WWW.KTUNOTES.IN

Module I

Introduction to software engineering- scope of software engineering – historical aspects, economic aspects, maintenance aspects, specification and design aspects, team programming aspects. Software engineering a layered technology – processes, methods and tools. Software process models – prototyping models, incremental models, spiral model, waterfall model.

Q1. Define software Engineering?

Software

- Computer programs and associated documentation
  - requirements, design models and user manuals.
- Software products may be
  - Generic - developed to be sold to a range of different customers
- e.g. PC software such as Excel or Word.
  - custom - developed for a single customer according to their specification.
- New software can be created by developing new programs, configuring generic software systems or reusing existing software.

What is computer Science

- Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
- Computer science theories are still insufficient to act as a complete underpinning for software engineering.

System engineering vs Software Engineering

- System engineering - computer-based systems development including hardware, software and process engineering.
- Software engineering is part of this process - developing the software infrastructure, control, applications and databases in the system.
- System engineers are involved in system specification, architectural design, integration and deployment.

Software Process

- A set of activities for the development or evolution of software.
- Generic activities in all software processes are:
  - Specification - what the system should do and its development constraints
  - Development - production of the software system
  - Validation - checking that the software is what the customer wants
  - Evolution - changing the software in response to changing demands.

1.1 Introduction to Software Engineering

- Software Engineering is an engineering discipline which is concerned with all aspects of software production from the early stages of system requirements through to maintaining the system after is has gone into use.
- Computer Science is concerned with the theories and methods which underlie computers and software systems.
- Software Engineering is concerned with the practical problem of producing software.
- "Engineering discipline" – Engineers make things work. They apply theories, methods and tools that are appropriate but use them selectively and always try to discover solutions to problems even when there are not applicable theories and methods to support them. Engineers have to work to organizational and financial constraints.
- "All aspects of" - Software engineering is not just concerned with the technical process of software development but also with activities such as software project management and the development of tools, methods and theories to support software product

Q2. Explain the various scope of software engineering?

1.2 Scope of software engineering

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

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

These five aspects can be categorized in the fields of Mathematics, Computer Science Economics, Management, and Psychology

1.3 Historical Aspects

- In the belief that software could be engineered on the same footing as traditional engineering disciplines a NATO study group coined the term "Software Engineering" in 1967. This was endorsed by the NATO Software Engineering Conference in 1968.
- Software engineering cannot be considered as engineered since an unacceptably large proportion of software products still are being
  - Delivered late
  - Over budget
    - With residual faults.
  - Solution is that a software engineer has to acquire a broad range of skills, both technical managerial. These skills have to be applied to Programming and every step of software production, from requirements to maintenance.

1.4 Economic Aspects

- Applying economic principles to software engineering requires the client to choose techniques that reduce long-term costs in terms of the economic sense.
- The cost of introducing new technology into an organization may involve training cost , a steep learning curve , unable to do productive work when attending the class etc.

1.5 Maintenance Aspects

- Classical View of Maintenance: Development-then-maintenance model. However, this model is unrealistic due to: – During the development, the client's requirements may change. This leads to the changes in the specification and design. Developers try to reuse parts of existing software products in the software product to be constructed.
- Modern view of Maintenance: It is the process that occurs when "software undergoes modifications to code and associated documentation due to a problem or the need for improvement or adaptation". That is, maintenance occurs whenever a fault is fixed or the requirements change, irrespective of whether this takes place before or after installation of the product.
- Classical Postdelivery Maintenance: All changes to the product once the product has been delivered and installed on the client's computer and passes its acceptance test.
- Modern Maintenance (or just maintenance): Corrective, perfective, or adaptive activities performed at any time. Classical post delivery maintenance is a subset of modern maintenance
- The importance of Post delivery Maintenance: – A software product is a model of the real world, and the real world is perpetually changing. As a consequence, software has to be maintained constantly for it to remain an accurate reflection of the real world. A major aspect of software engineering consists of techniques, tools, and practices that lead to a reduction in post delivery maintenance cost.

1.6. Requirements, Analysis, and Design Aspects

- The earlier we correct a fault, the better. That is, the cost of correcting a fault increases steeply since it is directly related to what has to be done to correct a fault.
- If the mistake is made while eliciting the requirements, the resulting fault will probably also appear in the specifications, the design, and the code. We have to edit the code, recompile and relink the code, and test.
- It is crucial to check that making the change has not created a new problem elsewhere in the product. All the relevant documentation, including manuals, needs to be updated. The corrected product must be delivered and reinstalled.

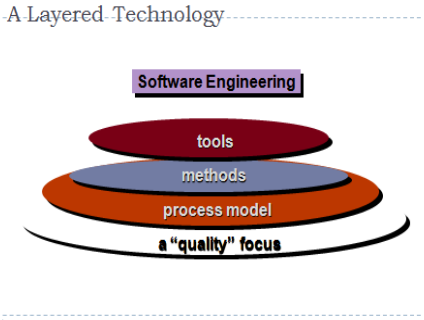
1.7 Team Development Aspects

- Team development leads to interface problems among code components and communication problems among team members.
- Unless the team is properly organized, an inordinate amount of time can be wasted in conferences between team members.
- It also includes human aspects, such as team organization, economic aspects, and legal aspects, such as copyright law.

Q3. Explain software engineering as a layered technology?

1.8 Software Engineering a Layered Technology

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines



a. Quality Focus

Continuous process improvement. Bedrock that supports software engineering.

b. Process

It is the foundation of software engineering. Defines a framework that must be established for

- Effective delivery of SE technology
- Management control of software project
- Context of technical methods applied
- Work products
- Milestones, Quality ensured
- Proper change management

c. Methods

- Provide technical how-to's for building software.
- Encompass a broad array of tasks that include requirements analysis, design modelling , program construction, testing, and support.
- Rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques

d. Tools

- Provide automated or semi-automated support for the process and the methods.
- CASE: computer-aided software engineering is a system for the support of software development.
    - Combines SW, HW, and a SE database (a repository containing important information about analysis, design, program construction, and testing)

Q4. Explain the various software process models?

1.9 Software Process Model

- A software engineer must incorporate a development strategy that encompasses
    - the process methods, and tools layers.
- This strategy is often referred to as a *process model* or a *software engineering paradigm*.
- A process model is chosen based on
    - the nature of project and application,
    - the methods and tools to be used,
    - and the controls and deliverables that are required

Q5. Explain about waterfall model

1.10 classical waterfall model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it can not be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the

classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model.
Classical waterfall model divides the life cycle into the following phases as shown in fig

- Feasibility Study
- Requirements Analysis and Specification
- Design
- Coding and Unit Testing
- Integration and System Testing
- Maintenance

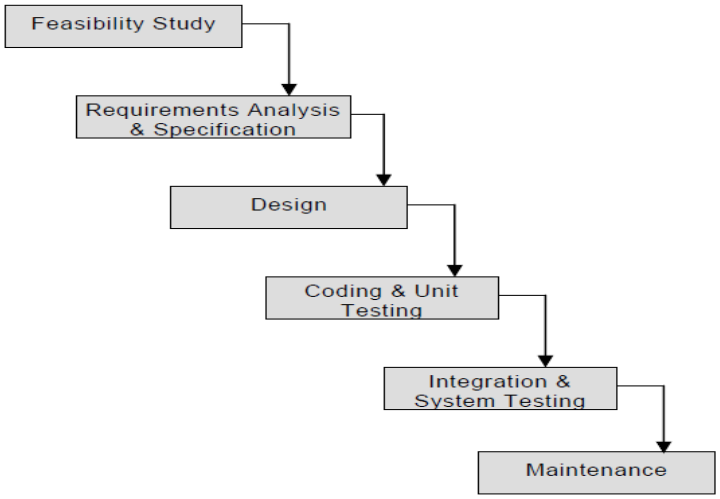


Fig Classical Waterfall Model

Activities in each phase of the life cycle

• **Activities undertaken during feasibility study: -**

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.

- After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.

- Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

The following is an example of a feasibility study undertaken by an organization. It is intended to give you a feel of the activities and issues involved in the feasibility study phase of a typical software project.

handling communication with the mine sites. He arrived at a cost to develop from the analysis. He found that the solution involving maintenance of local databases at the mine sites and periodic updating of a central database was financially and technically feasible. The project manager discussed his solution with the GMC management and found that the solution was acceptable to them as well.

• **Activities undertaken during requirements analysis and specification: -**

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

Requirements gathering and analysis, and

Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document.

The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

• **Activities undertaken during design: -**

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

Traditional design approach

Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

Object-oriented design approach

In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

• **Activities undertaken during coding and unit testing:-**

The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested.

During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

• **Activities undertaken during integration and system testing: -**

Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

- α – testing: It is the system testing performed by the development team.
- β – testing: It is the system testing performed by a friendly set of customers.
- acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

• **Activities undertaken during maintenance: -**

Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

Shortcomings of the classical waterfall model

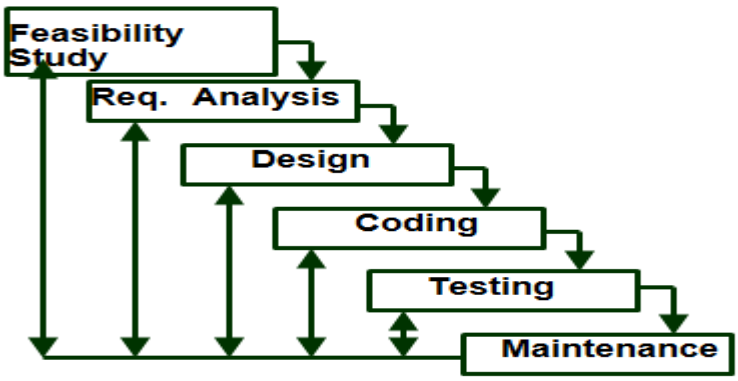
- The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc.
- . These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is

detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases.

- Phase-entry and phase-exit criteria of each phase
  - At the starting of the feasibility study, project managers or team leaders try to understand what is the actual problem by visiting the client side. At the end of that phase they pick the best solution and determine whether the solution is feasible financially and technically.
  - At the starting of requirements analysis and specification phase the required data is collected. After that requirement specification is carried out. Finally, SRS document is produced.
  - At the starting of design phase, context diagram and different levels of DFDs are produced according to the SRS document. At the end of this phase module structure (structure chart) is produced.
  - During the coding phase each module (independently compilation unit) of the design is coded. Then each module is tested independently as a stand-alone unit and debugged separately. After this each module is documented individually. The end product of the implementation phase is a set of program modules that have been tested individually but not tested together.
  - After the implementation phase, different modules which have been tested individually are integrated in a planned manner. After all the modules have been successfully integrated and tested, system testing is carried out.
  - Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use.

Iterative Waterfall Model

- Classical waterfall model is idealistic-assumes that no defect is introduced during any development activity.In practice defects do get introduced in almost every phase of the life cycle. Defects usually get detected much later in the life cycle:
  - For example, a design defect might go unnoticed till the coding or testing phase.
- Once a defect is detected:
  - we need to go back to the phase where it was introduced
  - redo some of the work done during that and all subsequent phases.
- **Therefore we need feedback paths in the classical waterfall model.**



- **Errors should be detected**

- in the same phase in which they are introduced.

- **For example:**
  - **if a design problem is detected in the design phase itself,**
    - the problem can be taken care of much more easily
    - than say if it is identified at the end of the integration and system testing phase.

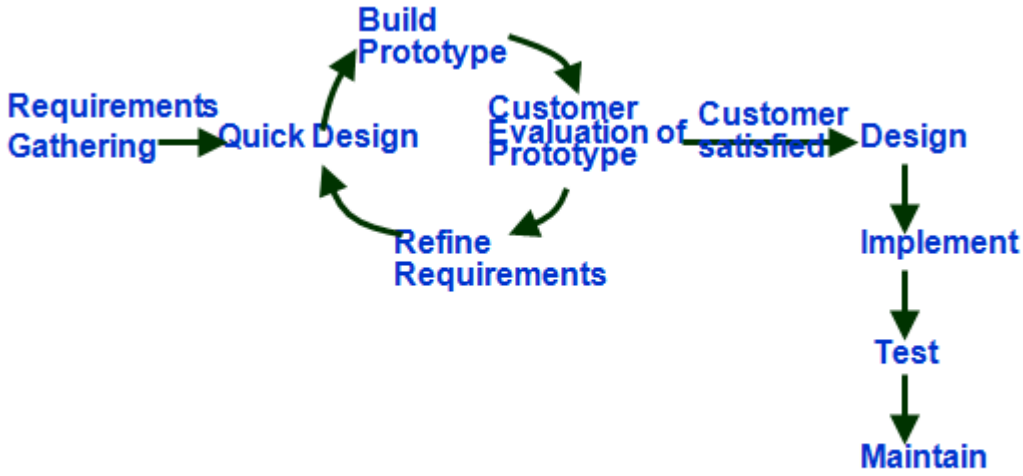
Q6. Summarize prototyping model with a neat diagram?

Prototyping Model

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

- **Start with approximate requirements.**
- **Carry out a quick design.**
- **The developed prototype is submitted to the customer for his evaluation:**
  - Based on the user feedback, requirements are refined.
  - This cycle continues until the user approves the prototype.

The actual system is developed using the classical waterfall approach



Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

- This is something similar to what the architectural designers of a building do; they show a prototype of the building to their customer. The customer can evaluate whether he likes it or

not and the changes that he would need in the actual product. A similar thing happens in the case of a software product and its prototyping model.

- Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

- A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development.

- Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

Examples for prototype model

A prototype of the actual product is preferred in situations such as:

• user requirements are not complete
• technical issues are not clear

Let's see an example for each of the above category.

Example 1: User requirements are not complete

In any application software like billing in a retail shop, accounting in a firm, etc the users of the software are not clear about the different functionalities required. Once they are provided with the prototy pe implementation, they can try to use it and find out the missing functionalities.

Example 2: Technical issues are not clear

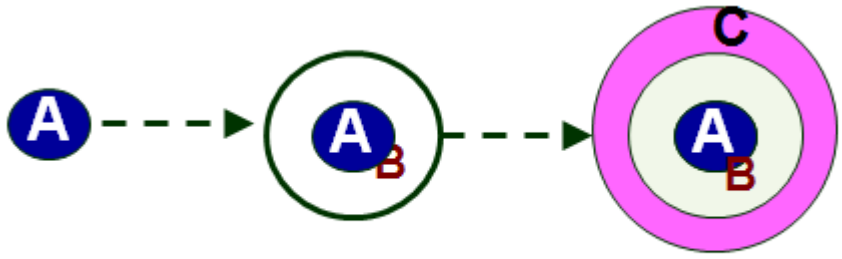
Suppose a project involves writing a compiler and the development team has never written a compiler.

In such a case, the team can consider a simple language, try to build a compiler in order to check the issues that arise in the process and resolve them. After successfully building a small compiler (prototype), they would extend it to one that supports a complete language.

Q7. Describe evolutionary or incremental model?

Evolutionary Model(Incremental Model)

- Evolutionary model (aka successive versions or incremental model):
    - The system is broken down into several modules which can be incrementally implemented and delivered.
- First develop the core modules of the system.
- The initial product skeleton is refined into increasing levels of capability:by adding new functionalities in successive versions
- Successive version of the product:
- functioning systems capable of performing some useful work.
- A new release may include new functionality:
- also existing functionality in the current release might have been enhanced



Advantages of Evolutionary Model

- Users get a chance to experiment with a partially developed system:
  - much before the full working version is released,
- Helps finding exact user requirements:
  - much before fully working system is developed.
- Core modules get tested thoroughly:
  - reduces chances of errors in final product.

Disadvantages of Evolutionary Model

- Often, difficult to subdivide problems into functional units:which can be incrementally implemented and delivered.
- evolutionary model is useful for very large problems, where it is easier to find modules for incremental implementation.

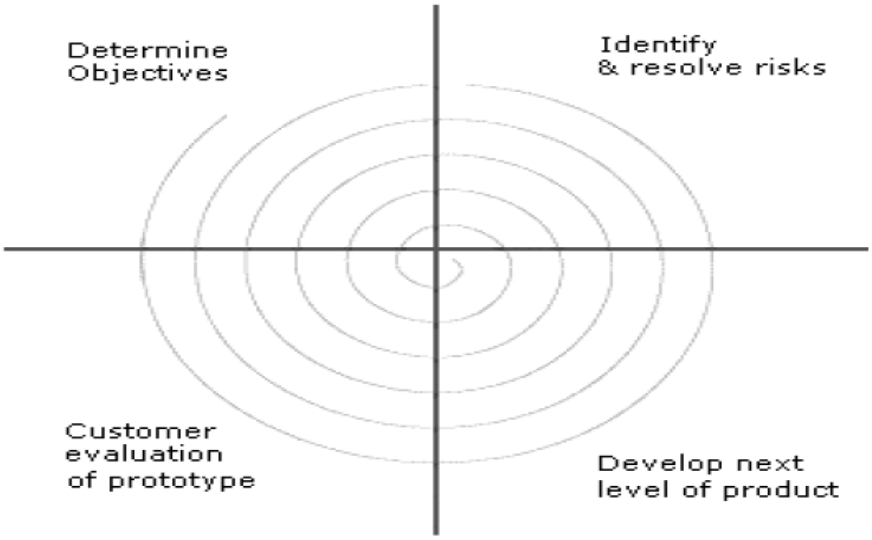
Q8. Explain Spiral model?

Spiral model

The Spiral model of software development is shown in fig. 2.2. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study. The next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. below The following activities are carried out during each phase of a spiral model.

- **First quadrant (Objective Setting)**
  - During the first quadrant, it is needed to identify the objectives of the phase.
  - Examine the risks associated with these objectives.
- **Second Quadrant (Risk Assessment and Reduction)**
  - A detailed analysis is carried out for each identified project risk.

• Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.



Spiral Model

- Third Quadrant (Development and Validation)

• Develop and validate the next level of the product after resolving the identified risks.

- Fourth Quadrant (Review and Planning)

• Review the results achieved so far with the customer and plan the next iteration around the spiral.

• Progressively more complete version of the software gets built with each iteration around the spiral.

Circumstances to use spiral model

- The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

Q9. Com[pare various process models?

Comparison of different life-cycle models

- The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model can not be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.
- This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.
- The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

- The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.
- The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.
- The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment.
- On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront
- **Iterative waterfall model**
  - most widely used model.
  - model is simple to understand and use
  - But, suitable only for well-understood problems**.**
- **Prototype model is suitable for projects**
  - either user requirements
  - Or technical aspects not well understood:
  - popular for development of the user-interface part of the projects.
- **Evolutionary model is suitable for large problems:**
  - can be decomposed into a set of modules that can be incrementally implemented,
  - incremental delivery of the system is acceptable to the customer.
  - widely used for object-oriented development projects.
- **The spiral model:**
  - It is a realistic approach to the development of a large scale system
  - suitable for development of technically challenging software products that are subject to several kinds of risks.
  - Complex model

Module II

Process Framework Models: Capability maturity model (CMM), ISO 9000. Phases in Software development – requirement analysis requirements elicitation for software, analysis principles, software prototyping, specification.

Q1. Explain CMM with suitable sketch

2.1 Capability Maturity Model

- Created by the Software Engineering Institute, a research center founded by Congress in 1984
- Describes an evolutionary improvement path for software organizations from an ad hoc, immature process to a mature, disciplined one.
- Provides guidance on how to gain control of processes for developing and maintaining software and how to evolve toward a culture of software engineering and management excellence.

Process Maturity Concepts

a.**Software Process**- it is a set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products (e.g., project plans, design documents, code, test cases, user manuals).

b. Software Process Capability – it describes the range of expected results that can be achieved by following a software process. It is the means of predicting the most likely outcomes to be expected from the next software project the organization undertakes.

c. Software Process Performance- actual results achieved by following a software process

d.Software Process Maturity - Extents to which a specific process is explicitly defined, managed, measured, controlled and effective. It implies potential growth in capability and it indicates richness of process and consistency with which it is applied in projects throughout the organization

CMM Levels

Maturity level indicates level of process capability:There are five maturity levels in CMM. They are.

- Initial
- Repeatable
- Defined
- Managed
- Optimizing

a. Level 1- Initial- The software process is characterized as ad hoc, . Few processes are defined, and success depends on individual effort.

- At this level, frequently have difficulty making commitments that the staff can meet with an orderly process

- Products developed are often over budget and schedule
- Wide variations in cost, schedule, functionality and quality targets
- Capability is a characteristic of the individuals, not of the organization

b. Level2- Repeatable

- Basic process management processes are established to track cost, schedule, and functionality.
- The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
- Realistic project commitments based on results observed on previous projects
- Software project standards are defined and faithfully followed
- Processes may differ between projects
- Process is disciplined
- earlier successes can be repeated

c. Level 3- Defined

- The software process for both management and engineering activities . It is documented, standardized, and integrated into a standard software process for the organization.
- All projects use an approved, tailored version of the organization's standard software process for developing an maintaining software.

d.Level 4- Managed

- Detailed measures of the software process and product quality are collected.
- Both the software process and products are quantitatively understood and controlled.
- Narrowing the variation in process performance - known limits are exceeded- corrective action can be taken
- Quantifiable and predictable
- predict trends in process and product quality

e.Level 5- Optimizing

- Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.
- Goal is to prevent the occurrence of defects
- Data on process effectiveness used for cost benefit analysis of new technologies and proposed process changes.

Key Processing Areas- Except for level 1, each level is decomposed into key process areas (KPA)

Each KPA identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important for enhancing software capability.

a.Level 2 KPA

- Requirements Management
  - Establish common understanding of customer requirements between the customer and the software project
  - Requirements is basis for planning and managing the software project
  - Not working backwards from a given release date
- Software Project Planning
  - Establish reasonable plans for performing the software engineering activities and for managing the software project
- Software Project Tracking and Oversight
  - Establish adequate visibility into actual progress
  - Take effective actions when project's performance deviates significantly from planned
- Software Subcontract Management

- Manage projects outsourced to subcontractors
- Software Quality Assurance
  - Provide management with appropriate visibility into process being used by the software projects and work products
- Software Configuration Management
  a. Establish and maintain the integrity of work products
  b. Product baseline
  c. Baseline authority

b. Level 3 KPA

- Organization Process Focus
  a. Establish organizational responsibility for software process activities that improve the organization's overall software process capability.
- Organization Process Definition
  a. Develop and maintain a usable set of software process assets which are stable foundation that can be institutionalized and a basis for defining meaningful data for quantitative process management
- Training Program
  - Develop skills and knowledge so that individual can perform their roles effectively and efficiently
  - Organizational responsibility
  - Needs identified by project
- Integrated Software Management
  - Integrated engineering and management activities
  - Engineering and management processes are tailored from the organizational standard processes
  - Tailoring based on business environment and project needs
- Software Product Engineering
  - technical activities of the project are well defined (SDLC)
  - correct, consistent work products
- Intergroup Coordination
  - Software engineering groups participate actively with other groups
- Peer Reviews
  - early defect detection and removal
  - better understanding of the products
  - implemented with inspections, walkthroughs, etc

c.Level 4 KPA

- Quantitative Process Management
  - control process performance quantitatively
  - actual results from following a software process
  - focus on identifying and correcting special causes of variation with respect to a baseline process
- Software Quality Management
  quantitative understanding of software quality- products and process

d.Level 5 KPA

- Process Change Management
  - continuous process improvement to improve quality, increase productivity, decrease cycle time
- Technology Change Management
  - identify and transfer beneficial new technologies
    - tools
    - methods

        - processes
    - Defect Prevention
        - causal analysis of defects to prevent recurrence

Benefits

- Helps forge a shared vision of what software process improvement means for the organization
- Defines set of priorities for addressing software problems
- Supports measurement of process by providing framework for performing reliable and consistent appraisals
- Provides framework for consistency of processes and product

Maturity Level	**Description**
1 – Initial	The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics. **Key Process areas-None**
2- Repeatable	Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications. **KPA –Requirements management,Software project planning,Software pjt tracking and oversight,Software Quality assurance,software configuration management**
3 – Defined	The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software. **KPA-Organization process focus,organization process definition,training programme,integrated software management,peer reviews,inter-group oordination etc**
4 – Managed	Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled. KPA-Quantitative process management and software quality management.
5 – Optimized	Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies. KPA-Defect prevention,technology change management,process change management

Q2. Explain ISO 9000

ISO 9000

- ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardization.
- ISO published its 9000 series of standards in 1987.
- The ISO 9000 standard specifies the guidelines for maintaining a quality system.
- ISO 9000 specifies a set of guidelines for repeatable and high quality product development.
- ISO 9000 standard is a set of guidelines for the production process

ISO 9001

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

ISO 9002

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

ISO 9003

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

Need for obtaining ISO 9000 Certification

- Confidence of customers in an organization increases
- ISO 9000 requires a well-documented software production process to be in place. A well-documented software production process contributes to repeatable and higher quality of the developed software.
- Makes the development process focused, efficient, and cost-effective.
- Points out the weak points of an organization and recommends remedial action.
- Sets the basic framework for the development of an optimal process and Total Quality Management

Summary of ISO 9001

- Main requirements related to software industry

Management Responsibility (4.1)

- The management must have an effective quality policy.
- The responsibility and authority of all those whose work affects quality must be defined and documented.
- A management representative, independent of the development process, must be responsible for the quality system.
- The effectiveness of the quality system must be periodically reviewed by audits.

Quality System (4.2)

- A quality system must be maintained and documented.

Contract Reviews (4.3)

- Before entering into a contract, an organization must review the contract to ensure that it is understood, and that the organization has the necessary capability for carrying out its obligations.

Design Control(4.4)

- The design process must be properly controlled, this includes controlling coding also.-a good configuration control system must be in place.
- Design inputs must be verified as adequate.

- Design must be verified.
- Design output must be of required quality.
- Design changes must be controlled.

Document Control(4.5)

- There must be proper procedures for document approval, issue and removal.
- Document changes must be controlled. Thus, use of some configuration management tools is necessary.

Purchasing (4.6)

- Purchasing material, including bought-in software must be checked for conforming to requirements.

Purchaser Supplied Product (4.7)

- Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

Product Identification (4.8)

- The product must be identifiable at all stages of the process. In software terms this means configuration management.

Process Control (4.9)

- The development must be properly managed.
- Quality requirement must be identified in a quality plan.

Inspection and Testing (4.10)

- In software terms this requires effective testing i.e., unit testing, integration testing and system testing. Test records must be maintained.

Inspection, Measuring and Test Equipment (4.11)

- If integration, measuring, and test equipments are used, they must be properly maintained and calibrated.

Inspection and Test Status (4.12)

- The status of an item must be identified. In software terms this implies configuration management and release control.

Control of Nonconforming Product (4.13)

- In software terms, this means keeping untested or faulty software out of the released product, or other places whether it might cause damage.

Corrective Action(4.14)

- This requirement is both about correcting errors when found, and also investigating why the errors occurred and improving the process to prevent occurrences. If an error occurs despite the quality system, the system needs improvement.

Handling, (4.15)

- This clause deals with the storage, packing, and delivery of the software product.

Quality records (4.16)

- Recording the steps taken to control the quality of the process is essential in order to be able to confirm that they have actually taken place.

Quality Audits (4.17)

- Audits of the quality system must be carried out to ensure that it is effective.

Training (4.18)

- Training needs must be identified and met.

Salient Features of ISO 9001 Certification

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

- Important documents should be independently checked and reviewed for effectiveness and correctness.
- The product should be tested against specification.
- Several organizational aspects should be addressed e.g., management reporting of the quality team.

Short Comings of ISO 9000 Certification

- ISO 9000 requires a software production process to be adhered to but does not guarantee the process to be of high quality.
- It also does not give any guideline for defining an appropriate process.
- ISO 9000 certification process has no international accreditation agency exists.
- Therefore it is likely that variations in the norms of awarding certificates can exist among the different accreditation agencies and also among the registrars.
- Organizations getting ISO 9000 certification often tend to downplay domain expertise.
- These organizations start to believe that since a good process is in place, any engineer is as effective as any other engineer in doing any particular activity relating to software development.
- ISO 9000 does not automatically lead to continuous process improvement,

3. Explain different types of Software Requirements?

- Requirements Engineering:- It is the process of establishing the services
  - that the customer requires from a system
  - and the constraints under which it operates and is developed
- The requirements themselves are the descriptions of the
  - system services
  - and constraints that are generated during the requirements engineering process

The types of requirements

- User requirements
  - Statements in natural language plus diagrams of the services the system provides and
  - its operational constraints. Written for customers
- System requirements
  - A structured document setting out detailed descriptions of the system services.
  - Written as a contract between client and contractor
- Software specification
  - A detailed software description which can serve as a basis for a design or implementation. Written for developers
- Functional requirements- Statements or 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
  - functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Domain requirements
- Define system properties and constraints
- e.g. reliability, response time and storage requirements.
- Constraints are I/O device capability, system representations, etc.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless

4. Compare functional and functional requirements

Non functional requirements-

- Define system properties and constraints
- e.g. reliability, response time and storage requirements.

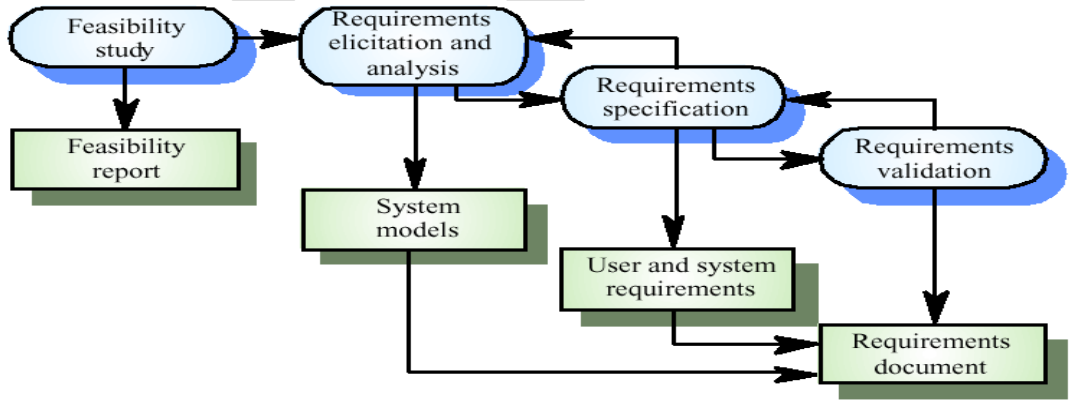
- Constraints are I/O device capability, system representations, etc.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless
    - **Product requirements-** Requirements which specify that the delivered product must behave in a particular 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.

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

5. Explain the activities in software requirement engineering process

- Requirement Engineering process- Processes used to discover, analyse and validate system requirements
- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements
- RE processes are
    - Requirements elicitation
    - Requirements analysis
    - Requirements validation
    - Requirements management



1. Feasibility study An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

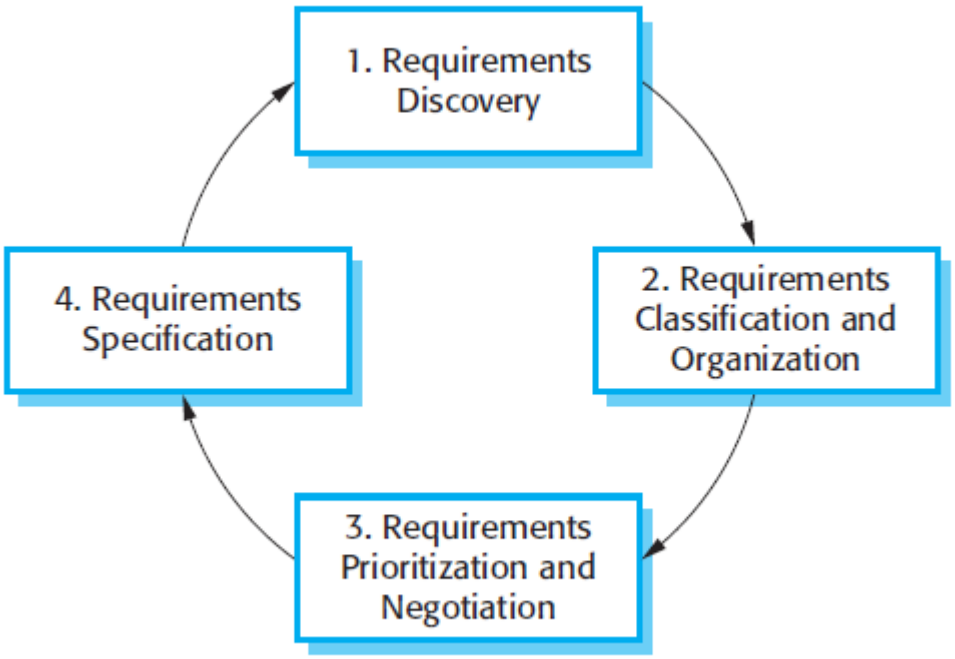
2. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.

3. *Requirements specification* Requirements specification is the activity of translating

the information gathered during the analysis activity into a document tha tdefines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.

4. *Requirements validation* This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

6. Explain the activities involved in requirement elicitation and analysis



The process activities are:

1. *Requirements discovery* This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity. There are several complementary techniques that can be used for requirements discovery, which I discuss later in this section.
2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.
3. *Requirements prioritization and negotiation* Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.
4. *Requirements specification* The requirements are documented and input into the

next round of the spiral.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:
1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
3. Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

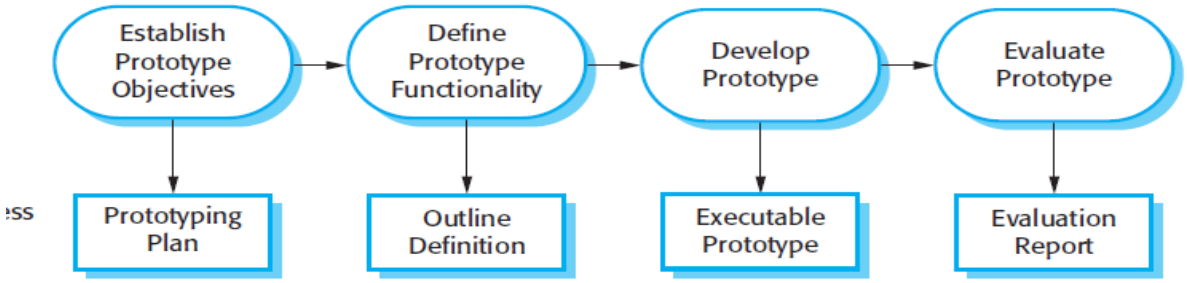
7. Explain Software Prototyping

- A prototype is an initial version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions.
- Rapid, iterative development of the prototype is essential so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.
- A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore particular software solutions and to support user interface design.

- System prototypes allow users to see how well the system supports their work. They may get new ideas for requirements, and find areas of strength and weakness in the software.
- Furthermore, as the prototype is developed, it may reveal errors and omissions in the requirements that have been proposed.
- Prototyping is also an essential part of the user interface design process. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements. Therefore, rapid prototyping with end-user involvement is the only sensible way to develop graphical user interfaces for software systems.
- A function described in a specification may seem useful and well defined. However, when that function is combined with other functions, users often find that their initial view was incorrect or incomplete. The system specification may then be modified to reflect their changed understanding of the requirements. A system prototype may be

used while the system is being designed to carry out design experiments to check the feasibility of a proposed design.



Activities in software prototyping

- A process model for prototype development is shown in Figure. The objectives of prototyping should be made explicit from the start of the process. These may be to develop a system to prototype the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the feasibility of the application to managers.
- The same prototype cannot meet all objectives. If the objectives are left unstated, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.
- The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.
- The final stage of the process is prototype evaluation. Provision must be made during this stage for user training and the prototype objectives should be used to derive a plan for evaluation. Users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirements errors and omissions

✓ A general problem with prototyping is that the prototype may not necessarily be used in the same way as the final system. The tester of the prototype may not be typical of system users.

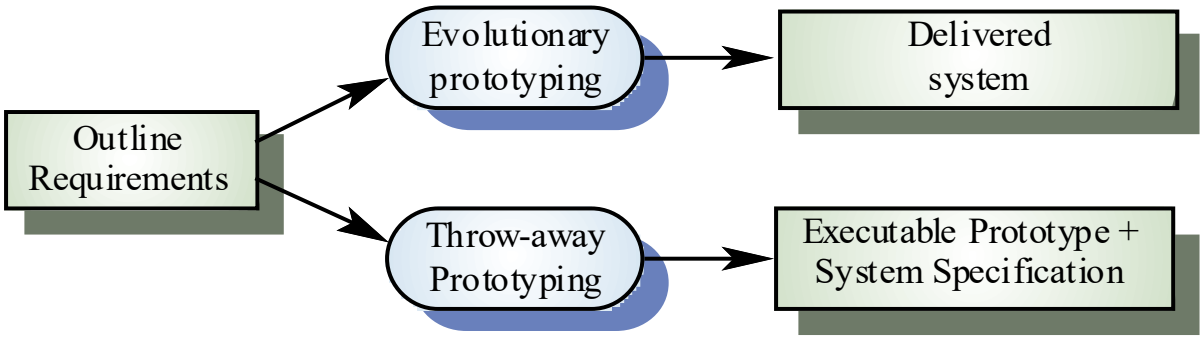
✓ The training time during prototype evaluation may be insufficient. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with better response in the final system, they may use it in a different way.

✓ Developers are sometimes pressured by managers to deliver throwaway prototypes, particularly when there are delays in delivering the final version of the software. However, this is usually unwise:

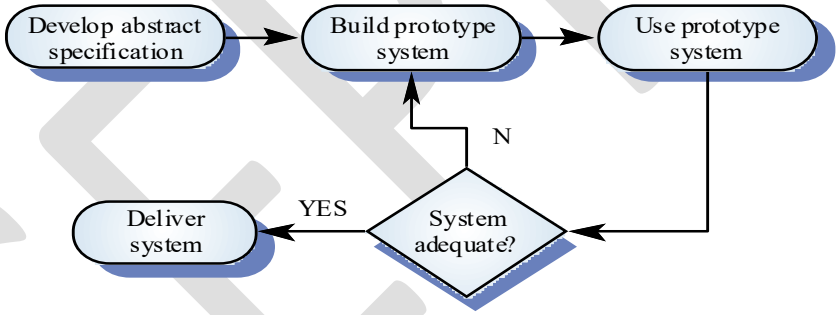
1. It may be impossible to tune the prototype to meet non-functional requirements, such as performance, security, robustness, and reliability requirements, which were ignored during prototype development.
2. Rapid change during development inevitably means that the prototype is undocumented. The only design specification is the prototype code. This is not good enough for long-term maintenance.
3. The changes made during prototype development will probably have degraded the system structure. The system will be difficult and expensive to maintain.
4. Organizational quality standards are normally relaxed for prototype development.

Prototypes do not have to be executable to be useful

Approaches to Prototyping



1. **Evolutionary prototyping**
   - An initial prototype is produced and refined through a number of stages to the final system
   - to deliver a working system to end-users. The development starts with those requirements which are best understood.
   - 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



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

Problems with evolutionary Prototyping

Management problems
- Existing management processes assume a waterfall model of development
- Specialist skills are required which may not be available in all development teams

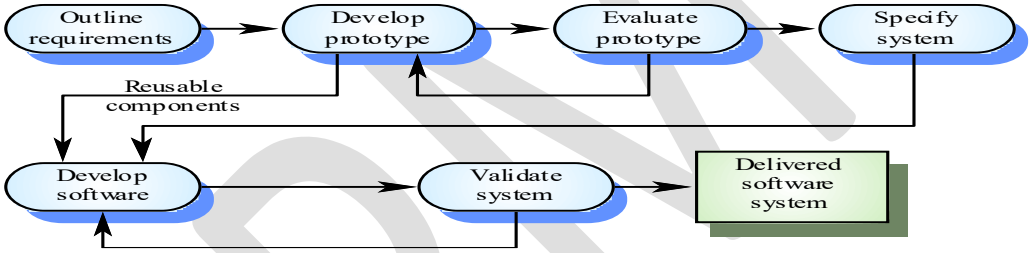
Maintenance problems
- Continual change tends to corrupt system structure so long-term maintenance is expensive.

Contractual problems
- Some parts of the requirements (e.g. safety-critical functions) may be impossible to prototype and so don't appear in the specification
- An implementation has no legal standing as a contract

- Non-functional requirements cannot be adequately tested in a system prototype

2. Throw-away prototyping

- 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.
- to validate or derive the system requirements. The prototyping process starts with those requirements which are poorly understood.
- 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



- Developers may be pressurised to deliver a throw-away prototype as a final system
- This is not recommended
  - It may be impossible to tune the prototype to meet non-functional requirements
  - The prototype is inevitably undocumented
  - The system structure will be degraded through changes made during development
  - Normal organisational quality standards may not have been applied

8. Explain Software specification

Requirements specification is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are more detailed description of the functionality to be provided.

Types of software system requirements:

- *Functional requirements,* describe the requested functionality/behaviour of the system: services (functions), reactions to inputs, exceptions, modes of operations
  - Depend on the system, the software, and the users
  - Can be expressed at different levels of detail (user/system requirements)
  - For a system, it is desirable to have a complete and consistent set of functional requirements
  - *Completeness*: all required system facilities are defined
  - *Consistency*: there are no contradictions in requirements
- *Non-functional requirements,* represent constraints on the system and its functionality: performance constraints, compliance with standards, constraints on the development process
  - Many apply to the system as a whole

- More critical than individual functional requirements
- More difficult to verify

Kinds of non-functional requirements:

- Product requirements
- Organizational requirements
- External requirements

- *Domain requirements,* can be either functional or non-functional and reflect the particularities of the application domain.
  - Domain requirements indicate specific computations, additional functionality, or constraints on other requirements

Software Requirement Specification

- This is the way of representing requirements in a consistent format
- It Serves as a contract between customer & developer.

It should

- Correctly define all requirements
- not describe any design details
- not impose any additional constraints

Characteristics of Good SRS

- Correct
  - An SRS is correct if and only if every requirement stated therein is one that the software shall meet.
- Unambiguous
  - An SRS is unambiguous if and only if, every requirement stated therein has only one interpretation.
- Complete
  - An SRS is complete if and only if, it includes the following elements
  - (i) All significant requirements, whether related to functionality, performance, design constraints, attributes or external interfaces.
  - (ii)Responses to both valid & invalid inputs.
  - (iii) Full Label and references to all figures, tables and diagrams in the SRS and definition of all terms and units of measure.
- Consistent
  - An SRS is consistent if and only if, no subset of individual requirements described in it conflict.
- Stability
  - An identifier is attached to every requirement to indicate either the importance or stability of that particular requirement
- Verifiable
  - An SRS is verifiable, if and only if, every requirement stated therein is verifiable.
- Modifiable
  - An SRS is modifiable, if and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining structure and style.
- Traceable
  - An SRS is traceable, if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

IEEE has published guidelines and standards to organize an SRS.

1. Introduction
   1.1 Purpose

1.2 Scope
1.3 Definition, Acronyms and abbreviations
1.4 References
1.5 Overview

2. The Overall Description
    2.1 Product Perspective
        2.1.1 System Interfaces
        2.1.2 Interfaces
        2.1.3 Hardware Interfaces
        2.1.4 Software Interfaces
        2.1.5 Communication Interfaces
        2.1.6 Memory Constraints
        2.1.7 Operations
        2.1.8 Site Adaptation Requirements
    2.2Product Functions
    2.3 User Characteristics
    2.4 Constraints
    2.5 Assumptions for dependencies
    2.6 Apportioning of requirements
3. Specific Requirements
    3.1 External Interfaces
    3.2 Functions
    3.3 Performance requirements
    3.4 Logical database requirements
    3.5 Design Constraints
    3.6 Software System attributes
    3.7 Organization of specific requirements
    3.8 Additional Comments.
4. Supporting information
    4.1 Table of contents and index
    4.2 Appendixes

Module III

Planning phase – project planning objective, software scope, empirical estimation models- COCOMO, single variable model, staffing and personal planning. Design phase – design process, principles, concepts, effective modular design, top down, bottom up strategies, stepwise refinement.

I. Planning Phase(Ref- Software Engg Ian Sommerville 9th edition –P No 618- 626

- 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.

Project Planning

- Project planning is one of the most important jobs of a software project manager.
- As a manager, you have to break down the work into parts and assign these to project team members, anticipate problems that might arise, and prepare tentative solutions to those problems.
- The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.

Plan Driven Project

- Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail.
- A project plan is created that records the work to be done, who will do it, the development schedule, and the work products
- . Managers use the plan to support project decision making and as a way of measuring progress.
- Plan-driven development is based on engineering project management techniques and can be thought of as the 'traditional' way of managing large software development projects.
- This contrasts with agile development, where many decisions affecting the development are delayed and made later, as required, duringthe development process.

Project Plan

- A project plan sets out the resources available to the project, the work breakdown, and a schedule for carrying out the work. The plan should identify risks to the project and the software under development, and the approach that is taken to risk management.
- Although the specific details of project plans vary depending on the type of project and organization, plans normally include the following sections:

1. ***Introduction***- This briefly describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the management of the project.

2. ***Project organization*** This describes the way in which the development team is organized, the people involved, and their roles in the team.

3. ***Risk analysis*** This describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies that are proposed.

4. ***Hardware and software resource requirements*** This specifies the hardware and support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included.

5. ***Work breakdown*** This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity. Milestones are key stages in the project where progress can be assessed; deliverables are work products that are delivered to the customer.

6. ***Project schedule*** This shows the dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities

7. ***Monitoring and reporting mechanisms*** This defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

The Planning Process

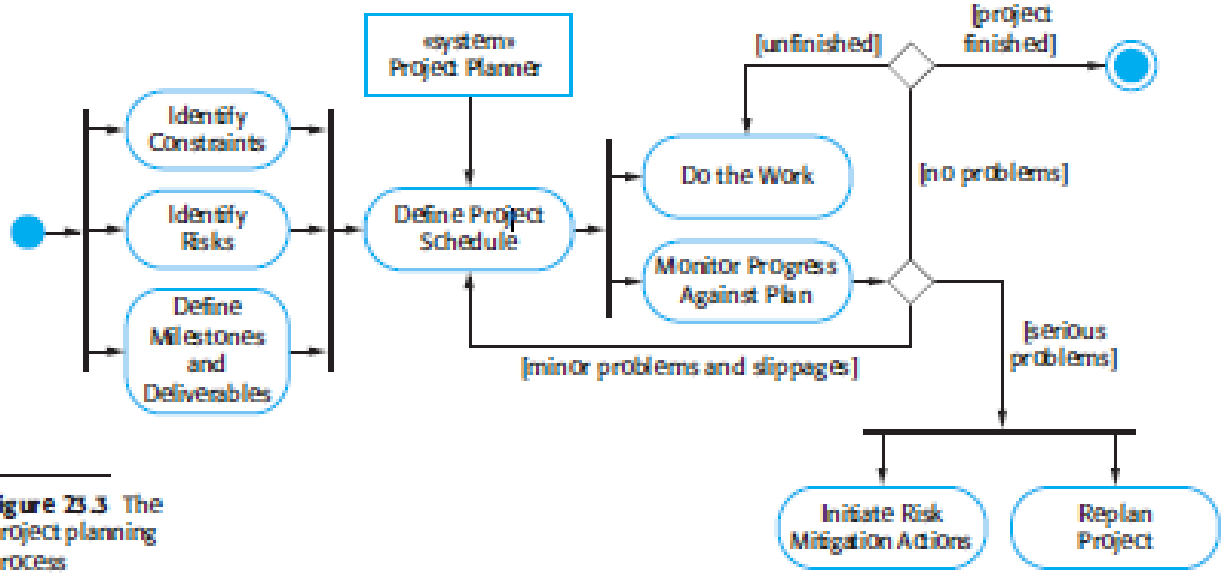


Figure 23.3 The project planning process

- At the beginning of a planning process, assess the constraints affecting

the project. These constraints are the required delivery date, staff available, overall budget, available tools, and so on. In conjunction with this, we should also identify the project milestones and deliverables.

- Milestones are points in the schedule against which we can assess progress, for example, the handover of the system for testing.
- Deliverables are work products that are delivered to the customer (e.g., a requirements

document for the system).

- The process then enters a loop. Draw up an estimated schedule for the project

and the activities defined in the schedule are initiated or given permission to continue.

- After some time (usually about two to three weeks), we should review progress and note discrepancies from the planned schedule. Because initial estimates of project parameters are inevitably approximate, minor slippages are normal and we will have to make modifications to the original plan.
- It is important to be realistic when we are creating a project plan. If there are serious problems with the development work that are likely to lead to significant delays, you need to initiate risk mitigation actions to reduce the risks of
- project failure. In conjunction with these actions, you also have to replan the project.
- This may involve renegotiating the project constraints and deliverables with the customer.
- A new schedule of when work should be completed also has to be established and agreed with the customer. If this renegotiation is unsuccessful or the risk mitigation actions are ineffective, then we should arrange for a formal project technical review.
- The objectives of this review are to find an alternative approach that will allow the project to continue, and to check whether the project and the goals of the customer and software developer are still aligned.
- The outcome of a review may be a decision to cancel a project. This may be a result of technical or managerial failings but, more often, is a consequence of external changes that affect the project. The development time for a large software project is often several years.

- During that time, the business objectives and priorities inevitably change. These changes may mean that the software is no longer required or that the original project requirements are inappropriate.
- Management may then decide to stop software development or to make major changes to the project to reflect the changes in the organizational objectives.

II. Software Scope(Ref: Software Engineering a practitioners approach by Pressman S P No- 677)

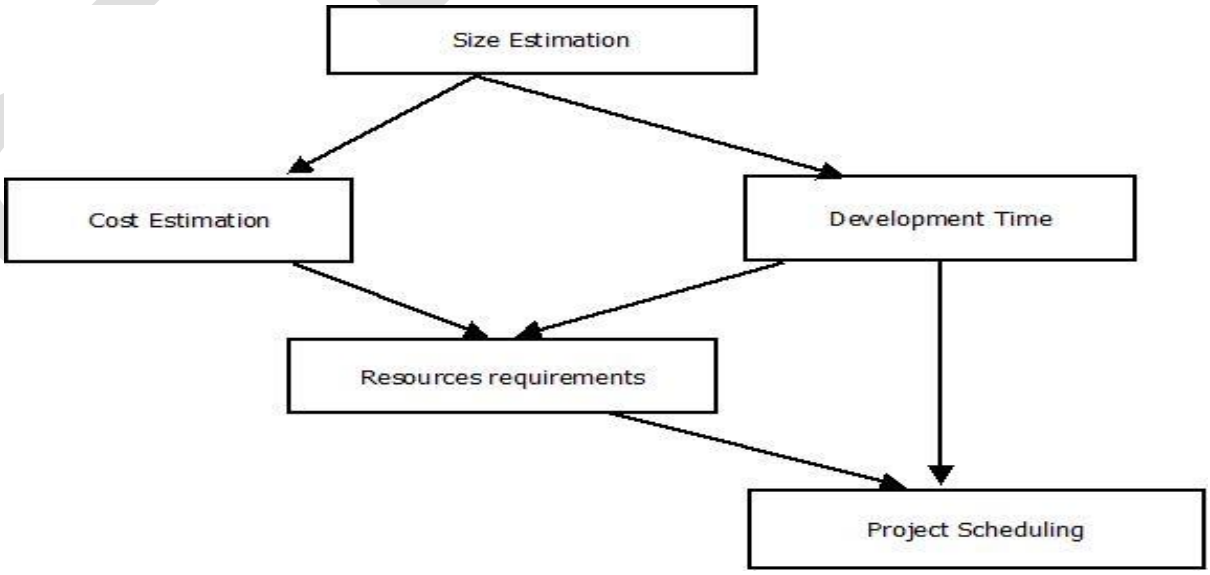
Software Scope- It describes the functions and features that are to be delivered to end users, the data that are input and output, the content that is presented to users as a consequence of using the software and the performance constraints interfaces and reliability that bound the system. Scope is defined by using 2 techniques

1. A narrative description of software scope is developed after communication with stakeholders
2. A set of use cases is developed by end users

- Functions described in the statement of scope (or within the use cases) are evaluated and in some cases refined to provide more detail prior to the beginning of estimation.
- Once scope is understood the software team and others must work to determine if it can be done within the dimensions just noted.

III. Emprical Estimation models(Ref- Software Engg, K.K Aggarwal ,Yogesh Singh)

- Software planning begins before technical work starts, continues as the software Evolves from concept to reality, and culminates only when the software is retired.



Software planning activities

Cost Estimation

- Approximate judgement of the costs for a project. It should be done throughout the entire life cycle.
- Why we need?
- To determine how much effort and time a software project requires.

- Important for making good management decisions.
- It facilitates competitive contract bids.
- It affect the planning and budgeting of a project.

- To determine the cost
  - Determine size of the product.
  - From the size estimate,
    - determine the effort needed.
  - From the effort estimate,
    - determine project duration, and cost.
- Three main approaches to estimation:
  - Empirical
  - Heuristic
  - Analytical
- Empirical techniques:
  - an educated guess based on past experience.
- Heuristic techniques:
  - assume that the characteristics to be estimated can be expressed in terms of some mathematical expression.
- Analytical techniques:
  - derive the required results starting from certain simple assumptions.

Emprical Estimation Model

- The structure of empirical estimation models is a formula, derived from data collected from past software projects, that uses software size to estimate effort. Size, itself, is an estimate, described as either lines of code (LOC) or function points (FP).
- No estimation model is appropriate for all development environments, development processes, or application types.
- Models must be customised (values in the formula must be altered) so that results from the model agree with the data from the particular environment. The typical formula of estimation models is:

$$E = a + b(S)c$$

where; E represents effort, in person months,
S is the size of the software development, in LOC or FP, and,
a, b, and c are values derived from data.

Static, Single Variable Models

Methods using this model use an equation to estimate the desired values such as cost, time, effort, etc. They all depend on the samevariable used as predictor (say, size). An example of the most common equations is :

$$C=a\ L^b$$

C is the cost ,L is the size and a,b are constants

$$E=1.4\ L^{0.93}$$

$$DOC = 30.4\ L^{0.90}$$

$$\mathbf{D = 4.6\ L^{0.26}}$$

Effort (E in Person-months),
documentation (DOC, in number ofpages)
and duration (D, in months) are calculated from the number of lines of code (L, in thousands of lines) used as a predictor.

IV . COCOMO Model(Ref Software engineering K.K. Aggarwal P No-152-161)

- The COCOMO model is a single variable software cost estimation model developed by Barry Boehm in 1981.
- The model uses a basic regression formula, with parameters that are derived from historical project data and current project characteristics.
- Divides software product developments into 3 categories:
  - Organic - Relatively small groups .working to develop well-understood applications.
  - Semidetached - Project team consists of a mixture of experienced and inexperienced staff
  - Embedded- The software is strongly coupled to complex hardware, or real-time systems.
- For each of the three product categories:
- From size estimation (in KLOC), Boehm provides equations to predict:
  - project duration in months
  - effort in programmer-months
- Boehm obtained these equations: examined historical data collected from a large number of actual projects.
- Software cost estimation is done through three stages:
- Basic COCOMO,
- Intermediate COCOMO,
- Complete COCOMO.

Mode	*Project size*	*Nature of Project*	*Innovation*	*Deadline of the project*	*Development Environment*
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

Basic Model

Basic COCOMO model takes the form

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

$$D = c_b (E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in table 4 (a).

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 4(a): Basic COCOMO coefficients

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size } (SS) = \frac{E}{D} \; Persons$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity } (P) = \frac{KLOC}{E} \; KLOC / PM$$

Example

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project

The semi-detached mode is the most appropriate mode; keeping in view the size, schedule and experience of the development team. Software Project Planning Average staff size

E = 3.0(200)1.12 = 1133.12 PM

D = 2.5(1133.12)0.35 = 29.3 PM

Average Staff size=E/D Persons

=1133.12/29.3

= 38.67 Persons

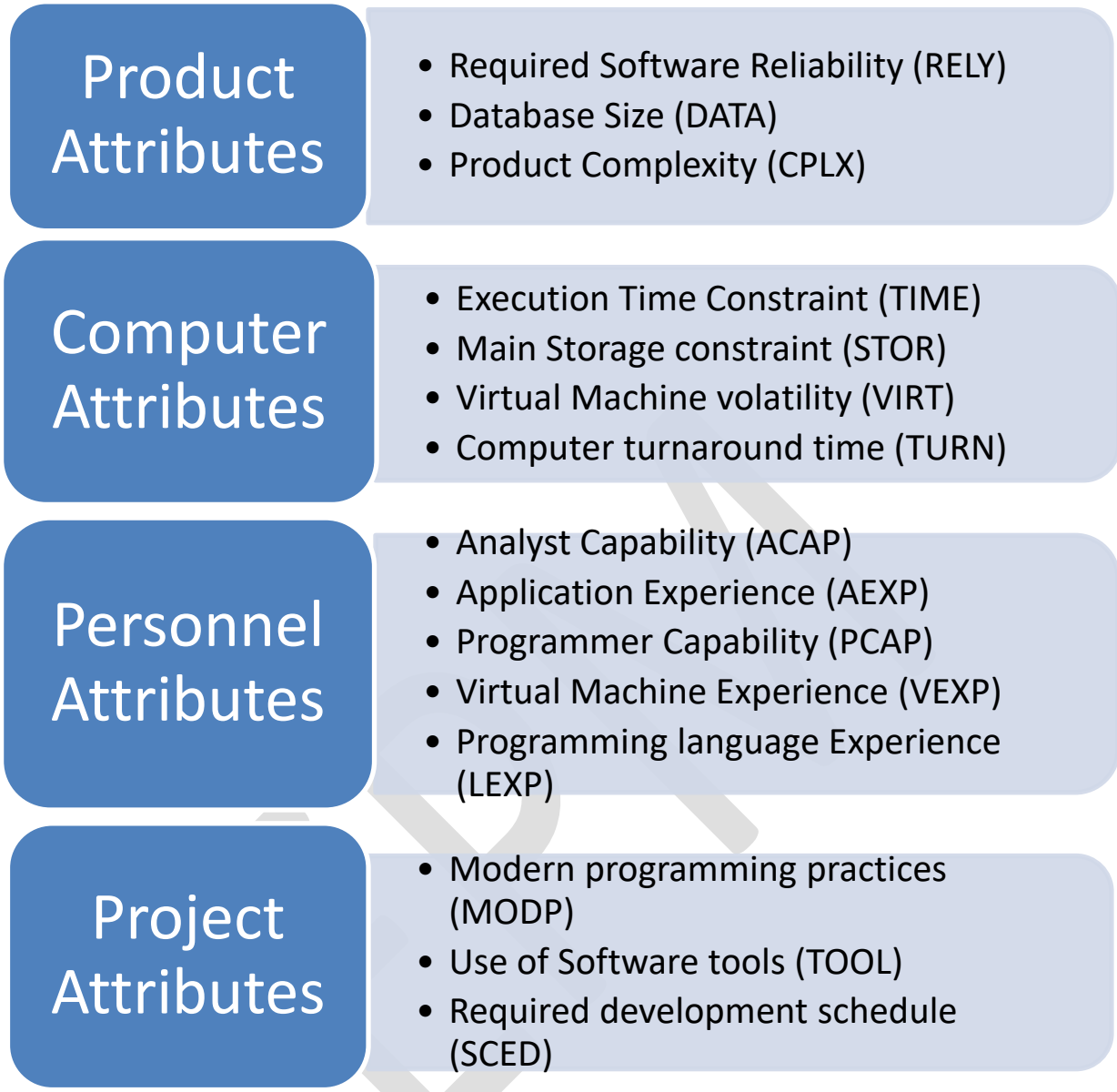
Productivity= KLOC/E= 200/1133.12

= 176 LOC /PM

Intermediate COCOMO

- Basic COCOMO model assumes
  - effort and development time depend on product size alone.
- However, several parameters affect effort and development time:
  - Reliability requirements
  - Availability of CASE tools and modern facilities to the developers
  - Size of data to be handled
- For accurate estimation,
  - the effect of all relevant parameters must be considered:
  - Intermediate COCOMO model recognizes this fact:
    - refines the initial estimate obtained by the basic COCOMO by using a set of 15 cost drivers (multipliers).
  - Multiply all 15 Cost Drivers to get **Effort Adjustment Factor(EAF)**
  - **E(Effort) = $a_b(KLOC)^{b}_{b}$ * EAF**(in Person-Month)
  - **D(Development Time) = $c_b(E)^{d}_{b}$** (in month)
  - **SS (Avg Staff Size) = E/D** (in persons)
  - **P (Productivity) = KLOC/E** (in KLOC/Person-month)

Project	a_b	b_b	c_b	d_b
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32



Intermediate COCOMO Example

A new project with estimated 400 KLOC embedded system has to be developed. Project manager hires developers of low quality but a lot of experience in programming language. Calculate the Effort, Development time, Staff size & Productivity.

Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
AEXP	**1.29**	1.13	1.00	0.91	0.82	--
LEXP	1.14	1.07	1.00	**0.95**	--	--

EAF = 1.29 * 0.95 = 1.22

400 LOC implies Embedded System

Effort = $2.8*(400)^{1.20}$ * 1.225 = 3712 * 1.22 = 4528 person-months

Development Time = $2.5 * (4528)^{0.32}$ = 2.5 * 14.78 = 36.9 months

Avg. Staff Size = E/D = 4528/36.9 = 122 persons

Productivity = KLOC/Effort = 400/4528 = 0.0884 KLOC/person-month

- **Detailed COCOMO** = Intermediate COCOMO + assessment of Cost Drivers impact on each phase.
- Phases
    1) Plans and requirements
    2) System Design
    3) Detailed Design
    4) Module code and test
    5) Integrate and test

➢ Cost of each subsystem is estimated separately. This reduces the margin of error.

➢ Multiply all 15 Cost Drivers to get **Effort Adjustment Factor(EAF)**

- **E(Effort) = $a_b(KLOC)^{b}_{b}$ * EAF**(in Person-Month)
- **D(Development Time) = $c_b(E)^{d}_{b}$** (in month)
- **E_p (Total Effort) = μ_p * E** (in Person-Month)
- **D_p (Total Development Time) = τ_p * D** (in month)

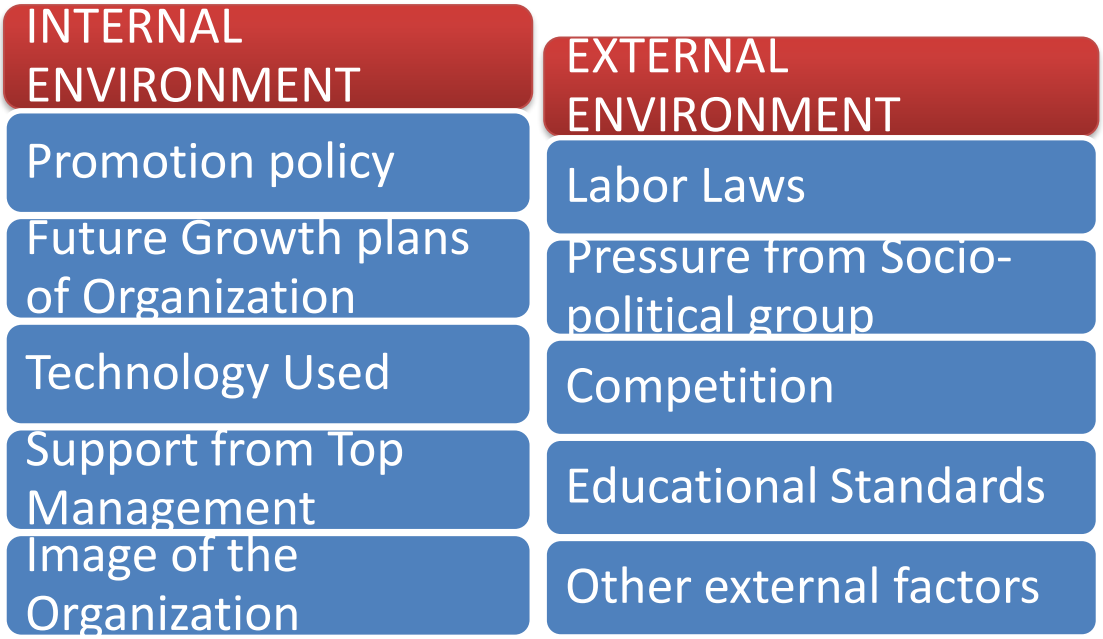
IV. Staffing and Personnel Planning(Ref Software Enginnering IAN Sommerville 7th edition P No-613)

- **Staffing-** The managerial function of staffing involves manning the organization structure through proper and effective selection, appraisal and development of the personnel to fill the roles assigned to the employers/workforce.

Nature of Staffing

- Staffing means filling and keeping filled, positions in the organisation structure."
- Staffing is an important managerial function.
- Staffing is a pervasive activity.
- Staffing is a continuous activity.
- The basis of staffing function is efficient management of personnel.
- Staffing helps in placing right men at the right job.
- Staffing is performed by all managers .

Factors Affecting Staffing



Staffing Process

- Manpower
- Recruitment
- Selection
- Orientation and Placement
- Training and Development
- Remuneration
- Performance Evaluation
- Promotion and transfer

Manpower Planning

- Manpower Planning which is also called as Human Resource Planning consists of putting right number of people, right kind of people at the right place, right time, doing the right things for which they are suited for the achievement of goals of the organization.

Steps in Manpower planning

- **Analysing the current manpower**
  - Type of organization
  - Number of departments
  - Number and quantity of such departments
  - Employees in these work units
- **Making future manpower forecasts-**
  - Expert Forecasts
  - Trend Analysis
  - Work Load Analysis
  - Work Force Analysis
- **Developing employment programmes**

- **Design training programmes**

Obstacles in Man Power Planning

- Under Utilization of Manpower
- Degree of Absenteeism
- Lack of Education and Skilled Labour
- Manpower Control and Review
  - Productivity = Output / Input.
  - Employee Productivity = Total Production / Total no. of employees

Types of Recruitment

1. Internal Recruitment- is a recruitment which takes place within the concern or organization. Internal sources of recruitment are readily available to an organization.
   a)Transfers
   b)Promotions
   c)Re-employment of ex-employees

2. External Recruitment- External sources of recruitment have to be solicited from outside the organization. But it involves lot of time and money.
   a)Employment at factory level
   b)Advertisement
   c)Employment exchanges
   d) Employment agencies
   e)Educational Institutions
   f)Recommendations
   g)Labor contractors

Employee Selection Process

Employee Selection is the process of putting right men on right job. It is a procedure of matching organizational requirements with the skills and qualifications of people.

Different process are

a. Interview- Every manager hired or promoted by a company is interviewed by one or more people. Techniques used to improve the interviewing process-
   -Interviewer-What to look for?
   -Should be prepared to ask the right questions
   -Conducting multiple interviews
   -Just one aspect of selection

b. Test - Primary aim of testing is to obtain data about applicants that help predict their probable success as managers.Some of the commonly used tests-
   Intelligence tests,Proficiency and aptitude tests,Vocational tests,Personality tests

c. Assessment Centers - A technique for selecting and promoting managers.

The usual center approach is to have candidates take part in a series of exercises.

- During this period, they are observed and assessed by psychologists or experienced managers. A typical assessment center-
  a)Various psychological tests
  b)Management games

Difference Between Recruitment and Selection

Recruitment	Selection
It is an activity of establishing contact between employers and applicants.	It is a process of picking up more competent and suitable employees.
It encourages large number of Candidates for a job.	It attempts at rejecting unsuitable candidates.
The candidates have not to cross over many hurdles.	Many hurdles have to be crossed.
It is a positive approach.	It is a negative approach.
It proceeds selection.	It follows recruitment.

- **Placement-**Once the candidates are selected for the required job, they have to be fitted as per the qualifications.
- Placement is said to be the process of fitting the selected person at the right job or place, i.e. fitting square pegs in square holes and round pegs in round holes.
- Once he is fitted into the job, he is given the activities he has to perform and also told about his duties.
- **Orientation-** During Orientation employees are made aware about the mission and vision of the organization
- Generally the information given during the orientation programme includes-
  - ✓ Employee's layout
  - ✓ Type of organizational structure
  - ✓ Departmental goals
  - ✓ Organizational layout
  - ✓ General rules and regulations
  - ✓ Standing Orders
  - ✓ Grievance system or procedure
  - **Training of Employees-** Training of employees takes place after orientation takes place. Training is the process of enhancing the skills, capabilities and knowledge of employees for doing a particular job.
  - Training process moulds the thinking of employees and leads to quality performance of employees. It is continuous and never ending in nature.

Importance of Training-Training is crucial for organizational development and success.

- It is fruitful to both employers and employees of an organization. An employee will become more efficient and productive if he is trained well.
- It also-
  - ✓ Improves morale of employees
  - ✓ Less supervision
  - ✓ Chances of promotion
  - ✓ Increased productivity

Employee Remuneration- refers to the reward or compensation given to the employees for their work performances.

- Remuneration provides basic attraction to a employee to perform job efficiently and effectively.
- There are mainly two types of Employee Remuneration
  - ✓ Time Rate Method- Hourly basis
  - ✓ Piece Rate Method- Fixed for a period

V. Design Phase(Ref Pressman)

- Software design is an iterative process.
- The requirements are translated into a blueprint for constructing the software.
- Quality- Important factor of design

Design Quality Attributes

1) A design should exhibit an architecture – Components and implemented in evolutionary fashion
2) A design should be modular-
3) A design should contain distinct representations of data, architecture, interfaces, and components
4) A design should lead to data structures that are appropriate for the classes to be implemented
5) A design should lead to components that exhibit independent functional characteristics
6) A design should lead to interfaces that reduce the complexity of connections
7) A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis
8) A design should be represented using a notation that effectively communicates its meaning

Goals of a Good Design

- The design must implement all of the explicit requirements and implicit requirements desired by the customer
- Must be a readable and understandable guide for those who generate code, and for those who test and support the software
- should provide a complete picture of the software, addressing the data, functional, and behavioural domains from an implementation perspective.

Some of the quality attributes are

- Functionality
- Usability
- Reliability
- Performance-Response time, resource consumption etc
- Supportability- Adaptability

Design Concepts and Principles(Ref- Software Engineering a practitioners approach by pressman p no- 265,chapter 9)

- 1. Abstraction
  - Procedural abstraction – a sequence of instructions that have a specific and limited function.
  - The specific details are suppressed
  - Data abstraction – a named collection of data that describes a data object

-- Hides the details of data attributes

- 2. Architecture
  - The overall structure of the software and the ways in which the structure provides conceptual integrity for a system
  - Consists of components, connectors, and the relationship between them

There are different Architectural Models to represent the architecture of software. They are

- Structural model-Organized Collection of program components
- Framework Model- Same frameworks may be used in similar types of applications
- Dynamic Model- Behavioral aspects of the program architecture
- Process Model- Design on the technical process that the system must accommodate
- Functional model-Functional hierarchy of a system

3. Patterns
  - A design structure that solves a particular design problem within a specific context
  - It provides a description that enables a designer to determine whether
    - the pattern is applicable,
    - whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns

- 4.Modularity
  - Separately named and addressable components (i.e., modules) that are integrated to satisfy requirements (divide and conquer principle)
  - Makes software intellectually manageable so as to grasp the control paths, span of reference, number of variables, and overall complexity
- 5. Information hiding
  - The designing of modules so that the algorithms and local data contained within them are inaccessible to other modules
  - This enforces access constraints to both procedural (i.e., implementation) detail and local data structures
- 6. Functional independence
  - Modules that have a "single-minded" function
  - High cohesion – a module performs only a single task

- Low coupling – a module has the lowest amount of connection needed with other modules

- 7. Stepwise refinement- Top down strategy
  - Development of a program by successively refining levels of procedure detail
  - Complements abstraction, helps to reveal the lower level details as design progress
- 8. Refactoring
  - A reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behavior
  - Removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures
- 9. Design classes
  - The classes to be implemented
  - Creates a new set of design classes that implement a software infrastructure to support the business solution
- The different types of design classes are
- **User interface classes** – define all abstractions necessary for human-computer interaction (usually via metaphors of real-world objects)
- **Business domain classes** – identify attributes and services (methods) that are required to implement some element of the business domain
- **Process classes** – implement business abstractions required to fully manage the business domain classes
- **Persistent classes** – represent data stores (e.g., a database) that will persist beyond the execution of the software
- **System classes** – implement software management and control functions that enable the system to operate and communicate within its computing environment and the outside world
- Characteristics of well defined design classes are
- Complete and sufficient
  - Contains the complete encapsulation of all attributes and methods that exist for the class
  - Contains only those methods that are sufficient to achieve the intent of the class
- Primitiveness
  - Each method of a class focuses on accomplishing one service for the class
- High cohesion
  - The class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities
- Low coupling
  - Collaboration of the class with other classes is kept to an acceptable minimum

- Each class should have limited knowledge of other classes in other subsystems

Effective Modular Design(Ref Software engineering K.K. Aggarwal)

Modular design

- Reduces complexity
- Facilitates change
- Results in easier implementation by supporting parallel development of different parts of the system.

Functional independence is achieved by developing modules with: Single minded function.

MODULARITY

There are many definitions of the term module. Range is from :

- Fortran subroutine
- Ada package
- Procedures & functions of PASCAL & C
- C++ / Java classes

A modular system consist of well defined manageable units with well defined interfaces among the units.

Properties :

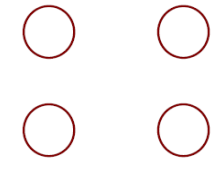
- Well defined subsystem
- Well defined purpose
- Can be separately compiled and stored in a
- library.
- Module can use other modules
- Module should be easier to use than to build
- Simpler from outside than from the inside.

➢ Modularity is the single attribute of software that allows a program to be intellectually manageable.

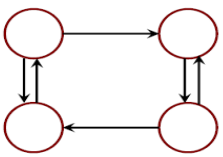
➢ It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of software product.

➢ Module Coupling

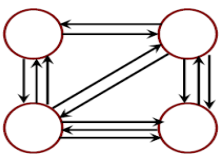
➢ Coupling is the measure of the degree of interdependence between modules.



(Uncoupled : no dependencies)



Loosely coupled:
some dependencies
(B)

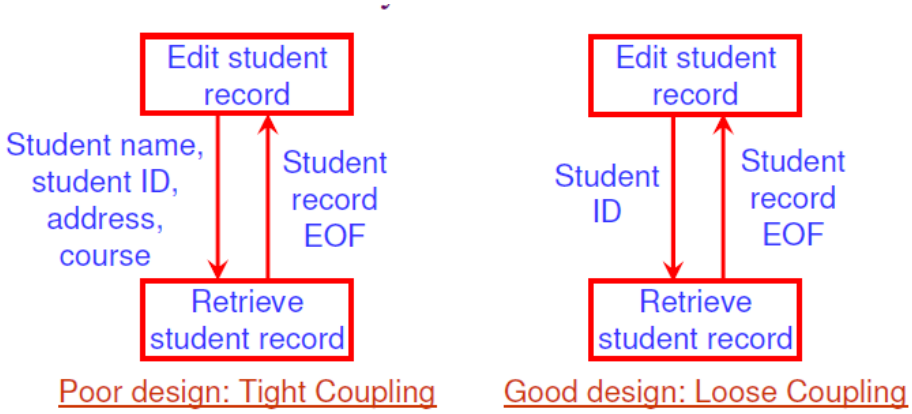


Highly coupled:
many dependencies
(C)

This can be achieved as:

- Controlling the number of parameters passed amongst modules.
- Avoid passing undesired data to calling module.
- Maintain parent / child relationship between calling & called modules.
- Pass data, not the control information.

Consider the example of editing a student record in a 'student information system'.



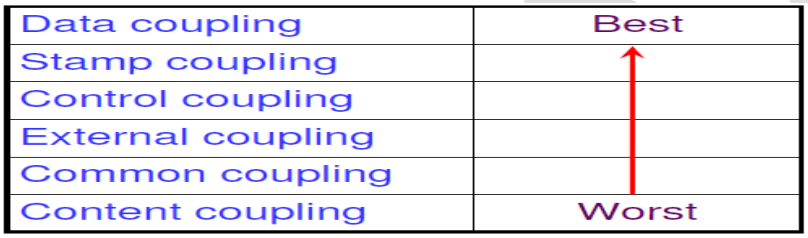


Fig. 7 : The types of module coupling

- Given two procedures A & B, we can identify number of ways in which they can be coupled.
- **Data coupling-**The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent.
- **Stamp coupling-** Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.
- **Control coupling-** Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.
- **Common coupling-** With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of changes.
- **Content coupling-** Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another.

Module Cohesion

- Cohesion is a measure of the degree to which the elements of a module are functionally related.

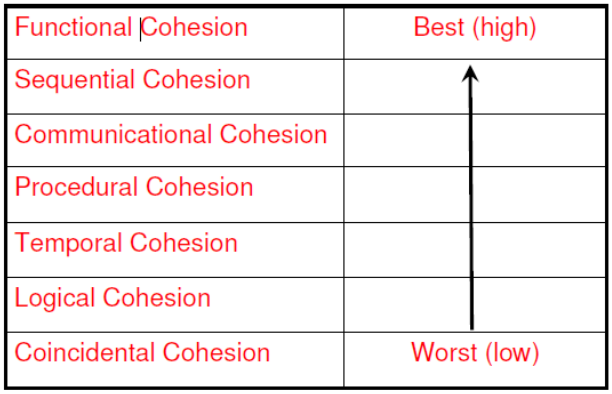


Fig. 11 : Types of module cohesion

- **Functional cohesion-** A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure.
- **Sequential cohesion-**Module A outputs some data which forms the input to B. This is the reason for them to be contained in the same procedure
- **Procedural cohesion-**Procedural Cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed**.**
- **Temporal cohesion -**Module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span.
- **Logical cohesion**- Logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions**.**
- **Coincidental cohesion-** Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.

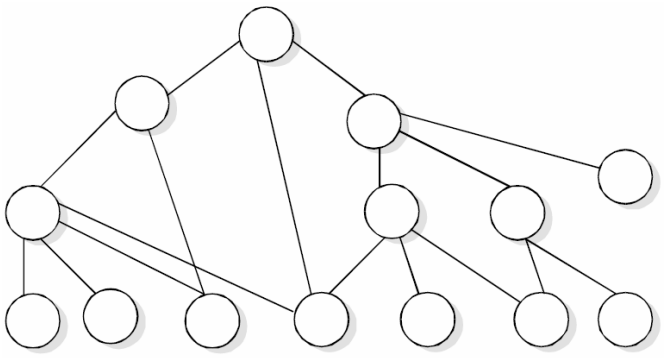
Relationship between Cohesion & Coupling- If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

STRATEGY OF DESIGN- A good system design strategy is to organize the program modules in such a way that are easy to develop and latter to, change.

- Structured design techniques help developers to deal with the size and complexity of programs.
- Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. It is important for two reasons:
    - First, even pre-existing code, if any, needs to be understood, organized and pieced together.
    - Second, it is still common for the project team to have to write some code and produce original programs that support the application logic of the system.

Bottom-Up Design

- These modules are collected together in the form of a “library”.

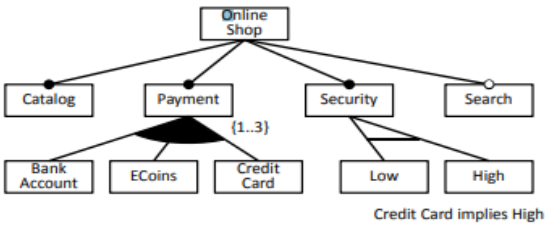


Bottom up tree structure

- To identify modules that are required by many programs
- These modules are collected together in the form of a library
- Here we combine the common modules to form the larger ones.
- Since the design progressed from bottom layer upwards the method is called bottom up design
- Draw Back-Need to use a lot of intuition to decide exactly what functionality a module should provide.
- If the assumption is wrong then we have to redesign from alower level

Top Down Design

- Starts by identifying the major modules of the system. Decomposing them into their lower level modules and iterating until the desired level of detail is achieved.
- This is also known as stepwise refinement. Starting from an abstract design in each step the design is refined to more concrete level, until no more refinement is needed.
- Most design methodologies are based on this approach.
- Suitable- if specifications are clear and development is from the scratch.
- Weakness- can be tested only after all the subordinate modules are coded.



Top down Approach

Hybrid Approach

- For top-down approach to be effective, some bottom-up approach is essential for the following reasons:
- Software Design To permit common sub modules.
- Near the bottom of the hierarchy, because there are more number of modules at low levels than high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

Design Approaches

- **Function Oriented Design**- is decomposed into set of interacting units where each unit has clearly defined function
- Conceals the details of an algorithm in a function but system state information is not hidden.
- The activities of this strategy;
    - Data-flow design
    - Structural decomposition
    - Detailed design description
- **Object-oriented design** - Is based on the idea of information hiding.
- System is viewed as a set of interacting objects, with their own private state.
- Dominant design strategy for new software systems.
- Objects communicate by calling on services offered by other objects rather than sharing variables. This reduces the overall system coupling.
- Message passing model allows objects to be implemented as concurrent processes.
- Kinds of concurrent object implementation
    - Passive objects
    - Active objects

Coding – programming practice, verification, size measures, complexity analysis, coding standards. Testing – fundamentals, white box testing, control structure testing, black box testing, basis path testing, code walk-through and inspection, testing strategies-Issues, Unit testing, integration testing, Validation testing, System testing.

1.Coding

- Any organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously.
- The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:
  - A coding standard gives a uniform appearance to the codes written by different engineers.
  - It enhances code understanding.
  - It encourages good programming practices.
- A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

2. Coding standards and guidelines

- Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they are going to develop.
- The following are some representative coding standards.
  1. **Rules for limiting the use of global:** These rules list what types of data can be declared global and what local and what not.
  2. **Contents of the headers preceding codes for different modules:** The information contained in the headers of different modules should be standard for an organization.
     - The exact format in which the header information is organized in the header can also be specified.
     - The following are some standard header data:
       - Name of the module.
       - Date on which the module was created.
       - Author's name.
       - Modification history.
       - Synopsis of the module.
       - Different functions supported, along with their input/output parameters.
       - Global variables accessed/modified by the module.

3. **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
4. **Error return conventions and exception handling mechanisms:** The way error conditions are reported by different functions in a program are handled should be standard within an organization.
   - For example, different functions while encountering an error condition should either return a 0 or 1 consistently. The following are some representative coding guidelines recommended by many software development organizations.
5. **Do not use a coding style that is too clever or too difficult to understand:** Code should be easy to understand. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.
6. **Avoid obscure side effects:** The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. Obscure side effects make it difficult to understand a piece of code.
   - For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.
7. **Do not use an identifier for multiple purposes:** Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result.
   - For such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong with this approach and hence should be avoided
   - Some of the problems caused by use of variables for multiple purposes as follows:
     • Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes.
     - Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.

     • Use of variables for multiple purposes usually makes future enhancements more difficult.
8. **The code should be well-documented:** As a rule of thumb, there must be at least one comment line on the average for every three-source line.
9. **The length of any function should not exceed 10 source lines:** A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions.
   - For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.
10. **Do not use goto statements**: Use of goto statements makes a program unstructured and makes it very difficult to understand.

3. Code review : Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated.

- Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module.
- These two types code review techniques are
    - code inspection
    - and code walk through.

- **Code Walk Throughs**
    - Code walk through is an informal code analysis technique.
    - In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated. A few members of the development team are given the code few days before the walk through meeting to read and understand code.
    - Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution).
    - The main objectives of the walk through are to discover the algorithmic and logical errors in the code.
    - The members note down their findings to discuss these in a walk through meeting where the coder of the module is present. Even though a code walks through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective.
    - Therefore, these guidelines should be considered as examples rather than accepted as rules to be applied dogmatically. Some of these guidelines are the following

• The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.

• Discussion should focus on discovery of errors and not on how to fix the discovered errors.

• In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting. Managers should not attend the walk through meetings.

Code Inspection

- In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.
- During code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk through.
- In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

- Following is a list of some classical programming errors which can be checked during code inspection:
    - Use of uninitialized variables.
    - Jumps into loops.
    - Non-terminating loops.

- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and de allocation.
- Mismatches between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equally of floating point variables, etc.

4. Verificat

➢ Verification and Validation

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .

Testing= Verification+Validation

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- Verification (Are the algorithms coded correctly?)
  - The set of activities that ensure that software correctly implements a specific function or algorithm
- Validation (Does it meet user requirements?)
  - The set of activities that ensure that the software that has been built is traceable to customer requirements

5. Size Measure

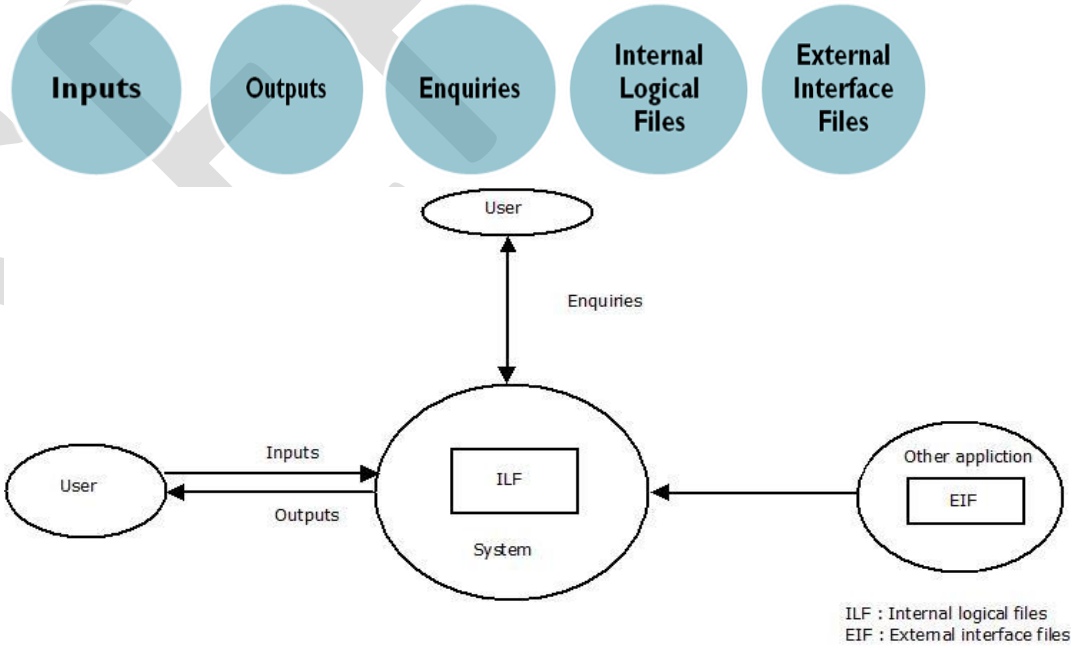
Size estimation: The estimation of size is very critical and difficult area of project planning. It indicates the size of the project. Mainly two methods

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

```
1.  int. sort (int x[ ], int n)
2.  {
3.  int i, j, save, im1;
4.  /*This function sorts array x in ascending order */
5.  If (n<2) return 1;
6.  for (i=2; i<=n; i++)
7.  {
8.  im1=i-1;
9.  for (j=1; j<=im; j++)
10. if (x[i] < x[j])
11. {
12. Save = x[i];
13. x[i] = x[j];
14. x[j] = save;
15. }
16. }
17. return 0;
18. }
```

- If LOC is simply a count of the number of lines then figure shown below contains 18 LOC .
- When comments and blank lines are ignored, the program in figure 2 shown below contains 17 LOC.
- The size of the program for specific functionality- to include executable statements. The only executable statements in figure shown above are in lines 5-17 leading to a count of13.

b. Function Point- Function point measures functionality which is a solution to the size measurement problem. It is decomposed into following functional units



Functional units	Low	Average	High
External Inputs	3	4	6

External Outputs	4	5	7
External Inquiries	3	4	5
Internal logical files (ILF)	7	10	15
External interface files (EIF)	5	7	10

Functional unit with weighing factors

The procedure for the calculation of UFP in mathematical form is given below

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} Z_{ij} Wij$$

UFP : Unadjusted Function Point

Where i indicates the row and j indicates the column of table.
W_{ij} : It is the entry of i^{th} row and j^{th} column of the table.
Z_{ij}: It is the count of the number of functional unit of type i that have been classified as having the complexity corresponding to column j

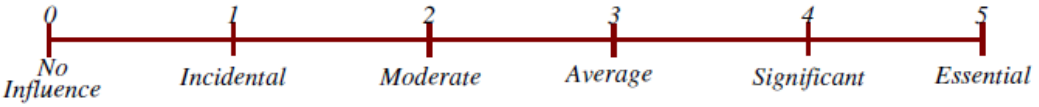
Organizations that use function point methods develop a criterion for determining whether a particular entry is Low, Average or High. Nonetheless, the determination of complexity is somewhat subjective.

$$FP = UFP * CAF$$

Where CAF is complexity adjustment factor and is equal to [0.65 + 0.01 x ΣF_i]. The F_i (i=1 to 14) are the degree of influence and are based on responses to questions noted in table 3.

Table 3 : Computing function points.

Rate each factor on a scale of 0 to 5.



Number of factors considered (F_i)

1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated on line ?
9. Is the inputs, outputs, files, or inquiries complex ?
10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

Functions points may compute the following important metrics:

Productivity	=	FP / persons-months
Quality	=	Defects / FP
Cost	=	Rupees / FP
Documentation	=	Pages of documentation per FP

Example: 4.1

Consider a project with the following functional units:

Number of user inputs	= 50
Number of user outputs	= 40
Number of user enquiries	= 35
Number of user files	= 06
Number of external interfaces	= 04

Assume all complexity adjustment factors and weighting factors are average. Compute the function points for the project.

Solution

We know

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

UFP = 50 x 4 + 40 x 5 + 35 x 4 + 6 x 10 + 4 x 7
= 200 + 200 + 140 + 60 + 28 = 628

CAF = $(0.65 + 0.01 \Sigma F_i)$
= (0.65 + 0.01 (14 x 3)) = 0.65 + 0.42 = 1.07

FP = UFP x CAF
= 628 x 1.07 = 672

Example:4.2

An application has the following:

10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 high external interface files, 12 average external inquiries, and a value of complexity adjustment factor of 1.10.

What are the unadjusted and adjusted function point counts ?

Solution

Unadjusted function point counts may be calculated using as:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

$= 10 \times 3 + 12 \times 7 + 20 \times 7 + 15 + 10 + 12 \times 4$
$= 30 + 84 + 140 + 150 + 48$
$= 452$

FP $= UFP \times CAF$
$= 452 \times 1.10 = 497.2.$

6.Testing Fundamentals

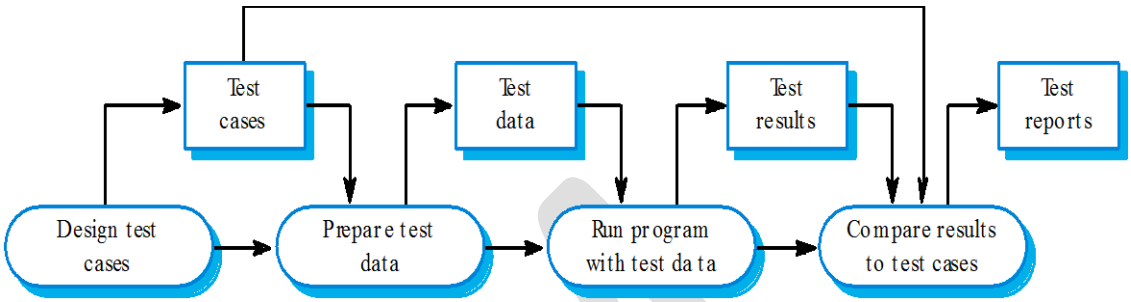
- **The testing Process-**

- Component testing
  - Testing of individual program components;
  - Usually the responsibility of the component developer;
  - Tests are derived from the developer's experience.
- System testing
  - Testing of groups of components integrated to create a system or sub-system;
  - The responsibility of an independent testing team;
  - Tests are based on a system specification.

Goals of Testing Process

- Validation testing
  - To demonstrate to the developer and the system customer that the software meets its requirements;
  - A successful test shows that the system operates as intended.
- Defect testing
  - To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification;

- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.
- The goal of defect testing is to discover defects in programs
- A *successful* defect test is a test which causes a program to behave in an anomalous way
- Tests show the presence not the absence of defects



Testing Process

Testing Policies

- Testing policies define the approach to be used in selecting system tests:
  - All functions accessed through menus should be tested;
  - Combinations of functions accessed through the same menu should be tested;
  - Where user input is required, all functions must be tested with correct and incorrect input.

6. Testing Strategies

- A strategy for software testing integrates the design of software test cases into a well-planned series of steps that result in successful development of the software
- The strategy provides a road map that describes the steps to be taken, when, and how much effort, time, and resources will be required
- The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation
- The strategy provides guidance for the practitioner and a set of milestones for the manager
- Because of time pressures, progress must be measurable and problems must surface as early as possible

General Characteristics of Strategic Testing

- To perform effective testing, a software team should conduct effective formal technical reviews
- Testing begins at the component level and work outward toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects) by an independent test group

- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

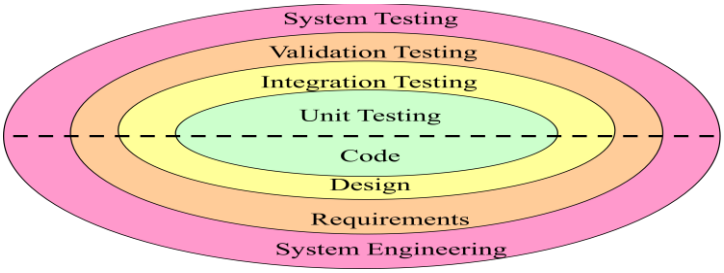


Fig: Strategy for testing conventional softwares

Levels of Testing for Conventional Softwares

- **Unit testing**
  - Concentrates on each component/function of the software as implemented in the source code
- **Integration testing**
  - Focuses on the design and construction of the software architecture
- **Validation testing**
  - Requirements are validated against the constructed software
- **System testing**
  - The software and other system elements are tested as a whole

Testing strategy applied to conventional software

- **Unit testing**
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
- **Integration testing**
  - Focuses on inputs and outputs, and how well the components fit together and work together
- **Validation testing**
  - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- **System testing**
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Strategic Issues in Testing

- Testing is a very important phase in software development life cycle. But the testing may not be very effective if proper strategy is not used.
- For the implementation of successful software testing strategy, the following issues must be taken care of: -
  - Before the start of the testing process, all the requirements must be specified in a quantifiable manner.
  - Testing objectives must be clarified and stated explicitly.

- A proper testing plan must be developed.
- Build "robust" software that is designed to test itself.
- Use effective formal technical reviews as a filter prior to testing. Formal technical reviews can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high-quality software.
- Conduct formal technical reviews to assess the test strategy and the cases themselves. Formal technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.
- Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

7. Unit Testing

- Concentrates on each component/function of the software as implemented in the source code

- Two strategies
  - White Box Testing- Structural Testing
  - Black box testing-Functional testing

White Box Testing- All statements and conditions have been executed atleast once.

Two types

- Basis Path Testing
- Control Structure Testing

8. Basis Path Testing

- Basis Path Testing- selecting a set of test paths through the program.
- Two Process
  1. generating a set of paths that will cover every branch in the program.
  2. finding a set of test cases that will execute every path in the set of program paths.
- The basis path testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once.
- A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.
- CFG describes the sequence in which the different instructions of a program get executed.
- To draw the CFG of a program, all the statements of a program must be numbered first.

Basis path Testing Steps

- Using the design or code as a foundation ,draw a corresponding flow graph
- Determine the cyclomatic complexity of resultant flow graph
- Determine a basis set of linearly independent paths.

- Prepare test cases that will force execution along each path

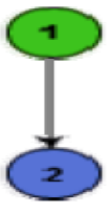
Example(Euclids GCD Computational Algorithm)

```
int compute_gcd(x, y)
int x, y;
{
    1  while (x! = y){
    2 if (x>y) then
    3 x= x – y;
    4 else y= y – x;
 }  5
    6 return x;
}
```

- **Step 1- Control Flow Graph**
    - CFG describes the sequence in which the different instructions of a program get executed.
    - To draw the CFG of a program, all the statements of a program must be numbered first.
    - The different numbered statements serve as nodes of the control flow graph.
    - An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node

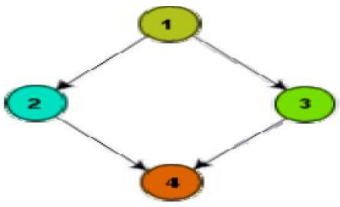
CFG for various types of Instructions

Sequence



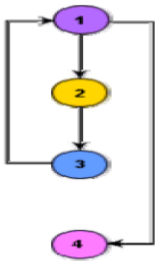
1. a=2
2.b=a*2

Selection



1.If a>b
2.c=3
3.Else c=5
4.c=c*c

Iteration



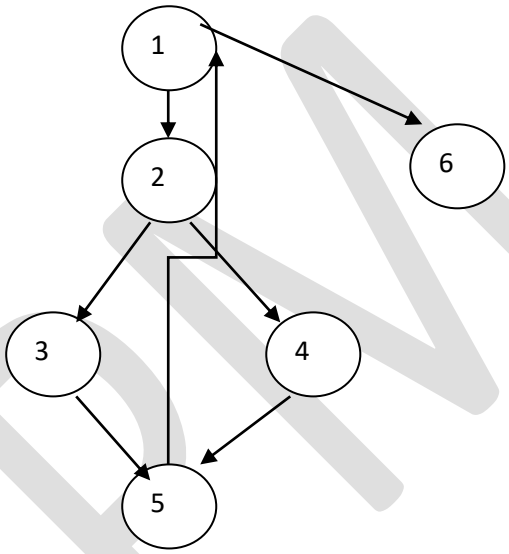
1.While (a>b){

2.b=b-1

3.b=b*a;}

4.c=a+b

CFG of GCD Algorithm is



Step 2: Determine Cyclomatic complexity

- Cyclomatic complexity(Structural Complexity of the program)

McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program.

- Method 1
- Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:
- V(G) = E – N + 2

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

- For the CFG of example shown in fig , E=7 and N=6. Therefore, the cyclomatic complexity = 7-6+2 = 3.
- Method 2
- If N is the number of decision statement of a program, or predicate nodes (each node that contain)then the McCabe's metric is equal to **N+1.**
- Method 3
- Another method is ,***V(G) =Total number of bounded areas or region +1***

Step 3: Determine a basis set of linearly independent paths.

- Number of independent paths: 3

- 1, 6
- 1, 2, 3, 5, 1, 6
- 1, 2, 4, 5, 1, 6

Step 4: Prepare test cases that will force execution along each path

- **1, 6 test case (x=1, y=1)**
- **1, 2, 3, 5, 1, 6 test case(x=1, y=2)**
- **1, 2, 4, 5, 1, 6 test case(x=2, y=1)**

9. Control Structure Testing

- Consists of three types of testing
  - Condition Testing
  - Data Flow testing
  - Loop Testing

1. Condition Testing

- It is a test case design method that exercise the logical conditions contained in a program module .
- Compound condition include (OR ,AND,or NOT statements)
- The condition testing method focus on testing each conditions in the program to ensure that it does not contain errors
- For Euclid's GCD computation algorithm , the test cases for branch coverage can be {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}.

```
int compute_gcd(x, y)
int x, y;
{
        1  while (x! = y){
        2 if (x>y) then
        3 x= x – y;
        4 else y= y – x;
 }
        6 return x;
}
```

- Test cases are designed to make each component of a composite conditional expression to assume both true and false values.
- For example, in the conditional expression ((c1.and.c2).or.c3), the components c1, c2 and c3 are each made to assume both true and false values

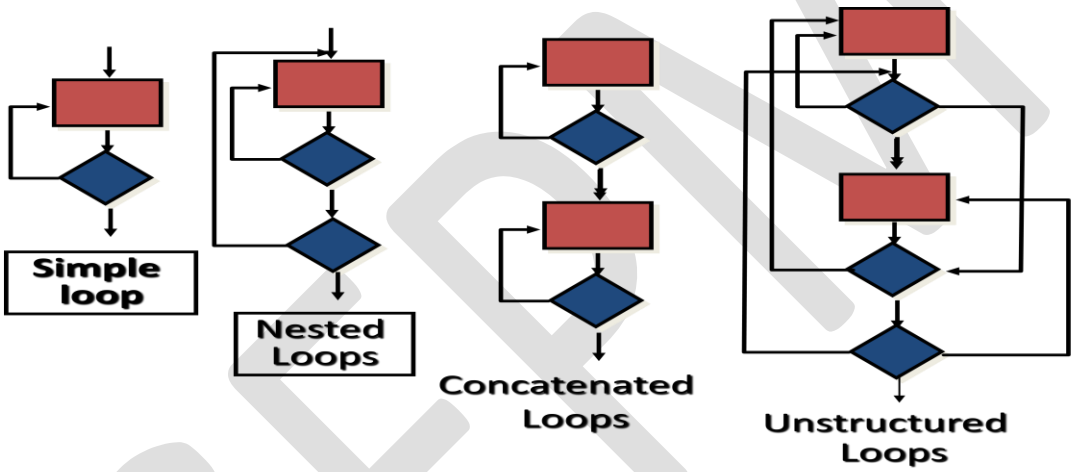
2. Data Flow Testing

It selects test path of a program according to the locations of definitions and uses of variables in the program.

- For a statement numbered S, let
- DEF(S) = {X/statement S contains a definition of X}, and
- USES(S) = {X/statement S contains a use of X}

- For the statement S:a=b+c;,
- DEF(S) = {a}. USES(S) = {b,c}.
- The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X.
- Definition – use chain is of the form [X,S,S'] where S and S' are statement number
- Every DU chain be covered at least ones.

3. **Loop Testing- 4 Types of loops**
   a. Simple loop
   b. Nested Loop
   c. Concatenated Loop
   d. Unstructured Loop



1. **Simple Loop**
   a. Skip the loop entirely
   b. One pass through the loop
   c. Two passes through the loop
   d. m passes through the loop where m<n
   e. n-1,n,n+1 passes through the loop
2. **Nested Loop**
   a. Start at the innermost loop. set all other loops to minimum value.
   b. Conduct the simple loop test for innermost loop
   c. Work outward. conducting tests for the next outer loop etc
   d. Continue until all loop have been tested
3. **Concatenated Loop**
   a. When the concatenated loops are independent take the approach of simple loop . Else take the approach of nested loop.

10 Black Box Testing

- Black Box Testing is also known as Behavioral Testing
- Focuses on the functional requirements of the software.
- Helps to identify
    - Incorrect or missing functions
    - Interface errors
    - Errors in data structures or external database access
    - Behavior or performance errors
    - Initialization and termination errors
- Unlike white-box testing, no knowledge about the internals of the code
- Test cases are designed based on specifications
    - Example: search for a value in an array
    - Postcondition: return value is the index of some occurrence of the value, or -1 if the value does not occur in the array
- We design test cases based on this spec
- Black Box Testing is of 4 types
    - Graph-Based Testing
    - Equivalence Class Partitioning
    - Boundary Value Analysis
    - Orthogonal Array Testing

1. Graph Based Testing-

- To understand the objects or functions
- Represent the objects with nodes
- Draw the links that represents the relationship between objects
- Graph Representation helps to identify
    - Transaction flow
    - Data Flow modelling etc

2. Equivalence Class Partitioning

- ivide input domain into classes of data
- Based on an evaluation of equivalence classes for an input condition
- Guidelines to define equivalence classes
    - Range input : One valid and two invalid equivalence
    - Specific value : One valid and two invalid equivalence
    - A member of a set : One valid and one invalid equivalence
    - Boolean : One valid and one invalid equivalence
- Basic idea: consider input/output domains and partition them into equiv. classes
    - For different values from the same class, the software should behave equivalently
- Use test values from each class

- Example: if the range for input **x** is 2..5, there are three classes: “<2”, “between 2..5”, “5<”
- Testing with values from different classes is more likely to uncover errors than testing with values from the same class

- Examples of equivalence classes
  - Input x in a certain range [a..b]: this defines three classes “x<a”, “a<=x<=b”, “b<x”
  - Input x is boolean: classes “true” and “false”
  - Some classes may represent invalid input

Example:

For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes:

1. The set of negative integers,
2. The set of integers in the range of 0 and 5000
3. The integers larger than 5000.

3.Boundary value Analysis

- A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs.
- For example, programmers may improperly use < instead of <=, or conversely <= for <.
- Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.
- Complement equivalence partitioning
- Test both sides of each boundary
- Look at output boundaries for test cases
- Test min, min-1, max, max+1, typical values
- Example : 1 <= x <=100
  - Valid : 1, 2, 99, 100
  - Invalid : 0 and 101

Example

- Suppose our spec says that the code accepts between 4 and 24 inputs, and each one is a 3-digit positive integer
- One dimension: partition the number of inputs
  - Classes are “x<4”, “4<=x<=24”, “24<x”
  - Chosen values: 3,4,5, 14, 23,24,25
- Another dimension: partition the integer values
  - Classes are “x<100”, “100<=x<=999”, “999<x”
  - Chosen values: 99,100,101, 500, 998,999,1000

4. Orthogonal Array Testing

- This technique is beneficial when we have to test with huge number data having many permutations and combinations.

- The beauty of this technique is that, it maximizes the coverage by comparatively lesser number of test cases. The pairs of parameters which are identified should be independent of each other.
- **Steps involved are**
    1. Identify the independent variables. These will be referred to as "Factors"
    2. Identify the values which each variable will take. These will be referred as "Levels"
    3. Search for an orthogonal array that has all the factors from step 1 and all the levels from step 2
    4. Map the factors and levels with our requirement
    5. Translate them into the suitable test cases
    6. Look out for the left over or special test cases (if any)

Example

We have to identify the test cases for a Web Page that has 4 sections: Headlines, details, references and Comments, that can be displayed or not displayed or show Error message. We are required to design the test condition to test the interaction between different sections.

Solution

1. Number of independent variables (factors) are = 4

2. Value that each variable can take = 3values(displayed, not displayed and error message)

3. find an appropriate array for 4 factors and 3 levels.

4. Select an orthogonal array for 3 levels and 4 factore

Table 2. $L_9(3^4)$ Standard orthogonal array.

Experiment no.	Factor A	Factor B	Factor C	Factor D
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

5. Map the requirements as below

    1 will represent "Is Displayed" value

    2 will represent "not displayed" value

    3 will represent "error message value"

    Factor A will represent " Headlines" section

    Factor B will represent "Details" section

Factor C will represent “references ”section
Factor D will represent “Comment” section.
Experiment no will represent “Test Cases #”

6. After mapping, the table will look like:

Test Case	Headlines	Details	References	Comments
TC01	is displyed	is displyed	is displyed	is displyed
TC02	is displyed	not displyed	not displyed	not displyed
TC03	is displyed	error	error	error
TC04	not displayed	is displyed	not displayed	error
TC05	not displayed	not displayed	error	is displyed
TC06	not displayed	error	is displyed	not displyed
TC07	error	is displyed	error	not displyed
TC08	error	not displayed	is displyed	error
TC09	error	error	not displayed	is displyed

7. Based on the table above, design our test cases. Also look out for the special test cases / left over test cases.

UNIT Testing

- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

Targets for unit test cases

- Module interface
  - Ensure that information flows properly into and out of the module
- Local data structures
  - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- Boundary conditions
  - Ensure that the module operates properly at boundary values established to limit or restrict processing
- Independent paths (basis paths)
  - Paths are exercised to ensure that all statements in a module have been executed at least once
- Error handling paths
  - Ensure that the algorithms respond correctly to specific error conditions

Common computational errors in execution path

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (int vs. float)
- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely (using == with float types)
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

Problems to uncover in error handling

- Error description is unintelligible or ambiguous
- Error noted does not correspond to error encountered
- Error condition causes operating system intervention prior to error handling
- Exception condition processing is incorrect
- Error description does not provide enough information to assist in the location of the cause of the error

Drivers and Stubs for Unit Testing

- Driver
  - A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
- Stubs
  - Serve to replace modules that are subordinate to (called by) the component to be tested
  - It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
- Drivers and stubs both represent overhead
  - Both must be written but don't constitute part of the installed software product

Integration Testing

- Defined as a systematic technique for constructing the software architecture
  - At the same time integration is occurring, conduct tests to uncover errors associated with interfaces

- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches
  - Non-incremental Integration Testing
  - Incremental Integration Testing

Non Incremental Integration Testing

- Commonly called the "Big Bang" approach
- All components are combined in advance
- The entire program is tested as a whole
- Chaos results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

Incremental integration Testing

- Three kinds
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration
- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

Top Down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated
- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

Bottom up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- Advantages
  - This approach verifies low-level data processing early in the testing process

  - Need for stubs is eliminated
- Disadvantages
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

Sandwich Integration

- Consists of a combination of both top-down and bottom-up integration
- Occurs both at the highest level modules and also at the lowest level modules
- Proceeds using functional groups of modules, with each group completed before the next
  - High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
  - Integration within the group progresses in alternating steps between the high and low level modules of the group
  - When integration for a certain functional group is complete, integration and testing moves onto the next group
- Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
- Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario

Regression Testing

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
  - Helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools
- Regression test suite contains three different classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed

SMOKE TESTING

- Taken from the world of hardware

  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

Benefits of Smoke Testing

- Integration risk is minimized
  - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- The quality of the end-product is improved
  - Smoke testing is likely to uncover both functional errors and architectural and component-level design errors
- Error diagnosis and correction are simplified
  - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
  - As integration testing progresses, more software has been integrated and more has been demonstrated to work

Managers get a good indication that progress is being made

Validation Testing

- Validation testing follows integration testing
- The distinction between conventional and object-oriented software disappears
- Focuses on user-visible actions and user-recognizable output from the system
- Demonstrates conformity with requirements
- Designed to ensure that
  - All functional requirements are satisfied
  - All behavioral characteristics are achieved
  - All performance requirements are attained
  - Documentation is correct

  - Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)
- After each validation test
  - The function or performance characteristic conforms to specification and is accepted
  - A deviation from specification is uncovered and a deficiency list is created
- A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle
- **Alpha testing**
  - Conducted at the developer's site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- **Beta testing**
  - Conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

SYSTEM Testing

- Recovery testing
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- Security testing
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- Stress testing
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance testing
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation

- Can uncover situations that lead to degradation and possible system failure

DEBUGGING Process

- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction

Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies
- **Brute force**
    - Most commonly used and least efficient method
    - Used when all else fails
    - Involves the use of memory dumps, run-time traces, and output statements
    - Leads many times to wasted effort and time
- **Backtracking**
    - Can be used successfully in small programs
    - The method starts at the location where a symptom has been uncovered
    - The source code is then traced backward (manually) until the location of the cause is found
    - In large programs, the number of potential backward paths may become unmanageably large
- **Cause elimination**
    - **I**nvolves the use of induction or deduction and introduces the concept of binary partitioning
        - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
        - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
    - Data related to the error occurrence are organized to isolate potential causes
    - A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis

- Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

Module V

Maintenance-Overview of maintenance process, types of maintenance. Risk management: software risks - risk identification-risk monitoring and management. Project Management concept: People – Product-Process-Project.

1. Software Maintenance(Ref:-Ian Sommerville 7th edition P.No 514,Chapter 21)

- 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 MUST be maintained therefore if they are to remain useful in an environment.

1.1 Types of Maintenance

- **Maintenance to repair software faults(Corrective Maintenance)**
  - Changing a system to correct deficiencies in the way meets its requirements.
- **Maintenance to adapt software to a different operating environment(Adaptive Maintenance)**
  - 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 –(Perfective Maintenance)**
  - Modifying the system to satisfy new requirements.

Distribution of maintenance effort

1.2 Maintenance Cost

- Usually greater than development costs.
- 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 .
- Development and maintenance contract may be different.
- 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.

1.3 Maintenance Prediction

- Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
  - Change acceptance depends on the maintainability of the components affected by the change;
  - Implementing changes degrades the system and reduces its maintainability;
- Maintenance costs depend on the number of changes and costs of change depend on maintain
-

1.4 Change Prediction

- Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- Tightly coupled systems require changes whenever the environment is changed.
- Factors influencing this relationship are
  - Number and complexity of system interfaces;
  - Number of inherently volatile system requirements;
  - The business processes where the system is used.

Process metrics

- Process measurements may be used to assess maintainability
  - Number of requests for corrective maintenance;
  - Average time required for impact analysis;
  - Average time taken to implement a change request;
  - Number of outstanding change requests.
- If any or all of these is increasing, this may indicate a decline in maintainability.

2. Software Maintenance Process

-
- Identify the changes first, Submit the proposal for changes. Which follows by software evolution process
- Finally the new system will be implemented
- Maintenance process vary considerably depending on the types of software being maintained, the development processes used in an organization and people involved in the process.

Change Request

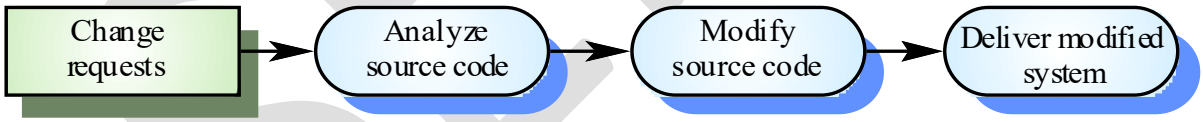
- Change requests are requests for system changes from users, customers or management
- All change requests should be carefully analysed as part of the maintenance process and then implemented
- Some change requests must be implemented urgently
  - Fault repair
  - Changes to the system's environment
  - Urgently required business changes

Change Implementation

Urgent Change Request

- Urgent changes may have to be implemented without going through all stages of the software engineering process
  - serious system fault
  - changes to the system's environment
  - business changes that require a very rapid response (e.g. the release of a competing product).

Emergency Change Repair



Why is maintenance in efficient

- Factors adversely affect maintenance
  - Lack of models or ignorance of available models (73%)
  - Lack of documentation (67.6%)
  - Lack of time to update existing documentation (54.1%)
- Other factors (1994 study)
  - Quality of original application
  - Documentation quality
  - Rotation of maintenance people
- More factors
  - Lack of human resources
  - Different programming styles conflict
  - Lack of documentation and tools
  - Bad maintenance management

- Documentation policy
- Turnover

2. Risk management: software risks - risk identification-risk monitoring and management. Software Engineering and project management, Press Man 6th Edition P-No-726

2.1 Software Risk

- A problem that could cause some loss or threaten the success of the project but which has not happened yet. Two approaches to risk management
  - Reactive Risk Management
  - Pro active risk Management

Reactive Risk Management:

- project team reacts to risks only when they occur
- The team comes into action in an attempt to correct the problem rapidly if something goes wrong. This is often called fire fighting mode
- Resource are found and applied when the risk strikes- fix on failure
- When this fails crisis management takes over and the project is in real jeopardy.

Proactive Risk management

- formal risk analysis is performed
- Potential risks are identified long before the technical work is initiated and their probability and impacts are assessed. And they are ranked by the importance.
- Then they establishes a plan for managing the risks
- organization corrects the root causes of risk
  - TQM concepts and statistical SQA
  - examining risk sources that lie beyond the bounds of the software
  - developing the skill to manage change

Software Risks

- Two characteristics
  - ✓ Uncertainty- the risk may or may not happen. There are no 100% possible risks.
  - ✓ Loss-If the risks becomes a reality unwanted consequences or losses will occur

They are different categories of Risk. They are

- **Project risk-** Threaten project plan.
  - Identify potential budgetary , schedule ,personnel , resource , stake holder, and requirements problem.
- **Technical risk-** The quality and timeline of the software to be implemented.

- Identify potential design ,implementation ,interface ,verification and maintenance problem.

- **Business risk**-threaten the viability of the software to be built.

  Business risks are
  - ✓ **Market risk-** excellent product that no one really wants.
  - ✓ **Strategic risk-** product that not support the existing strategy of company
  - ✓ **Sales risk-** sales team doesn't know how to sell it.
  - ✓ **Management risk**-Losing the support of senior management due to change in focus.
  - ✓ **Budget Risk-** Losing budgetary or personnel commitment
- **Known Risks-** that can be uncovered after the careful evaluation of the project plan, the business and the technical environment in which the project is being developed and other reliable information sources.
- **Predictable Risk-** They are extrapolated from past project experience(rg staff turn over, poor communication with the customer etc.)
- **Unpredictable risks-** They can do occur. But they are extremely difficult to identify in advance.

Seven Principles of Risk Management

1. **Maintain a global perspective**—view software risks within the context of system and the business problem
2. **Take a forward-looking view**—think about the risks that may arise in the future; establish contingency plans
3. **Encourage open communication**—if someone states a potential risk, don't discount it.
4. **Integrate**—a consideration of risk must be integrated into the software process
5. **Emphasize a continuous process**—the team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.
6. **Develop a shared product vision**—if all stakeholders share the same vision of the software, it likely that better risk identification and assessment will occur.
7. **Encourage teamwork**—the talents, skills and knowledge of should be pooled

2.2Risk Identification

- Risk Identification- It is a systematic attempt to specify threats to the project plan. By identifying known and predictable risks the project manager takes a first step toward avoiding them when possible and controlling them necessary.
- One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risk in the following subcategories.

- *Product size*—risks associated with the overall size of the software to be built or modified.
- *Business impact*—risks associated with constraints imposed by management or the marketplace.
- *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.
- *Technology to be built*—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work
- *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.

- The risk item check list can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answer to these questions allow the planner to estimate the impact of the risk

2.2.1 Assessing overall Project Risk

- The following questions have been derived from risk data obtained by surveying experienced software project managers in different parts of the world. Have top software and customer managers formally committed to support the project?
- Are end-users enthusiastically committed to the project and the system/product to be built?
- Are requirements fully understood by the software engineering team and their customers?
- Have customers been involved fully in the definition of requirements?
- Do end-users have realistic expectations?
- Is project scope stable?
- Does the software engineering team have the right mix of skills?
- Are project requirements stable?
- Does the project team have experience with the technology to be implemented?
- Is the number of people on the project team adequate to do the job?
- Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

➢ If any one of these questions is answered negatively mitigation , monitoring , and management steps should be instituted without fail.

➢ The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

2.2.2 Risk Components and Drivers

The project manager has to identify the risk drivers that affect software risk components- performance, cost, support and schedule. The risk components are defined in the following manner

- *Performance risk*—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- *cost risk*—the degree of uncertainty that the project budget will be maintained.
- *support risk*—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- *schedule risk*—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

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

2.3Risk Projection

- *Risk projection*, also called *risk estimation,* attempts to rate each risk in two ways
  - the likelihood or probability that the risk is real
  - the consequences of the problems associated with the risk, should it occur.
- There are four risk projection steps:
  - establish a scale that reflects the perceived likelihood of a risk- negligible,marginal,critical,catastrophic
  - delineate the consequences of the risk
  - estimate the impact of the risk on the project and the product,
  - note the overall accuracy of the risk projection so that there will be no misunderstandings.

2.4 Risk Mitigation Monitoring and Management

- Risk analysis activities are to assist the project team in developing a strategy for dealing with the risk. An effective strategy must consider three issues.
  - mitigation—how can we avoid the risk?
  - monitoring—what factors can we track that will enable us to determine if the risk is becoming more or less likely?
  - management—what contingency plans do we have if the risk becomes a reality?
- if a software team adopts a proactive approach to risk avoidance is always the best strategy. This achieved by developing a plan for risk mitigation.
- Example in case of high staff turn over
  - **Risk mitigation-** To mitigate the risk project management must develop a strategy for reducing turn over. The possible steps to be taken are
    - Meet with current staff to determine cause of turn over.
    - Mitigate those causes that are under control
    - Assume turn over will occur- develop techniques to ensure continuity
    - Organize project team-each development activity is widely dispersed
    - Define documentation standards

- Conduct peer reviews of all work
- Assign a back up staff for every critical technologist**.**

- **As** the project proceeds risk monitoring activities commences.. The project manager monitors factor that may provide an indication of whether the risk is becoming more or less likely. The following factors can be monitored
- **Risk monitoring-**
- Its a project tracking activity with three primary objectives

1. To assess whether predicted risks do in fact occur
2. To ensure the risk aversion steps defined for the risk are being properly applied
3. To collect information that can be used for future risk analysis. Another job of risk monitoring is to attempt to allocate origin caused which problems throughout the project
    - General attitude of team members based on project pressure
    - The degree to which the team has jelled
    - Interpersonal relationship among team members.
    - Potential problems with compensation and benefits
    - The availability of jobs.

- A project manager should also monitor the effectiveness of risk mitigation steps.
- **Risk management-** Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. As part of management the following activities can be performed.
    - Back up Staff
    - Information is documented.
    - Knowledge has been dispersed.
    - Knowledge transfer mode
- Risk mitigation, monitoring and management steps incur additional project cost.
- For large project 30 or 40 risks may be identified. If between 3 and 7 risk management steps are identified for each risk management may become a project in itself. For this 80 percent of the overall project risk can be accounted for by only 20 percent of the identified risk.
- Risk is not limited to software project itself. Risks can occur after the software has been successfully developed and delivered to the customer.

- **2.5 The RMMM Plan-**

- RMMM Plan documents all work performed as part of risk analysis and is used by the manager as part of overall project plan .

3 Concepts of Software Project Management

- Effective software project management focuses on these items (in this order)
- **The people-** Recruitment , Training , Organization and team development

- **The product-**Define scope and objectives of the product
- **The process-** Establish frameworks for software development
- **The project-**Understands the complexities of project development

3.1 The People

- Five categories of stakeholders
  - **Senior managers** – define business issues that often have significant influence on the project
  - **Project (technical) managers** – plan, motivate, organize, and control the practitioners who do the work
  - **Practitioners** – deliver the technical skills that are necessary to engineer a product or application
  - **Customers** – specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome
  - **End users** – interact with the software once it is released for production use

3.1.1 Team Leader

- Among the five stake holders project mangers are important stake holders.
- Qualities of a team leader
  - **Motivation** – the ability to encourage technical people to produce to their best ability
  - **Organizing** – the ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product
  - **Ideas or innovation** – the ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application
- Another set of useful leadership traits
  - **Problem solving** – diagnose, structure a solution, apply lessons learned, remain flexible
  - **Managerial identity** – take charge of the project, have confidence to assume control, have assurance to allow good people to do their jobs
  - **Achievement** – reward initiative, demonstrate that controlled risk taking will not be punished
  - **Influence and team building** – be able to "read" people, understand verbal and nonverbal signals, be able to react to signals, remain under control in high-stress situations

3.1.2 Software Team

- Seven project factors to consider when structuring a software development team
  - The difficulty of the problem to be solved
  - The size of the resultant program(s) in source lines of code
  - The time that the team will stay together

- The degree to which the problem can be modularized
- The required quality and reliability of the system to be built
- The rigidity of the delivery date
- The degree of sociability (communication) required for the project

- Four organizational paradigms for software development teams
  - **Closed paradigm** – traditional hierarchy of authority; works well when producing software similar to past efforts; members are less likely to be innovative
  - **Random paradigm** – depends on individual initiative of team members; works well for projects requiring innovation or technological breakthrough; members may struggle when orderly performance is required
  - **Open paradigm** – hybrid of the closed and random paradigm; works well for solving complex problems; requires collaboration, communication, and consensus among members
  - **Synchronous paradigm** – organizes team members based on the natural pieces of the problem; members have little communication outside of their subgroups
- Five factors that cause team toxity (i.e., a toxic team environment)
  - A frenzied work atmosphere
  - High frustration that causes friction among team members
  - A fragmented or poorly coordinated software process
  - An unclear definition of roles on the software team
  - Continuous and repeated exposure to failure
- How to avoid team toxity
  - Give the team access to all information required to do the job
  - Do not modify major goals and objectives, once they are defined, unless absolutely necessary
  - Give the team as much responsibility for decision making as possible
  - Let the team recommend its own process model
  - Let the team establish its own mechanisms for accountability
  - Establish team-based techniques for feedback and problem solving
- Coordination and Communication issues- Key characteristics of modern software make projects get into trouble
  - scale, uncertainty, interoperability- new system must communicate with existing
- To better ensure success
  - Establish effective methods for coordinating the people who do the work
  - Establish methods of formal and information communication among team members

3.1.3Agile Team

- The small highly motivated project team- agile team.
- Characteristics
  - Self Organizing- no single team structure

- Autonomy to team-take project management and technical decisions to an extent.
- Planning is minimum
- Self approach- process,methods and tools
- Team meeting-daily

3.2 The product

- The scope of the software development must be established and bounded
  - **Context** –
    - How does the software to be built fit into a larger system, product, or business context,
    - what constraints are imposed as a result of the context?
  - **Information objectives** –
    - What customer-visible data objects are produced as output from the software?
    - What data objects are required for input?
  - **Function and performance** –
    - What functions does the software perform to transform input data into output?
    - Are there any special performance characteristics to be addressed?
- Software project scope must be unambiguous and understandable at both the managerial and technical levels
- Problem decomposition- Also referred to as partitioning or problem elaboration
  - Sits at the core of software requirements analysis
- Two major areas of problem decomposition
  - The functionality that must be delivered
  - The process that will be used to deliver it
- Decomposition- Complex problem is partitioned into smaller problems that are more manageable.
  - Product cost and size to be estimated

3.3 The Process

It involves the following 4 activities

- Select Process Model
- Preliminary Project Plan
- Process Decomposition
- Complete Plan

1. Select Process Model

➢ The project manager must decide which process model is most appropriate based on
  - The customers who have requested the product and the people who will do the work

- The characteristics of the product itself
- The project environment in which the software team works

2. Preliminary Project Plan
   - Once a process model is selected, a preliminary project plan is established based on the process framework activities such as Communication, Planning, Modeling, Construction, Deployment

- Project Manger- has to do the following tasks
  - Has to estimate resource requirements for each activities
  - Start date and end date
  - Work products to be produced

3. Process Decomposition
   - How to accomplish the framework activity?
   - For each framework activity we have to frame out a set of tasks

Example Communication

- Develop list of clarification issue
- Meet with customer to address clarification issues.
- Jointly develop a statement of scope
- Review the statement with all concerned
- Modify the statement of scope as required

3.4 The Project

The following activities to be performed.

- Start on the right foot
  - Understand the problem; set realistic objectives and expectations; form a good team
- Maintain momentum
  - Provide incentives to reduce turnover of people;
  - emphasize quality in every task;
  - have senior management stay out of the team's way
- Track progress
  - Track the completion of work products;
  - collect software process and project measures;
  - assess progress against expected averages
- Make smart decisions
  - Keep it simple;
  - use COTS or existing software before writing new code;
  - follow standard approaches;
  - identify and avoid risks;
  - always allocate more time than you think you need to do complex or risky tasks
- Conduct a post mortem analysis
  - Track lessons learned for each project;
  - compare planned and actual schedules;
  - collect and analyze software project metrics;
  - get feedback from teams members and customers;

- record findings in written form

Signs that project is in Jeopardy

- Software people don't understand their customer's needs
- The product scope is poorly defined
- Changes are managed poorly
- The chosen technology changes
- Business needs change (or are poorly defined)
- Deadlines are unrealistic
- Users are resistant
- Sponsorship is lost (or was never properly obtained)
- The project team lacks people with appropriate skills
- Managers (and practitioners) avoid best practices and lessons learned

The Project W5H Principle

- A series of questions that lead to a definition of key project characteristics and the resultant project plan

- **Why** is the system being developed?
  - Assesses the validity of business reasons and justifications
- **What** will be done?
  - Establishes the task set required for the project
- **When** will it be done?
  - Establishes a project schedule
- **Who** is responsible for a function?
  - Defines the role and responsibility of each team member
- **Where** are they organizationally located?
  - Notes the organizational location of team members, customers, and other stakeholders
- **How** will the job be done technically and managerially?
  - Establishes the management and technical strategy for the project
- **How** much of each resource is needed?
  - Establishes estimates based on the answers to the previous questions

Module VI

Project scheduling and tracking: Basic concepts , relation between people and effort-defining task set for the software project-selecting software engineering task -Software configuration management: Basics and standards User interface design – rules. Computer aided software engineering tools – CASE building blocks, taxonomy of CASE tools, integrated CASE environment.

6.1 Project Scheduling : Basic Concepts

- Basic concept behind the project scheduling is to deliver the software product on time. The basic reasons for the late delivery of software is
    - An unrealistic deadline established by someone outside the software engineering group
    - Changing customer requirements that are not reflected in schedule changes
    - An honest underestimate of the amount of effort and /or the number of resources
    - Predictable and/or unpredictable risks that were not considered ,
    - Technical difficulties that could not have been foreseen in advance
    - Human difficulties that could not have been foreseen in advance
    - Miscommunication among project staff
    - A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.
- Aggressive deadlines are a fact of life in the software business.
- In case of an unrealistic deadline we can perform the following actions to overcome the crisis
    - Perform a detailed estimate using historical data from past projects
    - Meet with the customer and (using the detailed estimate) explain why the imposed deadline is unrealistic functionality until later
    - Using an incremental process model develop a software engineering strategy that will deliver critical functionality by the imposed deadline. But delay the other fu
    - Offer the incremental development strategy as an alternative and offer some options.

6.2 Project scheduling

- The reality of a technical project is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about impact on project completion date. Other task lie on the critical path. If these critical task fall behind schedule the completion of entire project is put into jeopardy.
- As a project manager we have to define all project tasks, build a network that depicts their interdependencies, identify the tasks that are critical within the network, and then track their progress.
- **Software Project Scheduling** is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. And the schedule should be evolved over time.

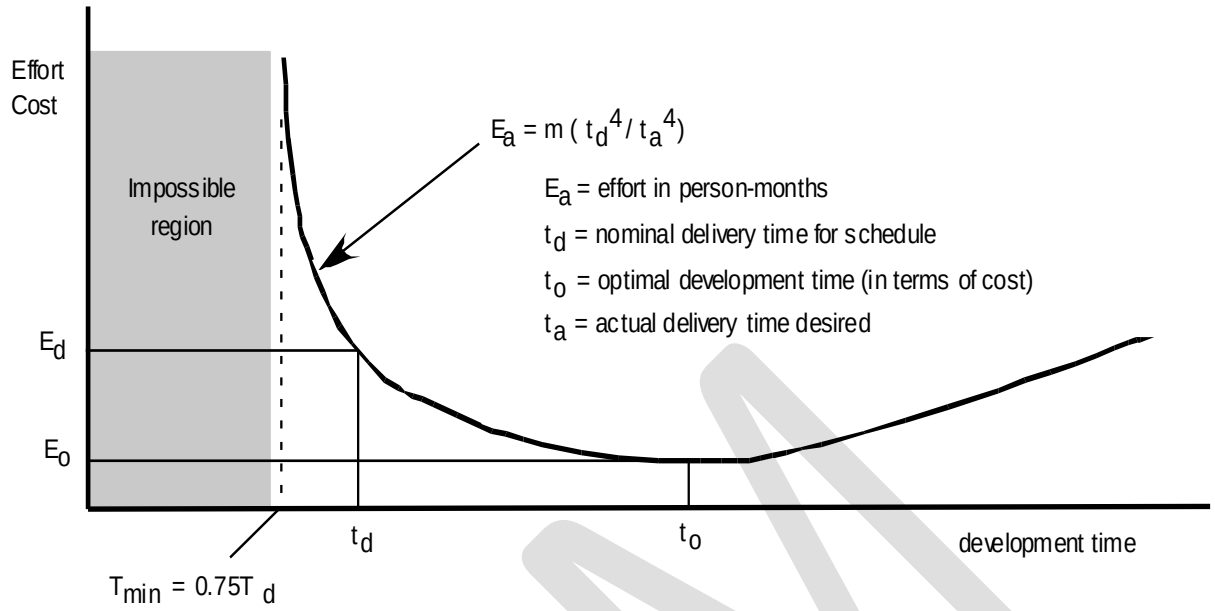
- During Early stages of planning a macroscopic schedule is developed. This types of schedule identifies all major process frame work activities and the product function to which they are applied. As the project gets under way each entry on the macroscopic schedule is refined to detailed schedule.
- Scheduling can be viewed in two different perspectives.
    - In the first, an end date for release of a computer based system has already been established. The software organization is constrained to distribute effort within the prescribed time frame.
    - The second view of software scheduling assumes that rough chronological bounds has been discussed but that the end date is determined by the software organization. Effort is distributed to make best use of resources and an end date is defined after careful analysis of the software.

6.2.1 Basic Principles

- **Compartmentalization**-Process of defining distinct tasks
- **Interdependency**-indicate task inter relationship
- **Time allocation-** Each task to be scheduled must be allocated some number of work units
- **Effort validation-**Every project has a defined number of people on the team
    - no more than the allocated number of people have been scheduled at any given time
- **Defined responsibilities**-Every task that is scheduled should be assigned to a specific team member
- **Defined outcomes-** Every task that is scheduled should have a defined outcome
- **Defined milestones-** every tasks or group of tasks should be associated with a project mile stone. A milestone is accomplished when one or more products has been reviewed for quality and has been approved.

6. 3 The relationship between people and effort

- In a small software development project a single person can analyse the requirements, perform design, generate code, and conduct test. As the size of the project increases more people must become involved.
- Common management myth: *If we fall behind schedule, we can always add more programmers and catch up later in the project*
- This practice actually has a disruptive effect and causes the schedule to slip even further
- The added people must learn the system
- The people who teach them are the same people who were earlier doing the work. During teaching, no work is being accomplished. Lines of communication (and the inherent delays) increase for each new person added
- **The Putnem- Norden-Rayleigh Curve** provides an indication of the relationship between effort applied and delivery time for a software project.



- Implies that delaying a project delivery can reduce costs significantly.

$$E=L^3/(^{P3}t^4)$$

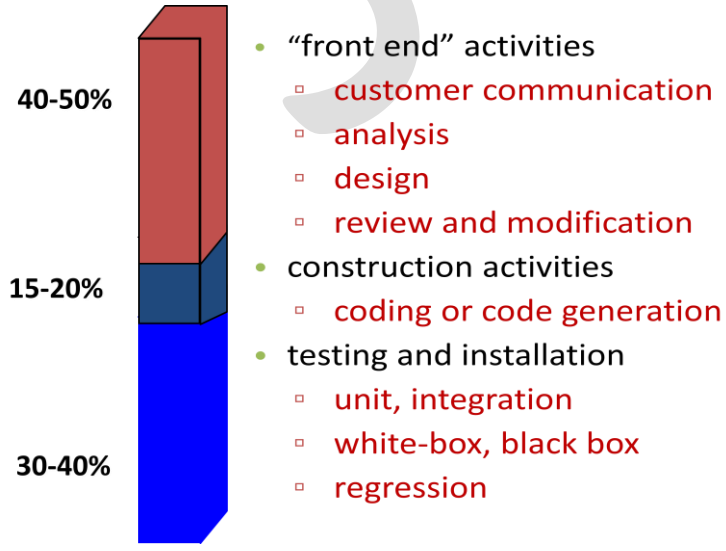
E- effort

- P-productivity parameter(2000-12000)
- L-Lines of code
- t-development time

6.3.1 Effort Distribution

• The effort distribution for various phases of software development is as follows

- Project Planning-2 to 3%
- Requirements Analysis-10 to 25%
- Software design-20 to 25%
- Code-15-20%
- Testing- 30-40%



6.4 Defining task set for the software project

- A task set is a collection of software engineering work tasks , milestones and work products that must be accomplished to achieve high software quality.
- Regardless of the process model that is chosen the work that a software team performs is achieved through a set of tasks that enable us to define develop and ultimately support computer software. No single task set is appropriate for all projects.
- A task set is the work breakdown structure for the project
- It varies depending on the project type and the degree of rigor. Degree of rigor is function of many project characteristics. 4 types of degree of rigor.

> **Casual.** All process framework activities (Chapter 2) are applied, but only a minimum task set is required. In general, umbrella tasks will be minimized and documentation requirements will be reduced. All basic principles of software engineering are still applicable.
>
> **Structured.** The process framework will be applied for this project. Framework activities and related tasks appropriate to the project type will be applied and umbrella activities necessary to ensure high quality will be applied. SQA, SCM, documentation, and measurement tasks will be conducted in a streamlined manner.
>
> **Strict.** The full process will be applied for this project with a degree of discipline that will ensure high quality. All umbrella activities will be applied and robust work products will be produced.
>
> **Quick reaction.** The process framework will be applied for this project, but because of an emergency situation[8] only those tasks essential to maintaining good quality will be applied. "Back-filling" (i.e., developing a complete set of documentation, conducting additional reviews) will be accomplished after the application/product is delivered to the customer.
>
> The project manager must develop a systematic approach for selecting the degree of rigor that is appropriate for a particular project. To accomplish this, project adaptation criteria are defined and a task set selector value is computed.

- The task set should provide enough discipline to achieve high software quality. But it must not burden the project team with unnecessary work
- The different **types of projects** are
    - **Concept development projects**- Explore some new business concept or application of some new technology
    - New application development- That are undertaken as a consequences of a specific customer request
    - Application enhancement- When existing software undergo major modifications to functions. Performance
    - Application maintenance –correct , adapt or extend existing software
    - Reengineering projects- That are undertaken with the intent of rebuilding an existing system in a whole.
    -

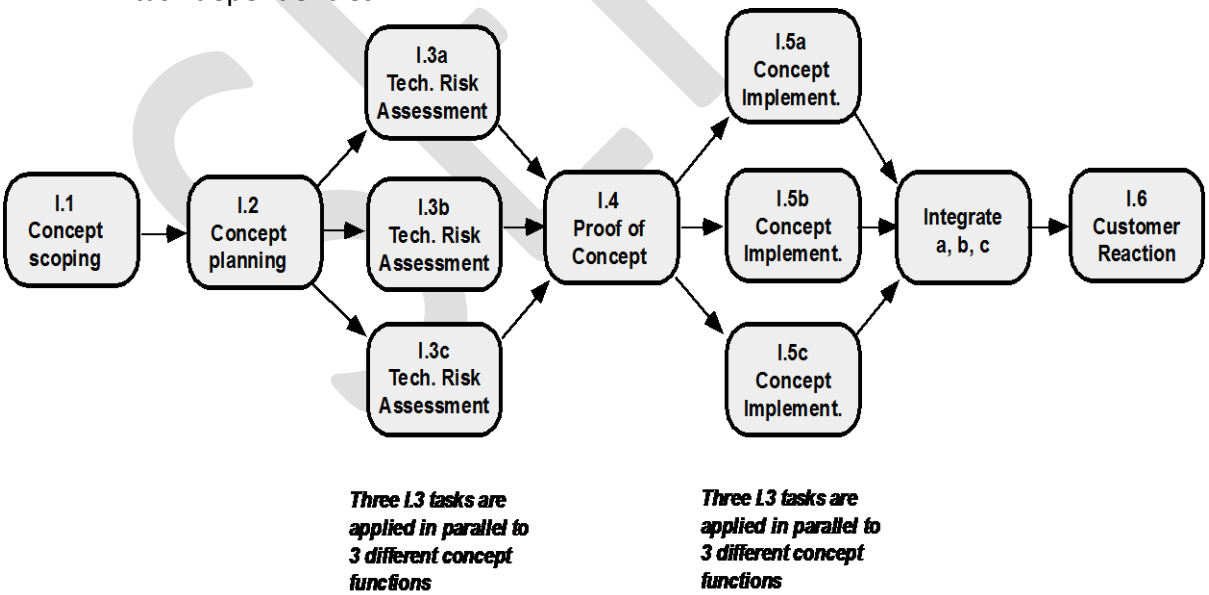
6.5 A Task Set Example- Selecting Software Engineering Task

Concept development projects are initiated when the potential for some new technology must be explored. Concept development projects are approached by applying the following actions.

- Concept Scoping- Determines the overall scope of the project.
- Preliminary Concept planning- establishes the organizations ability to undertake the work implied by the project scope.
- Technology Risk assessment- Evaluates the risk associated with the technology to be implemented as part of the project scope.
- Proof of concept- Demonstrates the viability of new technology in the software context.
- Concept implementation- implements the concept representation in a manner that can be reviewed by a customer and is used for marketing purpose when a concept must be sold to other customers or management.
- Customer reaction- to the concept solicits feedback on a new technology concept and targets specific customer application.

Defining a Task Network

- Task network is also called an activity network
- It is a graphic representation of the task flow for a project
- It depicts task length, sequence, concurrency, and dependency and points out inter-task dependencies



Refinement of Software Engineering Actions

- The software engineering actions described in the preceding section may be used to define a macroscopic schedule for a project. However the macroscopic schedule must be refined to create a detailed project schedule. Refinement begins by taking each action and decomposing it into a set of tasks .

As an example of task decomposition, consider Action 1.1, Concept Scoping. Task refinement can be accomplished using an outline format, but in this book, a process design language approach is used to illustrate the flow of the concept scoping action:

Task definition: Action 1.1 Concept Scoping
1.1.1 Identify need, benefits and potential customers;
1.1.2 Define desired output/control and input events that drive the application;
Begin Task 1.1.2
1.1.2.1 TR: Review written description of need[7]
1.1.2.2 Derive a list of customer visible outputs/inputs
1.1.2.3 TR: Review outputs/inputs with customer and revise as required; endtask Task 1.1.2
1.1.3 Define the functionality/behavior for each major function;
Begin Task 1.1.3
1.1.3.1 TR: Review output and input data objects derived in task 1.1.2;
1.1.3.2 Derive a model of functions/behaviors;
1.1.3.3 TR: Review functions/behaviors with customer and revise as required;
endtask Task 1.1.3
1.1.4 Isolate those elements of the technology to be implemented in software;
1.1.5 Research availability of existing software;
1.1.6 Define technical feasibility;
1.1.7 Make quick estimate of size;
1.1.8 Create a scope definition;
endtask definition: Action 1.1

6.6 Scheduling

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects.

Program evaluation and review technique (PERT) and the *critical path method* (CPM) are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities: estimates of effort, a decomposition of the product function, the selection of the appropriate process model and task set, and decomposition of the tasks that are selected.

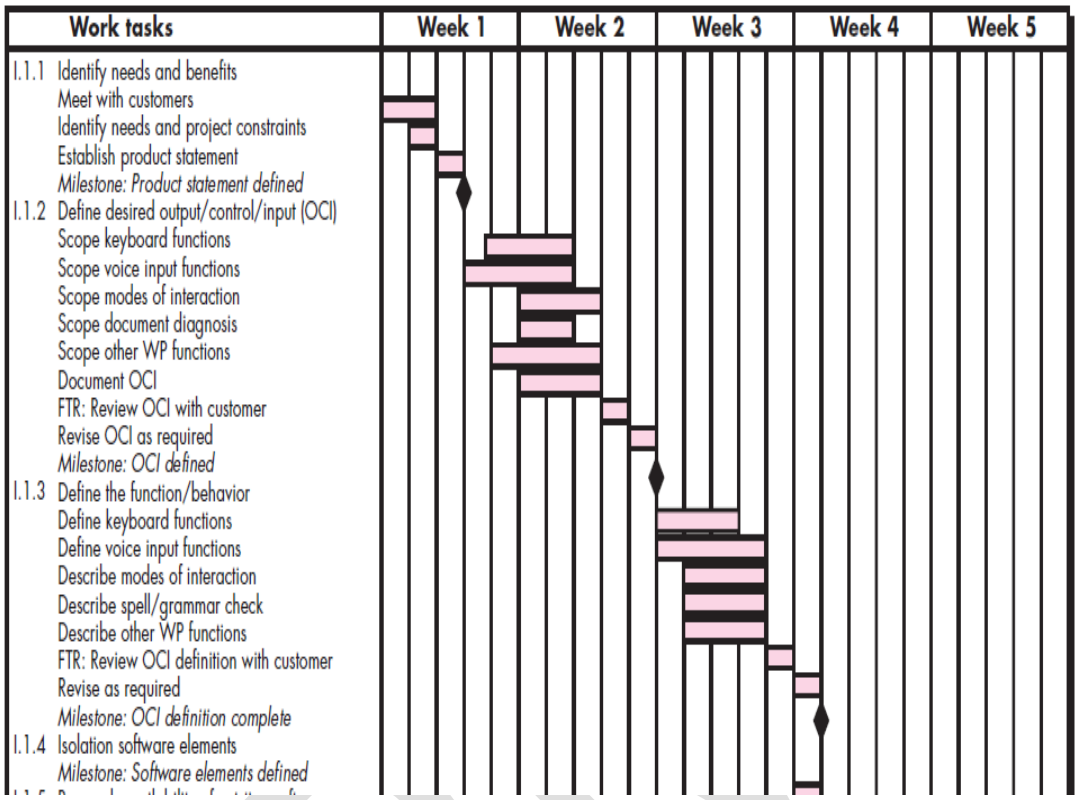
Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow you to (1) determine the critical path—the chain of tasks that determines the duration of the project, (2) establish "most likely" time estimates for individual tasks by applying statistical models, and (3) calculate "boundary times" that define a time "window" for a particular task.

The automated tools are

6.6.2. Timeline Chart/Gant Chart

- All project tasks are listed in the far left column
- The next few columns may list the start and finish dates.
- To the far right are columns representing dates on a calendar
- The length of a horizontal bar on the calendar indicates the duration of the task
- When multiple bars occur at the same time interval on the calendar, this implies task concurrency
- A diamond in the calendar area of a specific task indicates that the task is a milestone; a milestone has a time duration of zero.
-
-



Time line Chart Example

6.6.2 Tracking The Schedule

- Conduct periodic project status meetings
- Evaluate the results
- Determine whether formal project milestones (i.e., diamonds) have been accomplished by the scheduled date
- Compare actual start date to planned start date for each project task listed in the timeline chart
  - Meet informally with the software engineering team to obtain their subjective assessment of progress to date and problems on the horizon
- Quantitative approach
  - Use earned value analysis to assess progress quantitatively

6.6.2.1 Earned Value Analysis

- Its a quantitative method to track the schedule.
- Earned value analysis is a measure of progress by assessing the percent of completeness for a project.
- It gives accurate and reliable readings of performance very early into a project.
- It provides a common value scale (i.e., time) for every project task, regardless of the type of work being performed
- It includes the following activities

1. Compute the budgeted cost of work scheduled (BCWS) for each work task *i* in the schedule
    - The BCWS is the effort planned; work is estimated in person-hours or person-days for each task
    - To determine progress at a given point along the project schedule, the value of BCWS is the sum of the $BCWS_i$ values of all the work tasks that should have been completed by that point of time in the project schedule
2. Sum up the BCWS values for all work tasks to derive the budget at completion (BAC)
3. Compute the value for the budgeted cost of work performed (BCWP)
    - BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point of time on the project schedule

- Other project Indicated with earned value analysis are
    - SPI = BCWP/BCWS
        - Schedule performance index (SPI) is an indication of the efficiency with which the project is utilizing scheduled resources
        - SPI close to 1.0 indicates efficient execution of the project schedule
    - SV = BCWP – BCWS
        - Schedule variance (SV) is an absolute indication of variance from the planned schedule
    - PSFC = BCWS/BAC
        - Percent scheduled for completion (PSFC) provides an indication of the percentage of work that should have been completed by time *t*
    - PC = BCWP/BAC
        - Percent complete (PC) provides a quantitative indication of the percent of work that has been completed at a given point in time *t*
    - ACWP = sum of BCWP as of time t
        - Actual cost of work performed (ASWP) includes all tasks that have been completed by a point in time *t* on the project schedule
    - CPI = BCWP/ACWP
        - A cost performance index (CPI) close to 1.0 provides a strong indication that the project is within its defined budget

- CV = BCWP – ACWP
    - The cost variance is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project

6.6.2.2 Error Tracking

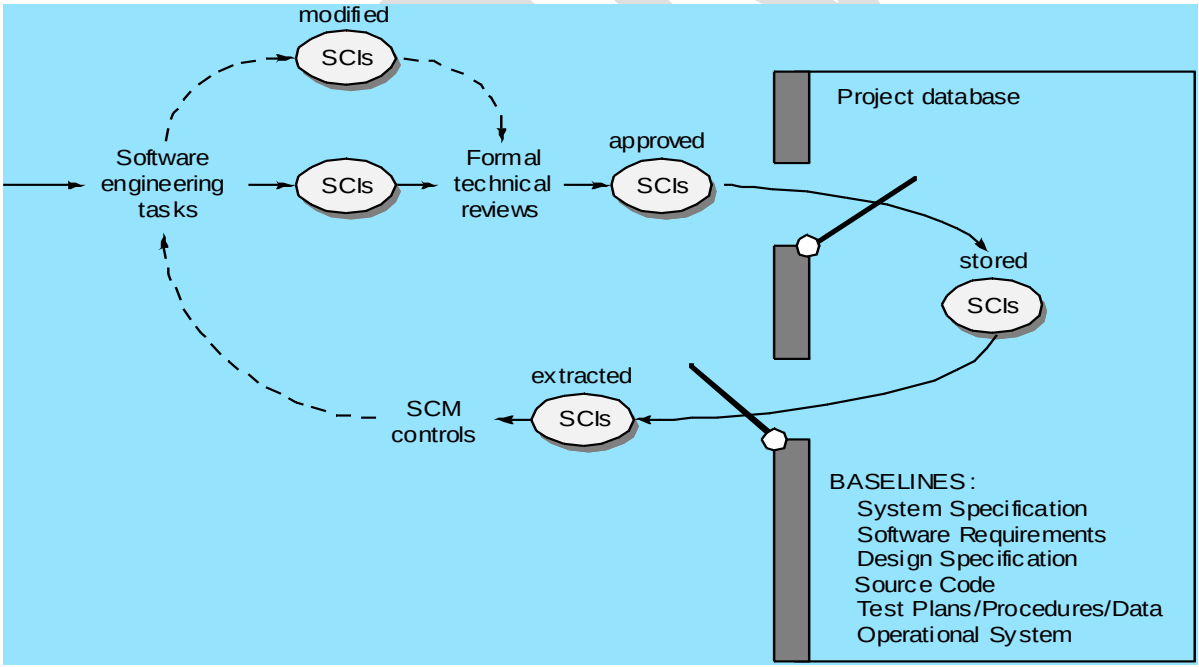
- ET is a process of assessing the status of the s/w project
- Defect Removal Efficiency (DRE) = E/(E+D)
- Where E is error and D is defect.
- Defects is Any error that remain uncovered and are found in later tasks are called defects.

6.7 Software Configuration Management

- SCM is an umbrella activity that is applied throughout the software process. Software Configuration Management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process.
- The SCM defines a series of tasks that have four primary objectives.
    - To identify all items that collectively define the software configuration
    - To manage changes to one or more of these items
    - To ensure that change is being properly implemented.
    - Report changes to others who may have an interest.
- There are mainly three outputs for a software. They are
    - Computer programs
    - Document
    - Data
- Software configuration- The items that comprise all information product as part of the software process. The changes can be occurred at any time. There are 4 fundamental sources of changes. They are

- New business or market conditions dictate changes in product requirements or business rules.
- New stakeholder needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

- **6.7.1 Baselines**
  - The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:
  - A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.



The process of modification in baseline.

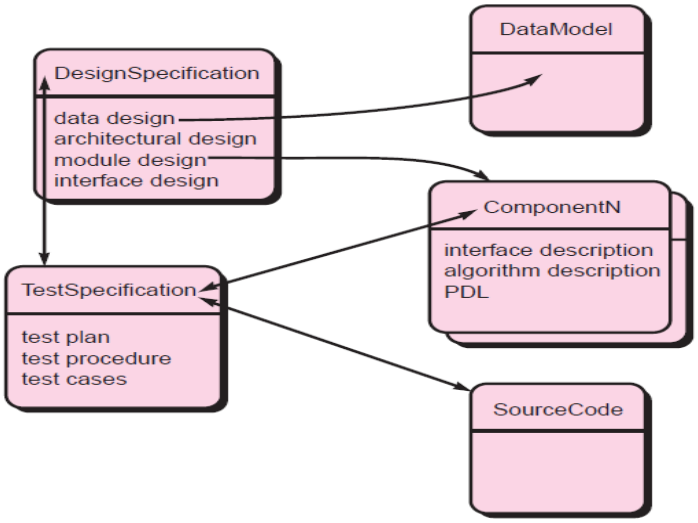
- If any one wants to access the SCI that are stored in a project database, a copy of item will be given to the user.
- After the modifications in the local copy the modified SCI have to be approved by software configuration management team after formal technical review. Then only the changes will be updated in the database.

6.7.2 The SCM Elements

- *Component elements*—a set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- *Process elements*—a collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering and use of computer software.
- *Construction elements*—a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- *Human elements*—to implement effective SCM, the software team uses a set of tools and process features

6.7.3Software Configuration Items

In reality, SCIs are organized to form configuration objects that may be cataloged in the project database with a single name. A *configuration object* has a name, attributes, and is "connected" to other objects by relationships. Referring to Figure 22.2, the configuration objects, **DesignSpecification, DataModel, ComponentN, SourceCode,** and **TestSpecification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is, **DataModel** and **ComponentN** are part of the object **DesignSpecification.** A double-headed straight arrow indicates an interrelationship. If a change were made to the **SourceCode** object, the interrelationships enable you to determine what other objects (and SCIs) might be affected.[2]



Configuration Objects

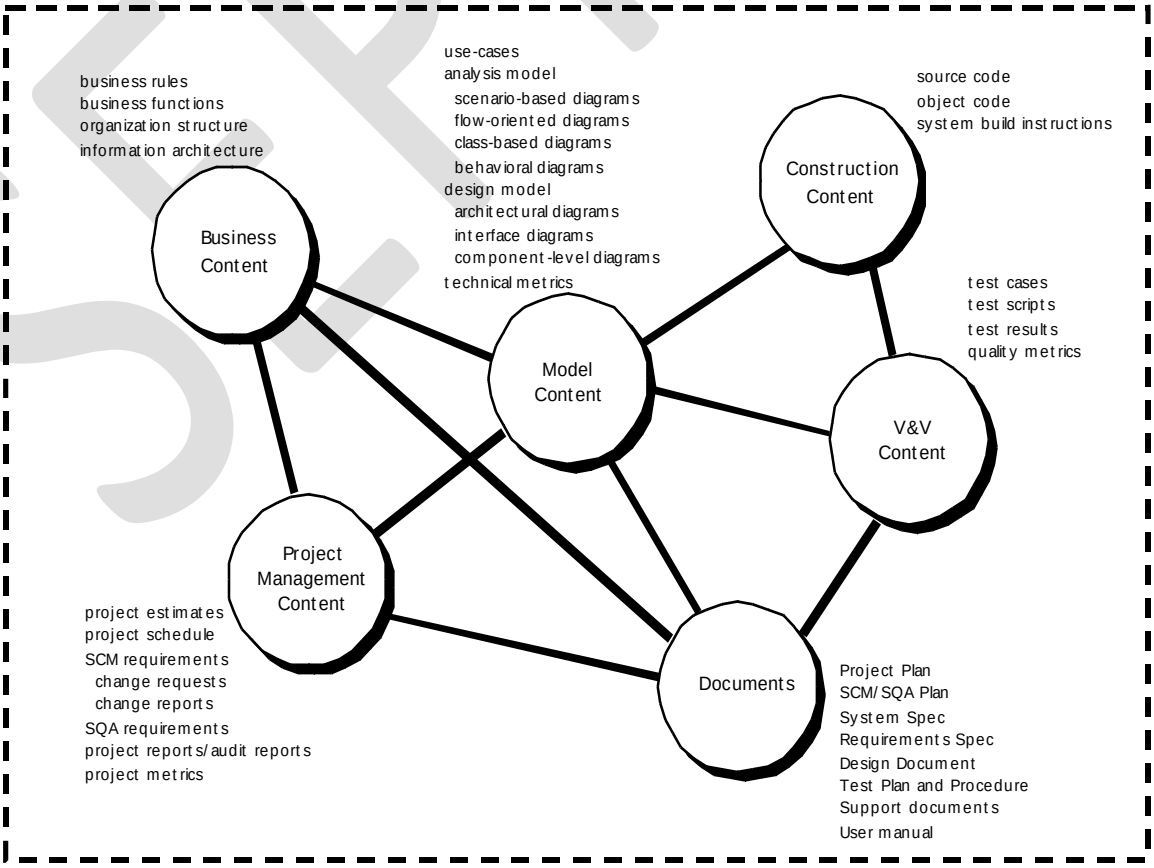
6.7.4The SCM Repository

The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner. It provides the obvious functions of a modern database management system by ensuring data integrity, sharing, and integration. In addition, the SCM repository provides a hub for the integration of software tools, is central to the flow of the software process, and can enforce uniform structure and format for software engineering work products.

To achieve these capabilities, the repository is defined in terms of a meta-model. The *meta-model* determines how information is stored in the repository, how data can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs.

The repository performs or precipitates the following functions [For89]:

- Data integrity
- Information sharing
- Tool integration
- Data integration
- Methodology enforcement
- Document standardization



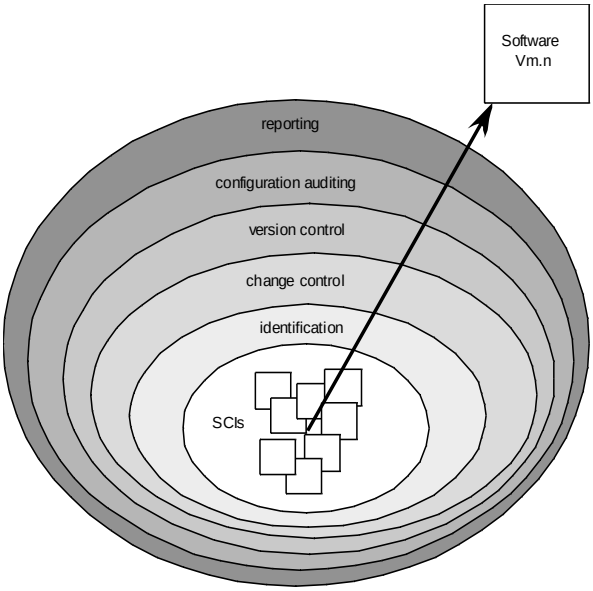
Repository Contents

6.7.5 SCM Features

- **Versioning.**
  - saves all of these versions to enable effective management of product releases and to permit developers to go back to previous versions
- **Dependency tracking and change management.**
  - The repository manages a wide variety of relationships among the data elements stored in it.
- **Requirements tracing.**
  - Provides the ability to track all the design and construction components and deliverables that result from a specific requirement specification
- **Configuration management.**
  - Keeps track of a series of configurations representing specific project milestones or production releases. Version management provides the needed versions, and link management keeps track of interdependencies.
- **Audit trails.**
  - establishes additional information about when, why, and by whom changes are made

6.8The SCM Process

- In order to identify the SCM process address the following questions
  - How does a software team identify the discrete elements of a software configuration?
  - How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
  - How does an organization control changes before and after software is released to a customer?
  - Who has responsibility for approving and ranking changes?
  - How can we ensure that changes have been made properly?
  - What mechanism is used to appraise others of changes that are made?
- The five SCM tasks are identification, version control, change control, configuration auditing and reporting.



SCM Process

1. Identify Objects

- Software configuration objects are stored in a repository.
- Each Object has
  - Name
  - Description- SCI type , project identifier , version control
  - List of resources- data type ,variable , functions etc

The identification scheme for software objects must recognize that objects evolve throughout the software process. Before an object is baselined, it may change many times, and even after a baseline has been established, changes may be quite frequent.

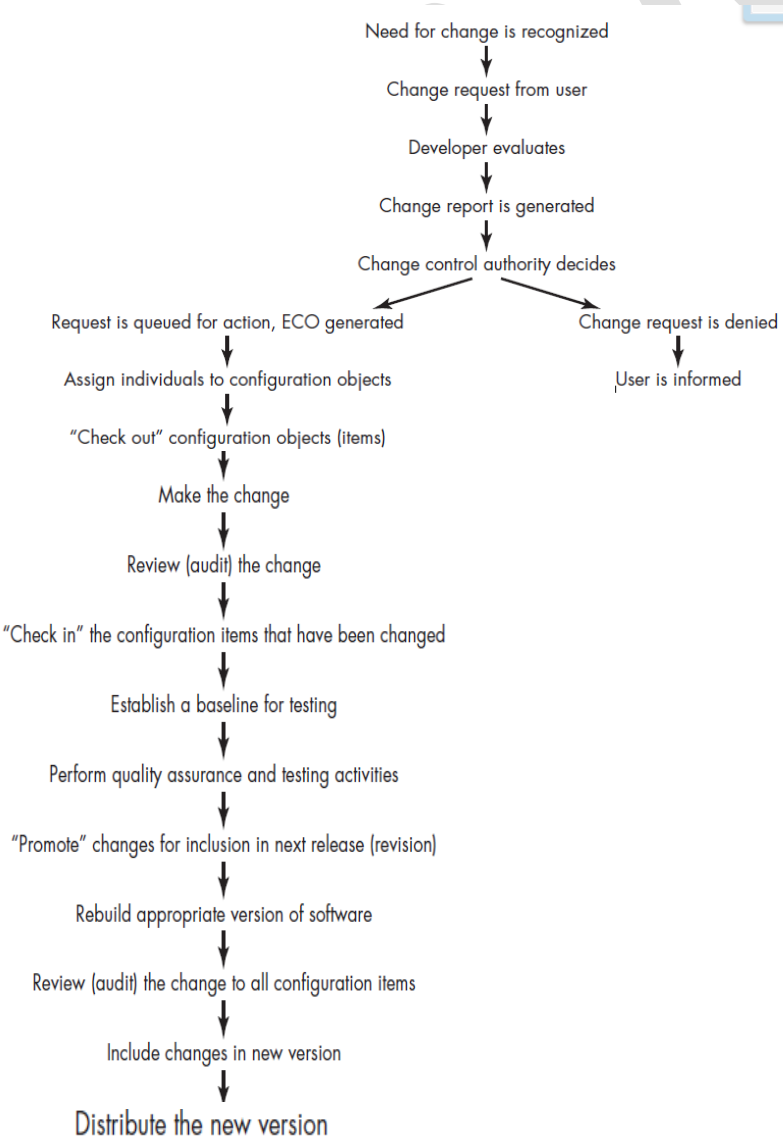
2. Version Control

- Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process
- A version control system implements or is directly integrated with four major capabilities:
  - a *project database (repository)* that stores all relevant configuration objects
  - a *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions);
- a *make facility* that enables the software engineer to collect all relevant configuration objects and construct a specific version of the software.
- an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

3. Change Control

For a large software project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change. The change control process is illustrated schematically in Figure 22.5. A *change request* is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a *change report,* which is used by a *change control authority* (CCA)—a person or group that makes a final decision on the status and priority of the change. An *engineering change order* (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit.

The object(s) to be changed can be placed in a directory that is controlled solely by the software engineer making the change. A version control system (see the CVS sidebar) updates the original file once the change has been made. As an alternative,



Change Control Process

4. Configuration Audit

Identification, version control, and change control help you to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can a software team ensure that the change has been properly implemented? The answer is twofold: (1) technical reviews and (2) the software configuration audit.

The technical review (Chapter 15) focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side effects. A technical review should be conducted for all but the most trivial changes.

A *software configuration audit* complements the technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a technical review been conducted to assess technical correctness?
3. Has the software process been followed and have software engineering standards been properly applied?
4. Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?
5. Have SCM procedures for noting the change, recording it, and reporting it been followed?
6. Have all related SCIs been properly updated?

5. Status Reporting

Configuration status reporting (sometimes called *status accounting*) is an SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in Figure 22.5. Each time an SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made. Each time a configuration audit is conducted, the results are reported as part of the CSR task. Output from CSR may be placed in an online database or website, so that software developers or support staff can access change information by keyword category. In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners appraised of important changes.

6.9User Interface Design

- It creates an effective communication medium between a human and a computer.
- Interface design should be
  - Easy to use
  - Easy to learn
  - Easy to understand
- Typical Design errors in an user interface are
  - lack of consistency
  - too much memorization
  - no guidance / help
  - no context sensitivity
  - poor response
  - unfriendly

6.9.1 Golden Rule

- The three golden rule for user interface design is
  - Place the user in control
  - Reduce the user's memory load
  - Make the interface consistent

1. Place the user in control

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if *spell check* is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multitouch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.

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

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

2. Reduce the users Memory Load

The more the user has to remember the more error prone the interaction with the system will be. It is for this reason that a well defined user interface does not tax the users memory.

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

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

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

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick. An example, common to many word-processing applications, is the underlining function. The function itself is one of a number of functions under a *text style* menu. However, every underlining capability is not listed. The user must pick underlining; then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

3. Make the interface Consistent

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

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

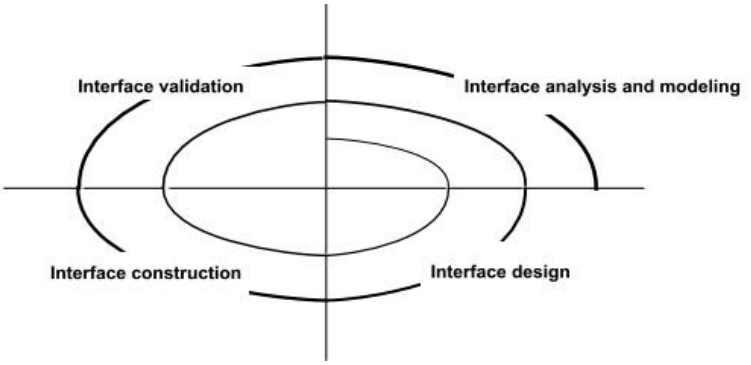
Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

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

6.10 User Interface Design Process

It includes 4 activities. They are

- Interface Analysis
- Interface Design
- Interface Implementation
- Interface Validation



1. **Interface Analysis**

- Interface analysis means understanding

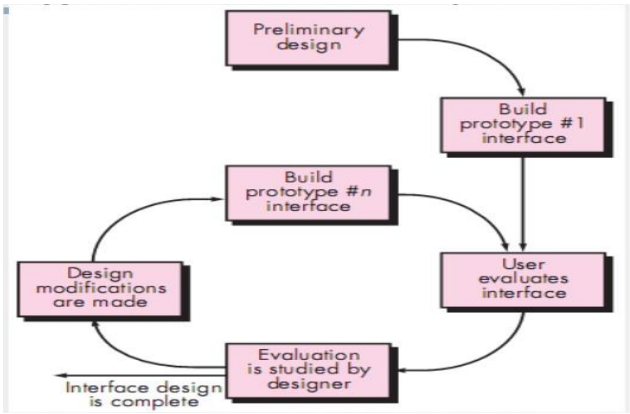
  (1) the people (end-users) who will interact with the system through the interface;

  (2) the tasks that end-users must perform to do their work,

  (3) the content that is presented as part of the interface

  (4) the environment in which these tasks will be conducted.

- **Interface Design**
  - Using information developed during interface analysis, define interface objects and actions (operations).
  - Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
  - Depict each interface state as it will actually look to the end-user.
  - Indicate how the user interprets the state of the system from information provided through the interface
- **Interface Implementation**
  - Begins with the creation of a prototype
  - A user interface tool kit may be used to complete the construction of the interface
  - Managing input device
  - Providing help and feedback
- **Interface Validation**
  - The ability of the interface to implement every user task correctly to achieve all general user requirements.
  - The degree to which the interface is easy to use and easy to learn.
  - The users acceptance of the interface as a useful tool in their work



The evaluation Process

Computer aided software engineering tools – CASE building blocks, taxonomy of CASE tools, integrated CASE environment.

6.10Computer Aided Software Engineering Tools

The workshop for software engineering has been called an *integrated project support environment* (discussed later in this chapter) and the tools that fill the workshop are collectively called *computer-aided software engineering*.

CASE provides the software engineer with the ability to automate manual activities and to improve engineering insight. Like computer-aided engineering and design tools that are used by engineers in other disciplines, CASE tools help to ensure that quality is designed in before the product is built.

6.10.1 CASE Building Blocks

CASE Building Blocks are

- CASE tools
- Integration framework(specialized programs allowing CASE tools to communicate with one another)
- Portability services (allow CASE tools and their integration framework to migrate across different operating systems and hardware platforms without significant adaptive maintenance)
- Operating system (database and object management services)
- Hardware platform
- Environmental architecture (hardware and system support)

The environment architecture, composed of the hardware platform and system support (including networking software, database management, and object management services), lays the ground work for CASE. But the CASE environment itself

demands other building blocks. A set of *portability services* provides a bridge between CASE tools and their integration framework and the environment architecture. The *integration framework* is a collection of specialized programs that enables individual CASE tools to communicate with one another, to create a project database, and to exhibit the same look and feel to the end-user (the software engineer). Portability services allow CASE tools and their integration framework to migrate across different hardware platforms and operating systems without significant adaptive maintenance.

6.10.2 CASE Taxonomy

CASE tools can be classified by function, by their role as instruments for managers or technical people, by their use in the various steps of the software engineering process, by the environment architecture (hardware and software) that supports them, or even by their origin or cost [QED89]. The taxonomy presented here uses function as a primary criterion.

* Business process engineering tools - represent business data objects, their relationships, and flow of the data objects between company business areas
* Process modeling and management tools - represent key elements of processes and provide links to other tools that provide support to defined process activities
* Project planning tools - used for cost and effort estimation, and project scheduling
* Risk analysis tools - help project managers build risk tables by providing detailed guidance in the identification and analysis of risks
* Requirements tracing tools - provide systematic database-like approach to tracking requirement status beginning with specification
* Metrics and management tools -management oriented tools capture project specific metrics that provide an overall indication of productivity or quality, technically oriented metrics determine metrics that provide greater insight into the quality of design or code
* Documentation tools - provide opportunities for improved productivity by reducing the amount of time needed to produce work products
* System software tools - network system software, object management services, distributed component support, and communications software
* Quality assurance tools -metrics tools that audit source code to determine compliance with language standards or tools that extract metrics to project the quality of software being built
* Database management tools - RDMS and OODMS serve as the foundation for the establishment of the CASE repository
* Software configuration management tools -uses the CASE repository to assist

with all SCM tasks (identification, version control, change control, auditing, status accounting)

* Analysis and design tools -enable the software engineer to create analysis and design models of the system to be built, perform consistency checking between models
* PRO/SIM tools - prototyping and simulation tools provide software engineers with ability to predict the behavior of real-time systems before they are built and the creation of interface mockups for customer review
* Interface design and development tools - toolkits of interface components, often part environment with a GUI to allow rapid prototyping of user interface designs
* Prototyping tools - enable rapid definition of screen layouts, data design, and report generation
* Programming tools - compilers, editors, debuggers, OO programming environments, fourth generation languages, graphical programming environments, applications generators, and database query generators
* Web development tools - assist with the generation of web page text, graphics, forms, scripts, applets, etc.
* Integration and testing tools
* Data acquisition (get data for testing)
* static measurement (analyze source code without using test cases)
* dynamic measurement (analyze source code during execution)
* simulation (simulate function of hardware and other externals)
* test management (assist in test planning, development, and control)
* cross-functional (tools that cross test tool category boundaries)
* Static analysis tools - code-based testing tools, specialized testing languages, requirements-based testing tools
* Dynamic analysis tools -intrusive tools modify source code by inserting probes to check path coverage, assertions, or execution flow, non-intrusive tools use a separate hardware processor running in parallel with processor containing the program being tested
* Test management tools - coordinate regression testing, compare actual and expected output, conduct batch testing, and serve as generic test drivers
* Client/server testing tools - exercise the GUI and network communications requirements for the client and server
* **Reengineering tools**
* reverse engineering to specification tools - generate analysis and design models from source code, where used lists, and other design information
* code restructuring and analysis tools -analyze program syntax, generate control flow graph, and automatically generates a structured program
* on-line system reengineering tools - used to modify on-line DBMS

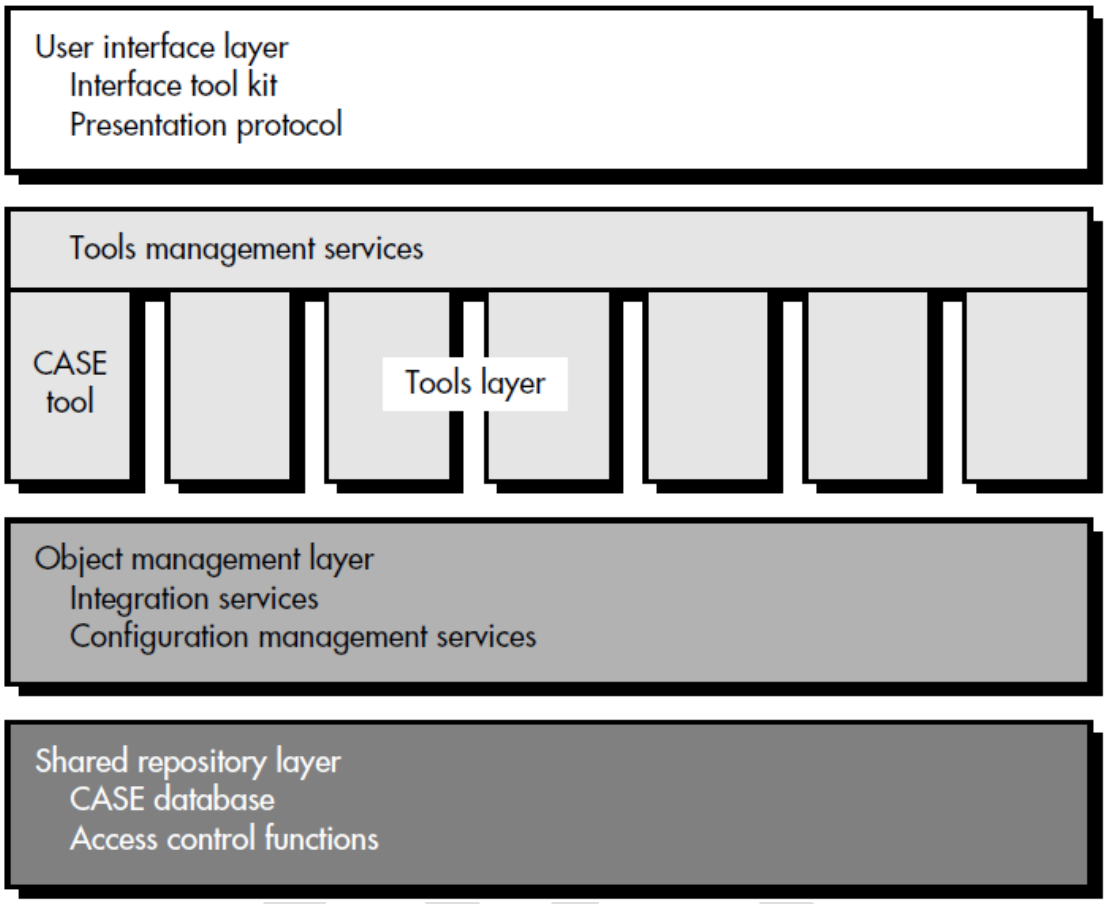
6.10.3 Integrated CASE Environment

- Integration of a variety of tools and information that enables *closure* of communication among tools, between people and across the software process
- Combination of CASE tools in an environment where interface mechanisms are standardized

An integrated CASE environment should

- Provide mechanism for sharing information among all tools contained in the environment
- Enable changes to items to be tracked to other information items
- Provide version control and overall configuration management
- Allow direct access to any tool contained in the environment
- Establish automated support for the chosen software process model, integrating CASE tools and SCI's into a standard work break down structure
- Enable users of each tool to experience a consistent look and feel at the human-computer interface
- Support communication among software engineers
- Collect both management and technical metrics to improve the process and the product

6.10.4 Integration Architecture



* User interface layer
* interface toolkit - contains software for UI management and library of display objects
* common presentation protocol - guidelines that give all CASE tools the same look and feel (icons, mouse behavior, menu names, object names)
* Tools layer
* tools management services - control behavior of tools inside environment
* CASE tools themselves

> The *tools layer* incorporates a set of tools management services with the CASE tools themselves. *Tools management services* (TMS) control the behavior of tools within the environment. If multitasking is used during the execution of one or more tools, TMS performs multitask synchronization and communication, coordinates the flow of information from the repository and object management system into the tools, accomplishes security and auditing functions, and collects metrics on tool usage.

* Object management layer (OML) - performs the configuration management function, working with the CASE repository OML provides integration services

The *object management layer* (OML) performs the configuration management functions described in Chapter 9. In essence, software in this layer of the framework architecture provides the mechanism for tools integration. Every CASE tool is "plugged into" the object management layer. Working in conjunction with the CASE repository, the OML provides integration services—a set of standard modules that couple tools with the repository. In addition, the OML provides configuration management services by enabling the identification of all configuration objects, performing version control, and providing support for change control, audits, and status accounting.

* Shared repository layer - CASE database and access control functions enabling the OML to interact with the database

6.10.5 Repository For I- CASE

The repository for an I-CASE environment is the set of mechanisms and data structures that achieve data/tool and data/data integration. It provides the obvious functions of a database management system, but in addition, the repository performs or precipitates the following functions [FOR89b]:

- *Data integrity* includes functions to validate entries to the repository, ensure consistency among related objects, and automatically perform "cascading" modifications when a change to one object demands some change to objects related to it.
- *Information sharing* provides a mechanism for sharing information among multiple developers and between multiple tools, manages and controls multi-user access to data and locks or unlocks objects so that changes are not inadvertently overlaid on one another.
- *Data/tool integration* establishes a data model that can be accessed by all tools in the I-CASE environment, controls access to the data, and performs appropriate configuration management functions.
- *Data/data integration* is the database management system that relates data objects so that other functions can be achieved.
- *Methodology enforcement* defines an entity-relationship model stored in the repository that implies a specific paradigm for software engineering; at a minimum, the relationships and objects define a set of steps that must be conducted to build the contents of the repository.
- *Document standardization* is the definition of objects in the database that leads directly to a standard approach for the creation of software engineering documents.

To achieve these functions, the repository is defined in terms of a meta-model. The

6.10.5 **Important DBMS Features Relevant to CASE Repositories**

- *Nonredundant data storage.* Each object is stored only once, but is accessible by all CASE tools that need it.
- *High-level access.* A common data access mechanism is implemented so data handling facilities do not have to be duplicated in each CASE tool.
    - *Data independence.* CASE tools and the target applications are isolated from physical storage so they are not affected when the hardware configuration is changed.
    - *Transaction control.* The repository implements record locking, two-stage commits, transaction logging, and recovery procedures to maintain the integrity of the data when there are concurrent users.
    - *Security.* The repository provides mechanisms to control who can view and modify information contained within it.
    - *Ad hoc data queries and reports.* The repository allows direct access to its contents through a convenient user interface such as SQL or a forms-oriented "browser," enabling user-defined analysis beyond the standard reports provided with the CASE tool set.
    - *Openness.* Repositories usually provide a simple import/export mechanism to enable bulk loading or transfer.
- *Multiuser support.* A robust repository must permit multiple developers to work on an application at the same time. It must manage concurrent access to the database by multiple tools and users with access arbitration and lock-

Special Features of CASE Repositories

* Storage of sophisticated data structures (diagrams, documents, files, simple variables, information model describing relationships and semantics of data stored in repository)
* Integrity enforcement (business rules, policies, constraints, and requirements on the information being entered into repository, triggers may be used to check the validity of the design models in real time)
* Semantic-rich tool interface (repository meta-model contains semantics that enable a variety of tools to interpret meaning of data stored in the repository)
* Process/project management(contains information about the software application, the characteristics of each project, and the

organization's general process for software development - phases, tasks, deliverables)

Software Configuration Management Features Relevant to CASE Repositories

* Versioning
* Dependency tracking and change management
* Requirements tracing
* Configuration management
* Audit trails