riemann's zeta function

5.9 EARNED VALUE ANALYSIS

- "Earned Value Analysis" is:
    - an industry standard way to:
        - measure a project's progress,
        - forecast its completion date and final cost, and
        - provide schedule and budget variances along the way.
- By integrating three measurements, it provides consistent, numerical indicators with which you can evaluate and compare projects.

What's more Important

- Knowing where you are on schedule?
- Knowing where you are on budget?
- Knowing where you are on work accomplished?

EVA Integrates All Three

- It compares the PLANNED amount of work with what has actually been COMPLETED, to determine if *COST , SCHEDULE, and WORK ACCOMPLISHED* are progressing as planned.
- Work is "Earned" or credited as it is completed.

Earned Value needed because

- Different measures of progress for different types of tasks
- Need to "roll up" progress of many tasks into an overall project status
- Need for a uniform unit of measure (dollars or work-hours).
- Provides an "Early Warning" signal for prompt corrective action.
    - Bad news does not age well.
    - Still time to recover

- Timely request for additional funds

Earned Value Definitions

- BCWS: "Budgeted Cost of Work Scheduled"
- *Planned cost of the total amount of work scheduled to be performed by the milestone date.*
- ACWP: "Actual Cost of Work Performed"
- *Cost incurred to accomplish the work that has been done to date.*
- BCWP: Budgeted Cost of Work Performed
    - *The planned (not actual) cost to complete the work that has been done.*

Some Derived Metrics

- SV: Schedule Variance (BCWP-BCWS)
    - A comparison of amount of work performed during a given period of time to what was scheduled to be performed.
    - A negative variance means the project is behind schedule
- CV: Cost Variance (BCWP-ACWP)
    - A comparison of the budgeted cost of work performed with actual cost.
    - A negative variance means the project is over budget.

Schedule Variance & Cost Variance

Schedule Variance = BCWP-BCWS

Cost Variance = BCWP-ACWP

Some More Derived Metrics

- SPI: Schedule Performance Index SPI=BCWP/BCWS
  SPI<1 means project is behind schedule
- CPI: Cost Performance Index CPI= BCWP/ACWP
  CPI<1 means project is over budget
- CSI: Cost Schedule Index (CSI=CPI x SPI)

  The further CSI is from 1.0, the less likely project recovery becomes.

Performance Metrics

SPI: BCWP/BCWS

CPI: BCWP/ACWP

CSI: SPI x CPI

Value of Earned Value

- Schedule Status Reporting
- Cost Status Reporting
- Forecasting

Requirements of Earned Value

- Proper WBS Design
- Baseline Budget Control Accounts
- Baseline Schedule
- Work measurement by Control Account
    - work-hours, dollars, units, etc.
- Good Project Management Practices

Shortcomings of Earned Value

- Quantifying/measuring work progress can be difficult.
- Time required for data measurement, input, and manipulation can be considerable.

5.10 CONFIGURATION MANAGEMENT

- New versions of software systems are created as they change
    - For different machines/OS
    - Offering different functionality
    - Tailored for particular user requirements
- Configuration management is concerned with managing evolving software systems
    - System change is a team activity
    - CM aims to control the costs and effort involved in making changes to a system

- Involves the development and application of procedures and standards to manage an evolving software product
- May be seen as part of a more general quality management process
- When released to CM, software systems are sometimes called *baselines* as they are a starting point for further development

CM standards

- CM should always be based on a set of standards which are applied within an organisation
- Standards should define how items are identified, how changes are controlled and how new versions are managed
- Standards may be based on external CM standards (e.g. IEEE standard for CM)
- Existing standards are based on a waterfall process model - new standards are needed for evolutionary development

Configuration management planning

- All products of the software process may have to be managed
    - Specifications
    - Designs
    - Programs
    - Test data
    - User manuals
- Thousands of separate documents are generated for a large software system

CM planning

- Starts during the early phases of the project
- Must define the documents or document classes which are to be managed (Formal documents)

- Documents which might be required for future system maintenance should be identified and specified as managed documents

The CM plan

- Defines the types of documents to be managed and a document naming scheme
- Defines who takes responsibility for the CM procedures and creation of baselines
- Defines policies for change control and version management
- Defines the CM records which must be maintained

- Describes the tools which should be used to assist the CM process and any limitations on their use
- Defines the process of tool use
- Defines the CM database used to record configuration information
- May include information such as the CM of external software, process auditing, etc.

Configuration item identification

- Large projects typically produce thousands of documents which must be uniquely identified
- Some of these documents must be maintained for the lifetime of the software
- Document naming scheme should be defined so that related documents have related names.
- A hierarchical scheme with multi-level names is probably the most flexible approach

The configuration database

- All CM information should be maintained in a configuration database
- This should allow queries about configurations to be answered
    - Who has a particular system version?

- What platform is required for a particular version?
- What versions are affected by a change to component X?
- How many reported faults in version T?

- The CM database should preferably be linked to the software being managed

Change management

- Software systems are subject to continual change requests
  - From users
  - From developers
  - From market forces
- Change management is concerned with keeping managing of these changes and ensuring that they are implemented in the most cost-effective way

Change request form

- Definition of change request form is part of the CM planning process
- Records change required, suggestor of change, reason why change was suggested and urgency of change(from requestor of the change)
- Records change evaluation, impact analysis, change cost and recommendations (System maintenance staff)

Change Request Form
Project: Proteus/PCL-Tools **Number:** 23/94 **Change requester:** I. Sommerville **Date:** 1/12/98 **Requested change:** When a component is selected from the structure, display the name of the file where it is stored. **Change analyser:** G. Dean **Analysis date:** 10/12/98 **Components affected:** Display-Icon.Select, Display-Icon.Display **Associated components:** FileTable **Change assessment:** Relatively simple to implement as a file name table is available. Requires the design and implementation of a display field. No changes to associated components are required.

Change priority: Low	
Change implementation:	
Estimated effort: 0.5 days	
Date to CCB: 15/12/98	**CCB decision date:** 1/2/99
CCB decision: Accept change. Change to be implemented in Release 2.1.	
Change implementor:	**Date of change:**
Date submitted to QA:	**QA decision:**
Date submitted to CM:	
Comments	

Change tracking tools

- A major problem in change management is tracking change status
- Change tracking tools keep track the status of each change request and automatically ensure
  that change requests are sent to the right people at the right time.
- Integrated with E-mail systems allowing electronic change request distribution

Change control board

- Changes should be reviewed by an external group who decide whether or not they are cost-effective from a strategic and organizational viewpoint rather than a technical viewpoint
- Should be independent of project responsible for system. The group is sometimes called a change control board
- May include representatives from client and contractor staff

Version and release management

- Invent identification scheme for system versions
- Plan when new system version is to be produced
- Ensure that version management procedures and tools are properly applied
- Plan and distribute new system releases

Versions/variants/releases

- *Version* An instance of a system which is functionally distinct in some way from other system instances
- *Variant* An instance of a system which is functionally identical but non-functionally distinct from other instances of a system
- *Release* An instance of a system which is distributed to users outside of the development team

Version identification

- Procedures for version identification should define an unambiguous way of identifying component versions
- Three basic techniques for component identification
    - Version numbering
    - Attribute-based identification
    - Change-oriented identification

Version numbering

- Simple naming scheme uses a linear derivation e.g. V1, V1.1, V1.2, V2.1, V2.2 etc.
- Actual derivation structure is a tree or a network rather than a sequence
- Names are not meaningful.
- Hierarchical naming scheme may be better

Release management

- Releases must incorporate changes forced on the system by errors discovered by users and by hardware changes
- They must also incorporate new system functionality
- Release planning is concerned with when to issue a system version as a release

System releases

- Not just a set of executable programs
- May also include
    - Configuration files defining how the release is configured for a particular installation
    - Data files needed for system operation

- An installation program or shell script to install the system on target hardware
- Electronic and paper documentation
- Packaging and associated publicity

- Systems are now normally released on CD-ROM or as downloadable installation files from the web

Release problems

- Customer may not want a new release of the system
  - They may be happy with their current system as the new version may provide unwanted functionality
- Release management must not assume that all previous releases have been accepted. All files required for a release should be re-created when a new release is installed

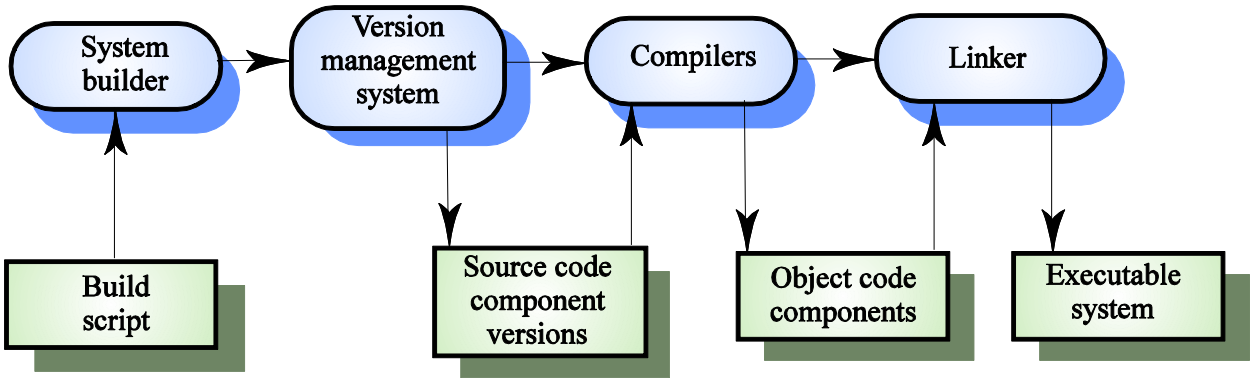
Release decision making

- Preparing and distributing a system release is an expensive process
- Factors such as the technical quality of the system, competition, marketing requirements and customer change requests should all influence the decision of when to issue a new system release

System building

- The process of compiling and linking software components into an executable system
- Different systems are built from different combinations of components
- Invariably supported by automated tools that are driven by 'build scripts

System building



System representation

- Systems are normally represented for building by specifying the file name to be processed by building tools. Dependencies between these are described to the building tools
- Mistakes can be made as users lose track of which objects are stored in which files

A system modelling language addresses this problem by using a logical rather than a physical system representation.

5.11 PROGRAM EVOLUTION DYNAMICS

- Program evolution dynamics is the study of the processes of system change
- After major empirical study, Lehman and Belady proposed that there were a number of 'laws' which applied to all systems as they evolved
- There are sensible observations rather than laws. They are applicable to large systems developed by large organisations. Perhaps less applicable in other cases

Lehman's laws

Prof. Meir M. Lehman, who worked at Imperial College London from 1972 to 2002, and his colleagues have identified a set of behaviours in the evolution of proprietary software. These behaviours (or observations) are known as *Lehman's Laws*, and there are eight of them:

1. Continuing Change
2. Increasing Complexity
3. Large Program Evolution
4. Invariant Work-Rate
5. Conservation of Familiarity
6. Continuing Growth
7. Declining Quality
8. Feedback System

The laws predict that change is inevitable and not a consequence of bad programming and that there are limits to what a software evolution team can achieve in terms of safely implementing changes and new functionality.

Maturity Models specific to software evolution have been developed to help improve processes to ensure continuous rejuvenation of the software evolves iteratively.

The "global process" that is made by the many stakeholders (e.g. developers, users, their managers) has many feedback loops. The evolution speed is a function of the feedback loop structure and other characteristics of the global system. Process simulation techniques, such as system dynamics can be useful in understanding and managing such global process.

Software evolution is not likely to be Darwinian, Lamarckian or Baldwinian, but an important phenomenon on its own. Giving the increasing dependence on software at all levels of society and economy, the successful evolution of software is becoming increasingly critical. This is an important topic of research that hasn't received much attention.

The evolution of software, because of its rapid path in comparison to other man-made entities, was seen by Lehman as the "fruit fly" of the study of the evolution of artificial systems.

Applicability of Lehman's laws

- This has not yet been established
- They are generally applicable to large, tailored systems developed by large organisations
- It is not clear how they should be modified for
    - Shrink-wrapped software products
    - Systems that incorporate a significant number of COTS components
    - Small organisations
    - Medium sized systems

5.12 SOFTWARE MAINTENANCE

- Modifying a program after it has been put into use
- Maintenance does not normally involve major changes to the system's architecture
- Changes are implemented by modifying existing components and adding new components to the system

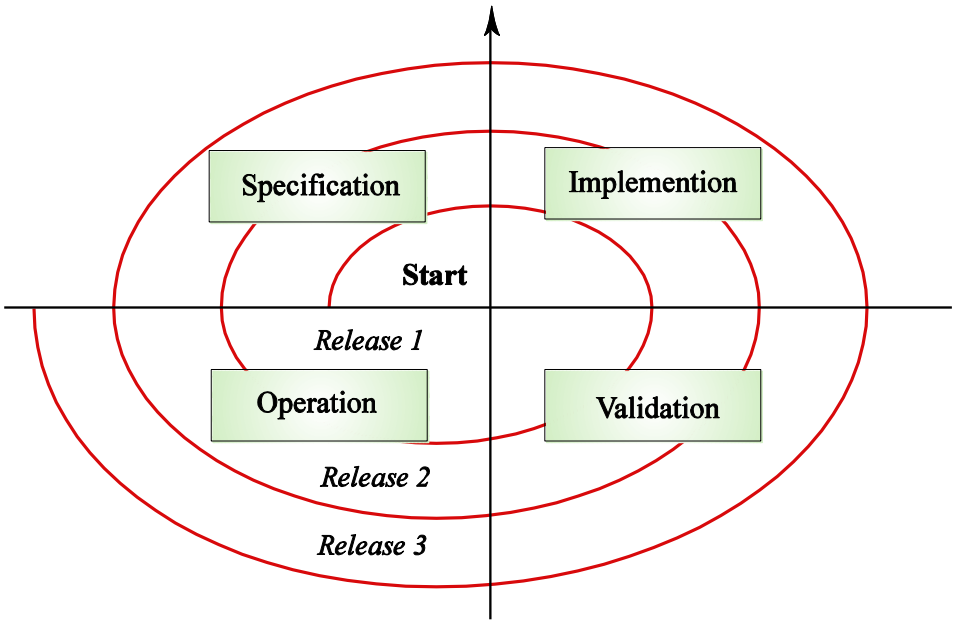
Maintenance is inevitable

- The system requirements are likely to change while the system is being developed because
  the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- Systems MUST be maintained therefore if they are to remain useful in an environment

Types of maintenance

- Maintenance to repair software faults
  - Changing a system to correct deficiencies in the way meets its requirements
- Maintenance to adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation
- Maintenance to add to or modify the system's functionality
  - Modifying the system to satisfy new requirements

Spiral maintenance model



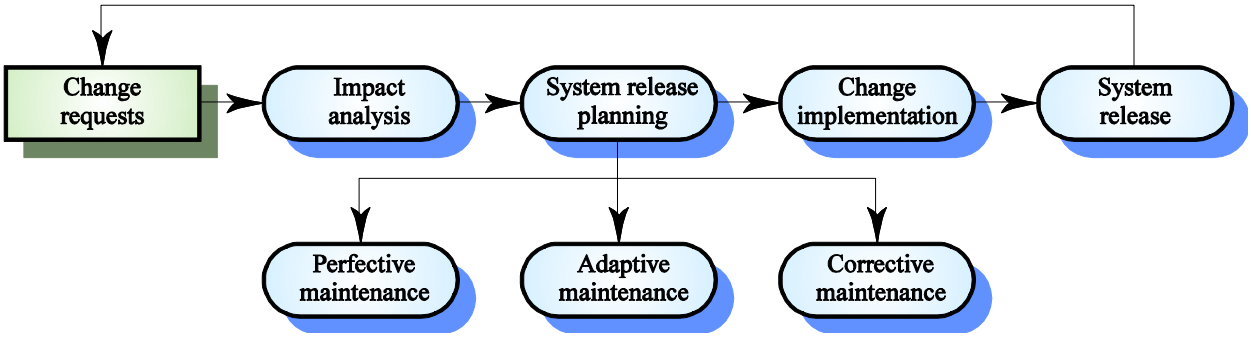
Maintenance costs

- Usually greater than development costs (2* to 100* depending on the application)
- Affected by both technical and non-technical factors
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- Ageing software can have high support costs (e.g. old languages, compilers etc.)

Maintenance cost factors

- Team stability
    - Maintenance costs are reduced if the same staff are involved with them for some time
- Contractual responsibility
    - The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change
- Staff skills
    - Maintenance staff are often inexperienced and have limited domain knowledge
- Program age and structure
    - As programs age, their structure is degraded and they become harder to understand and change

The maintenance process



SOFTWARE PROJECT MANAGEMENT

Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software

Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software

Management activities

- Proposal writing
- Project planning and scheduling
- Project costing
- Project monitoring and reviews
- Personnel selection and evaluation
- Report writing and presentations

5.13 PROJECT PLANNING

- Probably the most time-consuming project management activity
- Continuous activity from initial concept through to system delivery. Plans must be regularly revised as new information becomes available
- Various different types of plan may be developed to support the main software project plan that is concerned with schedule and budget

Project Planning Process

Establish the project constraints
Make initial assessments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled **loop**
Draw up project schedule
Initiate activities according to schedule
Wait (for a while)
Review project progress
Revise estimates of project parameters
Update the project schedule
Re-negotiate project constraints and deliverables
if (problems arise) **then**
Initiate technical review and possible revision
end if

end loop

Project plan structure

- Introduction
- Project organisation
- Risk analysis
- Hardware and software resource requirements
- Work breakdown
- Project schedule
- Monitoring and reporting mechanisms

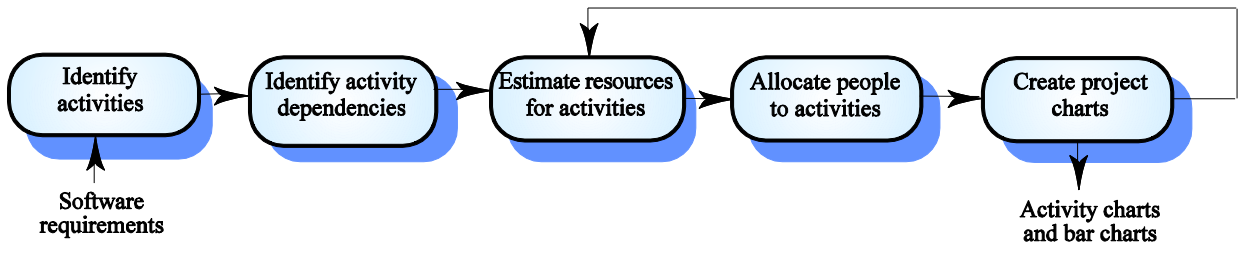
Activity organization

- Activities in a project should be organised to produce tangible outputs for management to judge progress
- *Milestones* are the end-point of a process activity
- *Deliverables* are project results delivered to customers
- The waterfall process allows for the straightforward definition of progress milestones

5.14 PROJECT SCHEDULING

- Split project into tasks and estimate time and resources required to complete each task
- Organize tasks concurrently to make optimal use of workforce
- Minimize task dependencies to avoid delays caused by one task waiting for another to complete
- Dependent on project managers intuition and experience

The project scheduling process



Scheduling problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard
- Productivity is not proportional to the number of people working on a task
- Adding people to a late project makes it later because of communication overheads
- The unexpected always happens. Always allow contingency in planning

5.15 Risk management:

Risks are potential problems that affect successful completion of project which involves uncertainty and potential lose . Risk analysis and management helps the s/w team to overcome the problems caused by the risks.

The work product is called a Risk Mitigation, Monitoring and Management Plan(RMMMP).

(a) risk strategies:

* reactive strategies: also known as fire fighting, project team sets resources aside to deal with the problem and does nothing until the risks became a problem.

* Pro active strategies: risk management begins long before technical works starts , risks are identified and prioritized by importance, the team lead builds a plan to avoid such risks.

(b) S/W risks:

* Project risk:

* Technical risk: threaten product quality and time line

* business risk: threaten the validity of the s/w.

* Known risk: predictable from careful evaluation of current project plan and those extrapolated from past project experience.

* Unknown risk: some problems simply occur without warning.

(c) Risk identification:

* Product – specific risk: the project plan and s/w statement of scope are examined to identify any specific characteristics.

* Generic risk: Potential threads to any s/w products.

(d) Risk impact:

* Risk components: performance, cost, support, schedule

* Risk impact: negligible, marginal , critical, catastrophic.

The risk drivers affecting each risk component are classified according to their impact category and potential consequence of each undetected s/w fault.

(e) Risk projection:

* Establish a scale that reflects the perceived likelihood of each risks.

Delineate the consequence of the risk.

Estimate the impact of the risk on project and product.

(f) risk table construction:

* List all risks in the first column of the table

* Classify each risk and enter the category label in column 2

* Determine a probability for each risk and enter to third column.

* Enter the severity of the risk in column 4.

* Sort the table by probability and impact value.

* First priority concerns must be managing(RMMM in fifth column).

(g) assessing risk impact:

* Factors affecting risk consequences: nature, scope, and timing of the risk.

* If costs are associated with each risk table entry Halstead's risk exposure can be adopted and added to risk table.

RE= probability * cost

(h) risk assessment:

* Defines referent levels for each project risk that can cause project termination.

* Attempt to develop a relationship between each risk triple.

* Predict the set of referent point that define a region of termination, bounded by a curve or areas of uncertainties.

(i) risk refinement:

* process of restating the risks as set of more detailed risk that will be easier to migrate, migrate, and manage.

* CTC format may be a good representation for the detailed risk(condition – transition – consequence)

(J) RISK MITIGATION, MONITORING AND MANAGEMENT:

* Risk mitigation: proactive planning for risk avoidance

* risk monitoring: accessing whether predicted risks occur or not, collect information for further risk analysis.

* Risk management and contingency planning : actions to be taken in the event that mitigation step have failed and the risk become a life problem.

(k) safety risks and hazards:

- Risks also associated with s/w failures that occur in the field after the development project has ended.
- Computers control may listen critical application in modern science.

- Software safety and hazard analysis and quality assurance activity

(l) risk information sheets:

- alternative to RMMM in which each risk is documented separately.
- RIS are maintain using a database system

(m) Examples:

- Pilot study
- Market research – study of business impact risks
- Training

(n) outline of risk management:

- identification-define risks for the project
- projection attempt to indicate quantitative likelihood that a risk will occur.
- Assessment – evaluate the accuracy of projection and prioritize risks.
- Management and monitoring: move to avert those risks that are of concern and monitor all circumstance that may leave to risks.

1. risk identification:

- A systematic attempt to specify thread to the project plan.
- Both generic and product specific risks
- Risk identification : check list

(i) people / staff
(ii) customer / user
(iii) business/business impact
(iv) application, product size, technology
(v) process maturity

2. risk projection:

- establish a scale that reflects the perceived likelihood of the risks.
- Define the consequence of the risks.
- Estimate the impact of the risk on the project and or the product

3. risk assessment:

- recording risks:
- building the risk table

(i) estimate the probability of the occurrence
(ii) estimate the impact on the project
(iii) add RMMM plan
(iv) sort the table by probability and impact

4. risk mitigation, monitoring and management:

- mitigation: how to avoid risk

- monitoring: what factors can be track that will enable us to determine the causes of risks.
- Management: what contingency plans do we have if the risks become a real.

5.16 Taxonomy of CASE tools:

What is CASE?

The computer aided software engineering (CASE) is an automated support for the software engineering process.

The workshop for software engineering has been called an integrated project support environment and the tools that fill the workshop are collectively called CASE.

CASE provides the software engineer with the ability to automate manual activities and to improve engineering insight. CASE tools help to ensure that quality is designed in before the produce is built.

Different levels of CASE technology:

1. **Production process support technology.**
   This includes support for process activities such as specification, design, implementation, testing, and so on. These were the earliest and consequently are the most nature CASE products.
2. **Process management technology.**
   This includes tools to support process modeling and process management. These tools will call on production process support tools to support specific process activities.
3. **Meta CASE technology.**
   Meta CASE tools are generators which are used to create production process and Process management support tools.

5.17

It is necessary to create taxonomy of CASE tools to better understand the breadth of CASE and to better appreciate where such tools can be applied in the software engineering process.

Different classification dimensions of CASE:
CASE tools can be classified

- By function.
- By their roles as instruments for managers or technical people.
- By their use in the various steps of the software engineering process.
- By environment architecture that supports them.
- By their origin or cost.

Various CASE tools based on functions:

1. **Business process engineering tools.**
   The primary objective of these tools is to represent business data objects, their relationships and how these data objects flow between different business areas within a company.

2. **Process modeling and management tools.**
   These tools provide links to other tools that provide support to defined process activities.

3. **Project planning tools.**
   Tools in this category focus on software project effort & cost estimation and project scheduling.

4. **Risk analysis tools.**
   These tools enable a project manager to build a risk table by providing detailed guidance in the identification and analysis of risks.

5. **Project management tools.**
   Tools in this category are often extensions to project planning tools.

6. **Requirements tracing tools.**
   The objective of these tools is to provide a systematic approach to the isolation of requirements.

1. **Documentation tools.**
   These tools provide an important opportunity to improve productivity.

8. **System software tools.**
   CASE is a workstation technology. Therefore, the CASE environment must accommodate high quality network system software.

9. **Quality assurance tools.**
   Most of these tools are actually metrics tools that audit source code to determine compliance with language standards.

10. **Database management tools.**

These tools for CASE are evolving from relational database management system to object oriented database management system.

11. Software Configuration Management tools.

These tools assist in all major SCM tasks namely identification, version control, change control, auditing and status accounting.

12. Analysis and design tools.

These tools enable a software engineer to create models of the system to be built.

13. PRO/SIM tools.

These tools provide the software engineer with the ability to predict the behavior of a real time system prior to the time that it is built.

14. Interface design and development tools.

These tools are actually a tool kit of software components such as menus, buttons, window structures, icons, scrolling mechanisms, and device drivers.

15. Prototyping tools.

A variety of such tools are used. Some of these tools enable the creation of a data design, coupled with both screen and report layouts.

16. Programming tools.

These tools encompass the compilers, editors, and debuggers that are available to support most conventional programming languages.

17. Web development tools.

These tools assist in the generation of text, graphics, font, scripts, applets, and other elements of a web page.

18. Integration and testing tools.

These tools involve data acquisition, static measurement, dynamic measurement, simulation, and test management.

19. Static analysis tools.

These tools assist the software engineer in deriving test cases.

20. Dynamic analysis tools.

These tools interact with an executing program, checking path coverage, and testing assertions about the value of specific variables.

21. Test management tools.

These tools are used to control and coordinate software testing for each of the major testing steps.

22. **Client/server testing tools.**
    These tools exercise the graphical user interface and the network communications requirements for client and server.

23. **Reengineering tools.**
    Some of these tools are Reverse engineering to specification tools, Code capturing and analysis tools, and On-line system reengineering tools.