Software Engineering

Purbanchal University

BCA 5th Semester

S@R0Z

CHAPTER 01

1.1 The Evolving Role of Software

- Software takes on a dual role: product and at the same time, the vehicle for delivering a product.
- As a product, it delivers the computing potential embodied by computer hardware, or more broadly, a network of computers that are accessible by local hardware.
- As the vehicle used to deliver the product, software acts as the basis for the control of the computer, the communication of information, and the creation and control of other programs.
- The role of computer software has undergone significant change over a time span of little more than 50 years.
- Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems.
- The programmers are asked some questions when modern computer-based systems are built:
  - Why does it take so long to get software finished?
  - Why are development costs so high?
  - Why can't we find all the errors before we give the software to customer?
  - Why do we continue to have difficulty in measuring progress as software is being developed?

1.2 Software

- Software is (1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs.

1.2.1. Software Characteristics

- Software is logical rather than a physical system element.
- Software has characteristics that are considerably different than those of hardware:

1. Software is developed or engineered; it is not manufactured in the classical sense.

- Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different.
- In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.
- Software costs are concentrated in engineering.
- Software projects cannot be managed as if they were manufacturing projects.

2. Software doesn't "wear out."

- The failure rate of the hardware is high as the hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.

- Stated simply, the hardware begins to wear out.
- Software is not susceptible to the environmental maladies that cause hardware to wear out.
- Undiscovered defects will cause high failure rates early in the life of a program.
- However, these are corrected ideally, without introducing other errors.
- Software does not wear out, but it deteriorates.
- Software maintenance involves considerably more complexity than hardware maintenance.

3. **Although the industry is moving toward component-based assembly, most software continuous to be custom built.**

- Each integrated circuit has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines.
- After each component is selected, it can be ordered off the shelf.
- As an engineering discipline evolves, a collection of standard design components is created.
- The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new.
- A software component should be designed and implemented so that it can be reused in many different programs.

1.2.2. Software Application

- Software may be applied in any situation for which a pre-specified set of procedural steps has been defined.
- Information content and determinacy are important factors in determining the nature of a software application.
- Content refers to the meaning and form of incoming and outgoing information.
- Information determinacy refers to the predictability of the order and timing of information.
- It is somewhat difficult to develop meaningful generic categories for software applications.
- The following software areas indicate the breadth of potential applications:

System Software

- System software is a collection of programs written to service other programs.
- Some system software (e.g. compilers, editors, and file management utilities) process complex, but determinate, information structures.
- Other system applications (e.g. drivers, operating system components etc.) process largely intermediate data.
- In either case, the system software area is characterized by heavy interaction with computer hardware, resource sharing, and sophisticated process management.

Real time Software

- Software that monitors, analyzes, controls real world events as they occur is called real time software.
- Elements of real time software include a data gathering component that collects and formats information from an external environment, a control output component that coordinates all other components so that real-time response can be maintained.

Business Software

- Business information processing is the largest single software application area.
- In addition to conventional data processing application business software applications also encompass interactive computing.
- Applications in this area restructure existing data in a way that facilitates business operations or management decision making.

Engineering and scientific Software

- Engineering and scientific software have been characterized by “number crunching” algorithms.
- Applications range from astronomy to volcano-logy and from molecular biology to automated manufacturing.

Embedded Software

- Embedded software resides in read-only memory and is used to control products and systems for the customer and industrial markets.
- They can perform very limited and esoteric functions or provide significant function and control capability.

CHAPTER 02

2.1 Software Engineering: A Layered Technology

- The definition of Software Engineering proposed by Fritz Bauer is: *[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.*
- This definition says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of a mature process.
- The IEEE [IEEE93] has developed a more comprehensive definition when it states: *Software Engineering: 1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. 2) The study of approaches as in 1).*

2.1.1 Process, Methods, and Tools

- The foundation of Software Engineering is the *process* layer.
- Software Engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.
- Process defines a framework for a set of key process areas that must be established for effective delivery of Software Engineering technology.
- Software Engineering *methods* provide the technical how-tos for building software.
- Software Engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.
- Software Engineering tools provide automated or semi-automated support for the process and the methods.
- When tools are integrated so that another can use information created by one tool, a system for the support of software development, called computer-aided software engineering, is established.

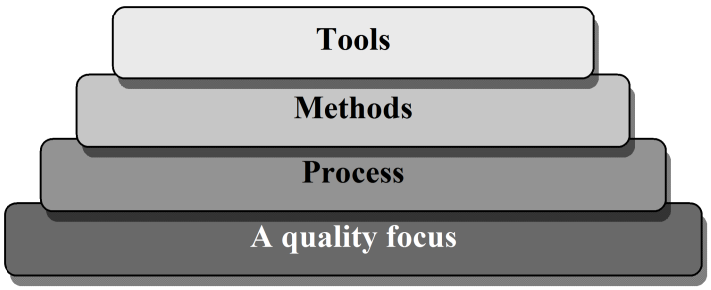

Fig: *Software Engineering Layers*

2.1.2 A generic view of Software Engineering

- Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Regardless of the entity to be engineered, the following questions must be asked and answered:

- What is the problem to be solved?
- What characteristics of the entity are used to solve the problem?
- How will the entity (and the solution) be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors that were made in the design and construction of the entity?
- How will the entity be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the entity?

- **The definition phase focuses on what.**
- The Software Engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system.

- **The development phase focuses on how.**
- During development a software engineer attempts to define how data are to be structured, how function is to be implemented within software architecture, how procedural details are to be implemented within software architecture, how design will be translated into a programming language, and testing will be performed.
- The support phase focuses on change associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements.
- The support phase re-applies the steps of the definition and development phases but does so in the context of existing software.

- Four types of change are encountered during the support phase:

  - Correction
  - Adaptation
  - Enhancement
  - Prevention

2.2 The Software Process

- A software process can be characterized as shown in fig. below:
- A common process framework is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.
- The software Engineering Institute (SEI) has developed a comprehensive model predicted on a set of Software Engineering capabilities that should be present as organizations reach different levels of process maturity.
- The SEI approach provides a measure of the global effectiveness of a company's Software Engineering practices and establishes five process maturity levels that are defined in the following manner:

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

- Level 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.

- Level 3: **Defined.** The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software.

- Level 4: **Managed.** Detailed measures of the software process and product quality are collected. Both software process and products are quantitatively understood and controlled using detailed measures.

- Level 5: **Optimizing.** Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies.

- The five levels defined by the SEI were derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM.
- The SEI has associated key process areas (KPAs) with each of the maturity levels.
- The KPAs describe those software engineering functions that must be present to satisfy good practice at a particular level.
- Each KPA is described by identifying the following characteristics:
  - Goals- the overall objectives that the KPA must achieve.
  - Commitments- requirements that must be met to achieve the goals or provide proof to intent to comply with the goals.
  - Abilities- those things that must be in place to enable the organization to meet the commitments
  - Activities- the specific tasks required to achieve the KPA function.
  - Methods for monitoring implementation- the manner in which proper practice for the KPA can be verified.

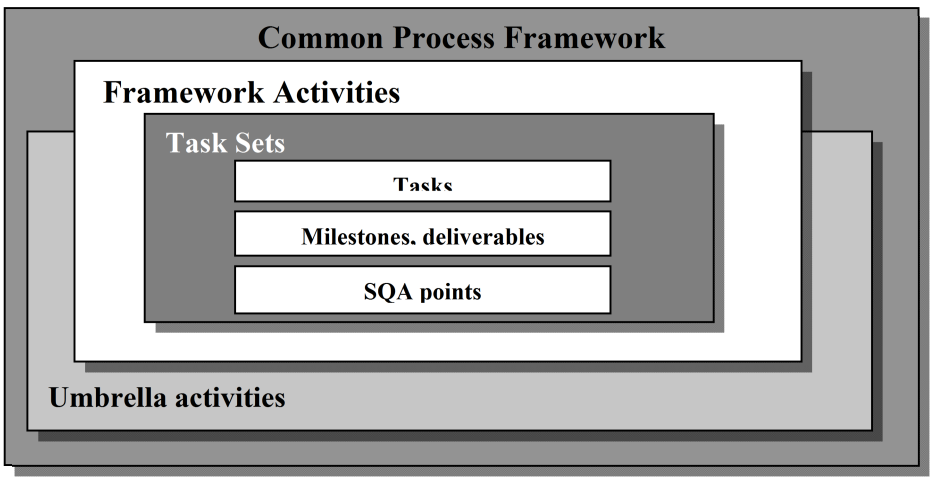

Fig: The software process

2.3 Software Process Model

- A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required.

- All software development can be characterized as a problem solving loop (fig. a) in which four distinct stages are encountered: status quo, problem definition, technical development, and solution integration.
- Status quo "represents the current state of affairs"; problem definition identifies the specific problem to be solved; technical development solves the problem through the application of some technology, and solution integration delivers the results to those who requested the solution in the first place.
- This problem solving loop applies to software engineering work at many different levels of resolution.

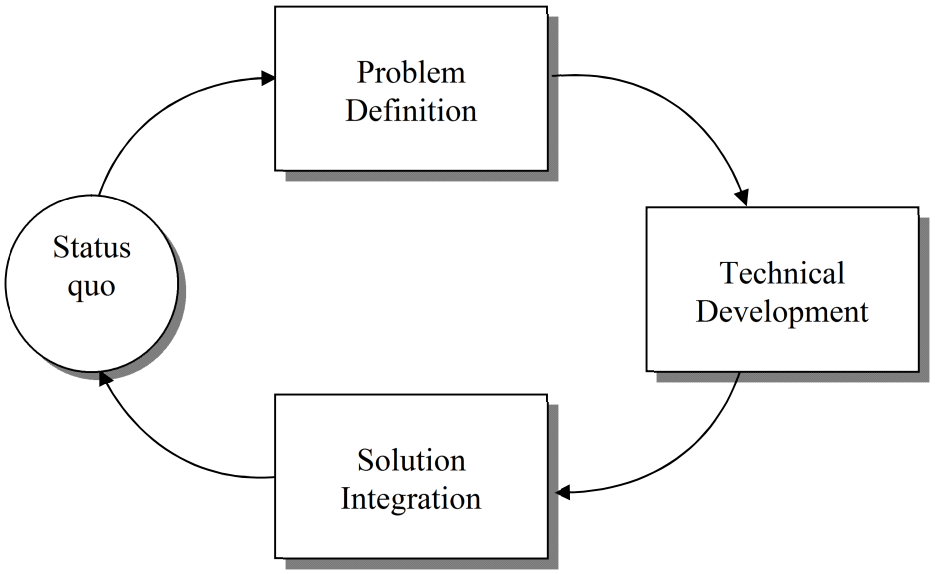

Fig :(a) *The phases of a problem solving loop*

- In (fig. b) each stage in the problem solving loop contains an identical problem solving loop, which contains still another problem solving loop; for software, a line of code.
- Realistically, it is difficult to compartmentalize activities as neatly as (fig. b) implies because cross talk occurs within and across stages.
- Yet, this simplified view leads to a very important idea: regardless of the process model that is chosen for a software project, all of the stages- status quo, problem definition, technical development and solution integration- coexist simultaneously at some level of detail.

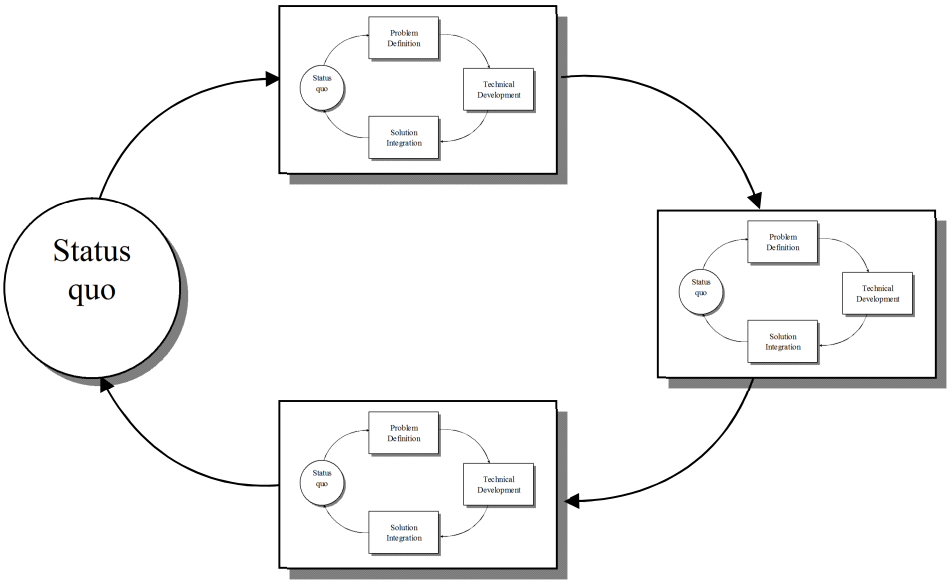

Fig:(b) *The phases within phases of a problem solving loop*

2.4 The Linear Sequential Model

- The linear sequential model also known as classic life cycle or the waterfall model, suggests a systematic, sequential approach to software development that begins at the system level and progress through analysis, design, coding, testing and support.
- The fig. below illustrates the linear sequential model for software engineering.

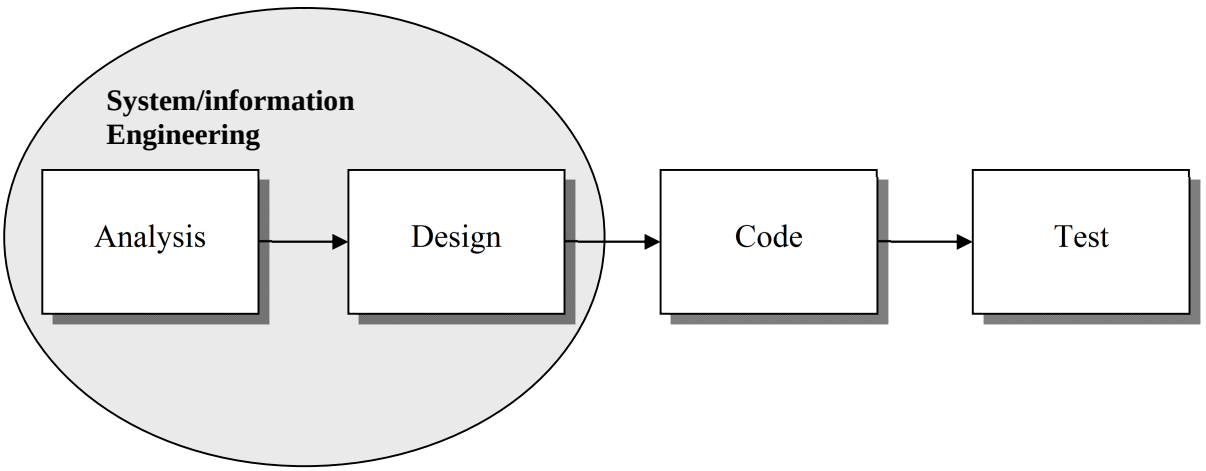

Fig: *The linear sequential model*

- The linear sequential model encompasses the following activities

  - **System/information engineering and modeling.**

    - Because software is always part of a larger system, work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software.

- This system view is essential when the software must interact with other elements such as hardware, people, and database.
- Information engineering encompasses requirements gathering at the strategic business level and at the business area level.

➢ **Software requirements analysis.**

- The requirements gathering process is intensified and focused specially on software.
- To understand the nature of programs to be built, the software engineer must understand the information domain for the software, as well as required function, behavior, and interface.
- Requirements for both the system and the software are documented and reviewed with the customer.

➢ **Design.**

- Software design is actually a multi-step process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations and procedural detail.
- The design process translates requirements into a representation of the software that can be assessed for quality before coding begins.
- The design is documented and becomes part of the software configuration.

➢ **Code generation.**

- The design must be translated into a machine-readable form.
- The code generation step performs this task.
- If design is performed in a detailed manner, code generation can be accomplished mechanistically.

➢ **Testing.**

- Once code has been generated, program testing begins.
- The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

➢ **Support.**

- Software will undoubtedly undergo change after it is delivered to the customer.
- Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment, or because the customer requires functional or performance enhancements.

- The linear sequential model is the oldest and the most widely used paradigm for software engineering.
- Among the problems that are sometimes encountered when the linear sequential model is applied are:

➢ Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly; changes can cause confusion as the project team proceeds.
➢ It is often difficult for the customer to state all requirements explicitly.
➢ A customer must have patience. A major blunder, if undetected until the working program is reviewed, can be disastrous.

- The classic life cycle paradigm has a definite and important place in software engineering work.
- It does have weakness; it is significantly better than a haphazard approach to software development.

2.5 The Prototyping Model

- The prototyping paradigm (fig. below) begins with requirements gathering.
- Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- The prototype is evaluated by the customer by the customer/user and used to refine requirements for the software to be developed.
- Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.
- If a working prototype is built, the developer attempts to use existing program fragments or applies tools that enable working programs to be generated quickly.
- The prototype can serve as "the first system".
- Prototyping can also be problematic for the following reasons:

  - The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability.

  - The developer often makes implementation compromises in order to get a prototype working quickly. The less-than-ideal choice has now become an integral part of the system.

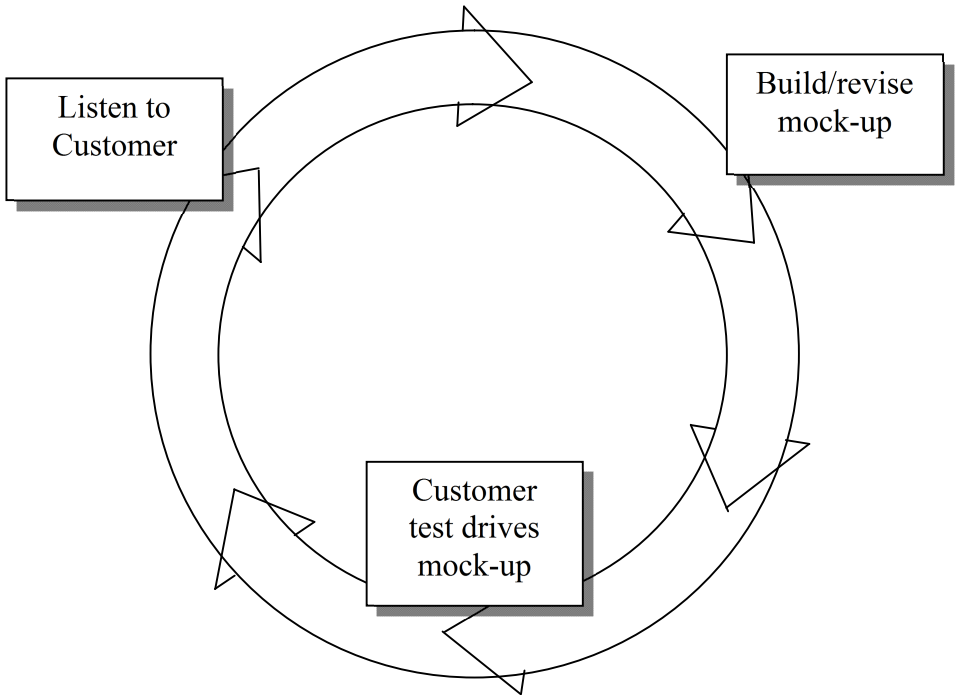

Fig: *The Prototyping paradigm*

- Although problems can occur, prototyping can be an effective paradigm for software engineering.
- The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements.
- It is then discarded and the actual software is engineered with an eye toward quality and maintainability.

2.6 The RAD Model

- Rapid Application Development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle.
- The RAD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction.
- The RAD process enables a development team to create a "fully functional system" within very short time periods.
- The RAD approach encompasses the following phases:
  - ***Business modeling.*** The information flow among business functions is modeled in a way that answers the following questions.
    - What information drives the business process?
    - What information is generated?
    - Who generates it?
    - Where does the information go?
  - ***Data modeling.*** The information flow defined as part of the business-modeling phase is refined into a set of data objects that are needed to support the business. The characteristics of each object are identified and relationships between these objects defined.
  - ***Process modeling.*** The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function
  - ***Application generation.*** RAD assumes the use of fourth generation techniques. RAD process works to reuse existing program components or create reusable components.
  - ***Testing and turnover.*** Since RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time.

Like all process models, the RAD approach also has drawbacks:

- For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much-abbreviated time frame.
- Not all types of applications are appropriate for RAD. If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
- RAD is not appropriate when technical risks are high. This occurs when a new application makes heavy use of new technology or when the new software requires a high degree of interoperability with existing computer programs.

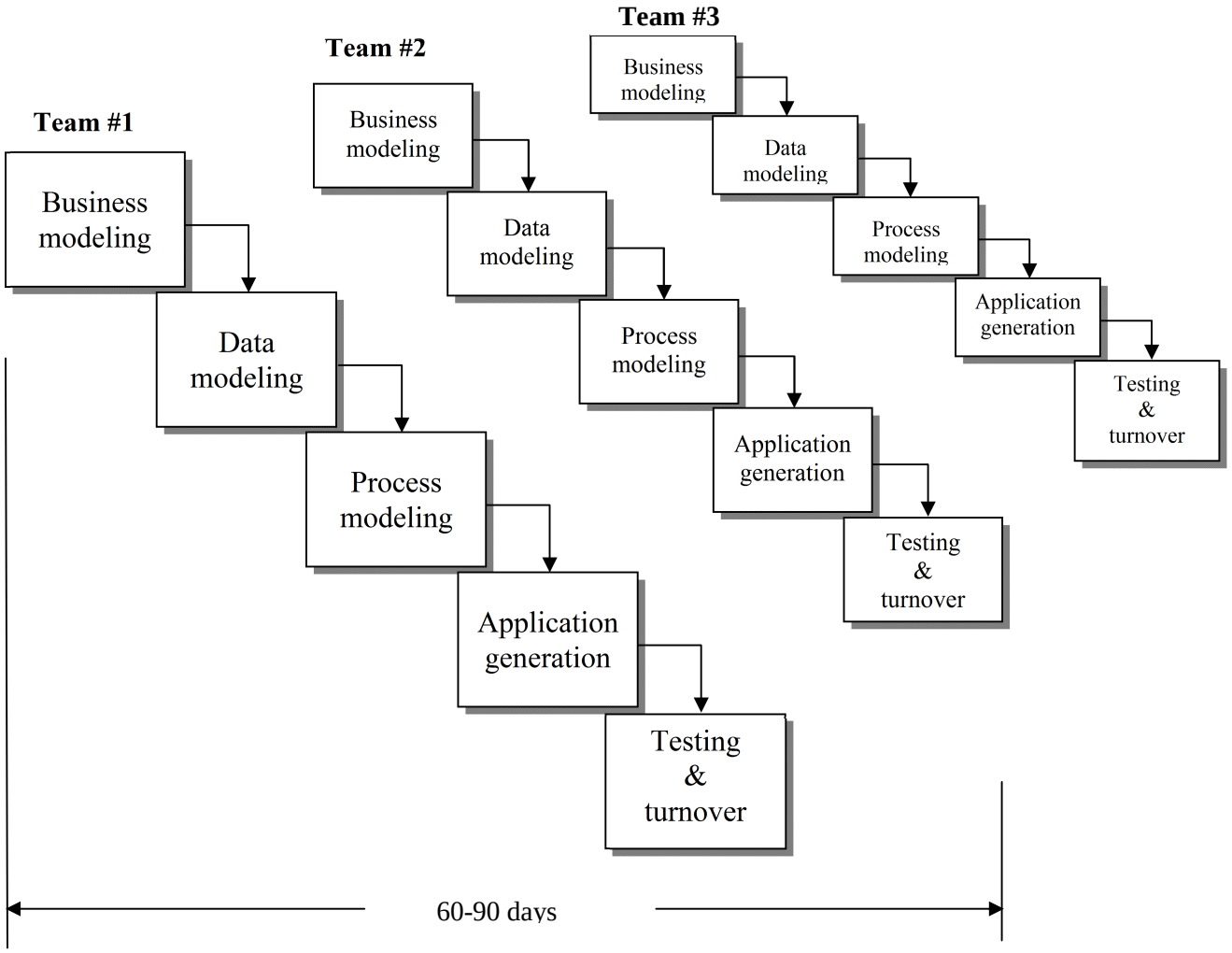

Fig: *The RAD Model*

2.7 Evolutionary Software Process Models

- Business and product requirements often change as development proceeds, making a straight path to an end product unrealistic.
- Software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.
- Evolutionary model are iterative.

2.7.1 The Incremental Model

- The incremental combines elements of the linear sequential model with iterative philosophy of prototyping.
- When an incremental model is used, the first increment is often a core product.
- The core product is used by customer.
- As a result of use and/or evaluation, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.

- This process is repeated following the delivery of each increment, until the complete product is produced.
- Unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment.
- Early increments can be implemented with fewer peoples.
- If the core product is well received, then additional staff can be added to implement the next increment.

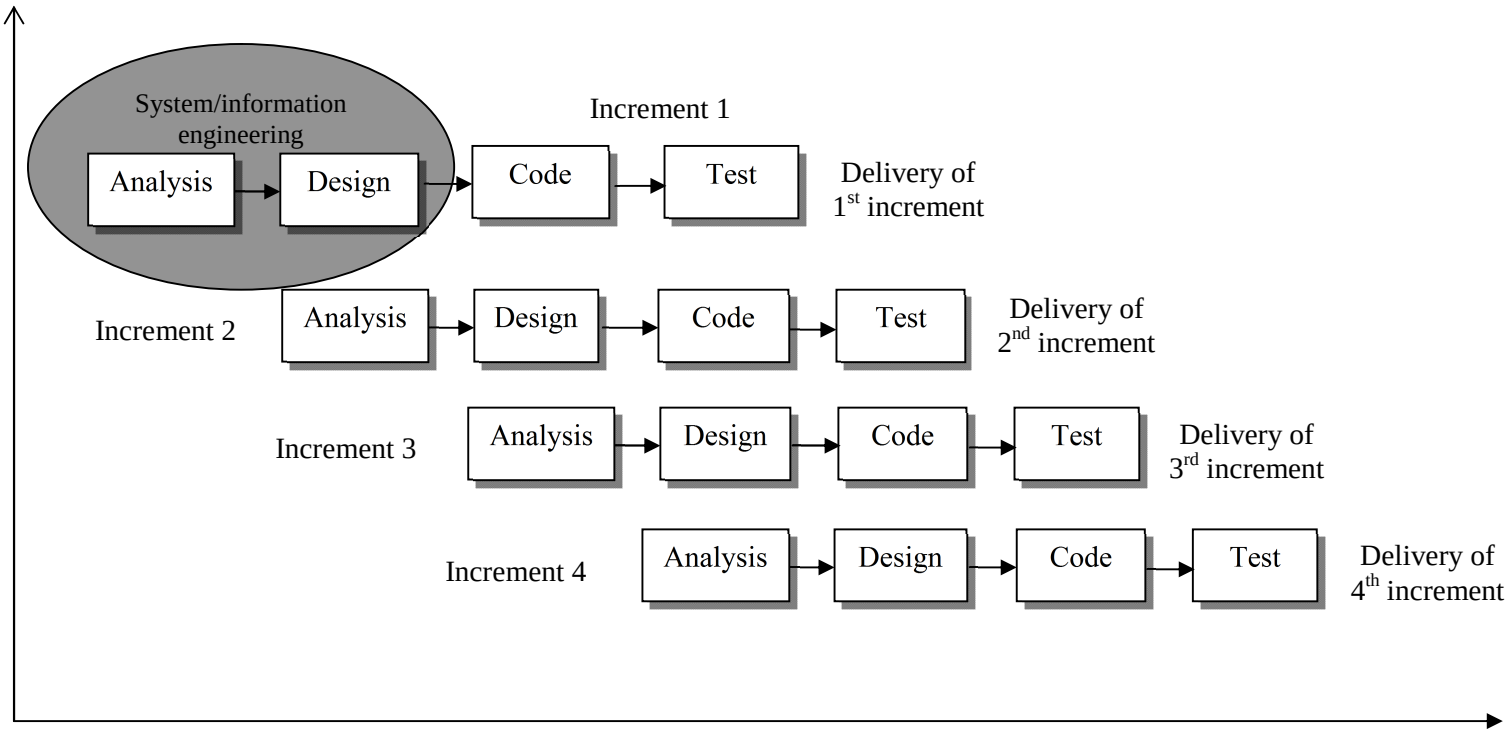

Fig: *The Incremental model*

2.7.2 The Spiral Model

- The Spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model.
- It provides the potential for rapid development of incremental versions of the software.
- Using spiral model, software is developed in a series of incremental releases.
- A spiral model is divided into a number of framework activities, also called task regions.
- Typically there are between three and six tasks regions.
  - ***Customer communication-*** tasks required establishing effective communication between developer and customer.
  - ***Planning-*** tasks required defining resources, timelines, and other project-related information.
  - ***Risk analysis-*** tasks required to assess both technical and management risks.
  - ***Engineering-*** tasks required building one or more representations of the application.
  - ***Construction and release-*** tasks required to construct, test, install, and provide user support.

- ***Customer evaluation-*** tasks required obtaining customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

- Each of the regions is populated by a set of work tasks, called a task set, that are adapted to the characteristics of the project undertaken.
- As this evolutionary process begins, the software engineering teams moves around the spiral in a clockwise direction, beginning at the center.
- Unlike classical process models that end when the software is delivered, the spiral model can be adapted to apply throughout the life of the computer software.
- A "concept development project" starts at the core of the spiral and will continue until concept development is complete.
- The spiral model is a realistic approach to the development of large-scale systems and software.
- The spiral model uses prototyping as a risk reduction mechanism but, more important, enables the developer to apply the prototyping approach at any stage in the evolution of the product.
- The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.
- It may be difficult to convince customers that the evolutionary approach is controllable.
- This model has not been used as widely as the linear sequential or prototyping paradigms.

2.7.3 The WINWIN Spiral Model

- Boehm's WINWIN spiral model defines a set of negotiation activities at the beginning of each pass around the spiral.
- Rather than a single customer communication activity, the following activities are defined.

  - Identification of the system or subsystem's key "stakeholders"

  - Determination of the stakeholders' "win conditions"

  - Negotiation of the stakeholders' wins conditions to reconcile them into a set of win-win conditions for all concerned.

- Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to software and system definition.
- The WINWIN spiral model introduces three process milestones, called anchor points that help establish the completion of one cycle around the spiral and provide decision milestones before the software project proceeds.

  - The first anchor point, life cycle objectives (LCO), defines a set of objectives for each major software engineering activity.

  - The second anchor point, life cycle architecture (LCA), establishes objectives that must be met as the system and software architecture is defined.

- Initial operational capability (IOC) is the third anchor point and represents a set of objectives associated with the preparation of the software for installation/distribution, and assistance required by all parties that will use or support the software.

2.7.4 The Concurrent Development Model

- The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states.
- The concurrent process model is often used as the paradigm for the development of client/server applications.
- A client/server system is composed of a set of functional components.
- In reality, the concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project.

2.7.5 Component-based development

- The component-based development (CBD) model incorporates many of the characteristics of the spiral model.
- It is evolutionary in nature, demanding an iterative approach to the creation of software.
- The component-based development model composes applications from prepackaged software components.
- The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.
- The unified software development process is representative of a number of component-based development models that have been proposed in the industry.
- Using the unified Modeling language (UML), the unified process defines the components that will be used to build the system and the interfaces that will connect the components.

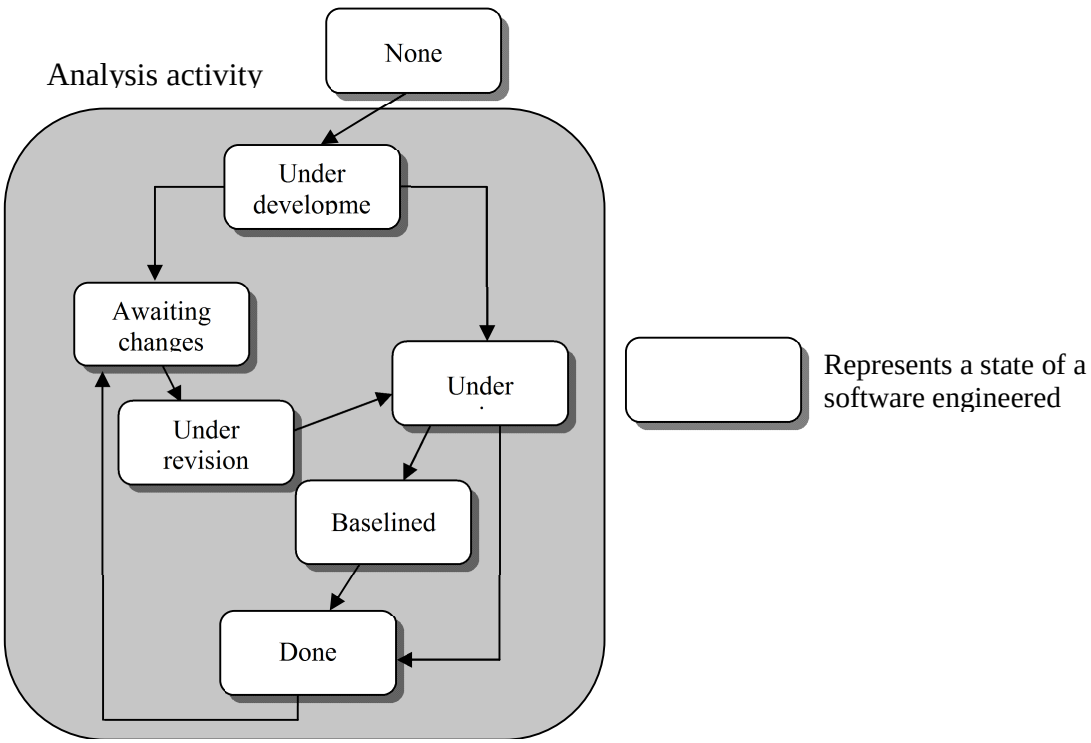

Fig: *One element of the concurrent process model*

CHAPTER 03

3.1 Computer based systems

- The word system is possibly the most overused and abused term in the technical lexicon.
- We use the adjective describing system to understand the context in which the word is used.
- We define computer system as:
    - *A set of arrangement of elements that are organized to accomplish some predefined goal by processing information.*
- To accomplish the goal, a computer-based system makes use of a variety of system elements:
    - **Software.** Computer programs, data structures, and related documentation that serve to affect the logical method, procedure, or control that is required.
    - **Hardware.** Electronic devices that provide computing capability, the interconnectivity devices that enable the flow of data, and electromechanical devices that provide external world function.
    - **People.** Users and operators of hardware and software.
    - **Database.** A large, organized collection of information that is accessed via software.
    - **Documentation.** Descriptive information that portrays the use and / or operation of the system.
    - **Procedures.** The steps that define the specific use of each system element or the procedural context in which the system resides.
- One complicating characteristic of computer-based systems is that the elements constituting one system may also represent one macro element of a still larger system.
- The macro element is a computer-based system that is one part of a larger computer-based system.
- At lowest level of the hierarchy we have a numerical control machine, robots, and data entry devices.
- At level in the hierarchy, a manufacturing cell is defined.
- The role of the system engineer is to define the elements for a specific computer-based system in the context of the overall hierarchy of systems.

3.2 The System Engineering Hierarchy

- The system engineering process usually begins with a “world view”.
- The world view is refined to focus more fully on specific domain of interest.
- Within a specific domain of interest, the need for targeted system elements is analyzed.
- Finally the design and construction of a targeted system element is initiated.
- At the top of the hierarchy, a very broad context is established and, at a bottom, detailed technical activities, performed by the relevant engineering discipline, are conducted.
- The world view is composed of a set of domains, which can each be a system or system of systems in its own way.

- Each domain is composed of specific elements each of which serves some in accomplishing the objective and goals of the domain or component.
- Finally, each element is implemented by specifying the technical components that achieve the necessary function for an element.
- In the software context, a component could be a computer program, a reusable program component, a module, a class or object, or even a programming language statement.

3.2.1 System Modeling

- System engineering is a modeling process.
- Whether the focus is on the world view or the detailed view, the engineer creates models that:
  - Define the processes that serve the needs of the view under consideration.
  - Represent the behavior of the processes and the assumptions on which the behavior is based.
  - Explicitly define both exogenous and endogenous input to the model.
  - Represent all linkages that will enable the engineer to better understand the view.
- To construct a system model, the engineer should consider a number of restraining factor:
  - Assumptions that reduce the number of possible permutations and variations, thus enabling a model to reflect the problem in a reasonable manner.
  - Simplifications that enable the model to be created in a timely manner.
  - Limitations that help to bound the system.
  - Constraints that will guide the manner in which the model is created and the approach taken when the model is implemented.
  - Preferences that indicate the preferred architecture for all data, functions, and technology.

3.2.2 System Simulation

- Many computer-based systems interact with the real world in a reactive fashion.
- The developers of reactive systems sometimes struggle to make them perform properly.
- Until recently, it has been difficult to predict the performance, efficiency, and behavior of such systems prior to building them.
- In a very real sense, the construction of many real-time systems was an adventure in “flying”.
- If the system crashed due to incorrect function, inappropriate behavior, or poor performance, we picked up the pieces and started over again.
- Many systems in the reactive category control machines and/or processes that must operate with an extremely high degree of reliability.
- Today, software tools for system modeling and simulation are being used to help to eliminate surprises when reactive, computer-based systems are built.

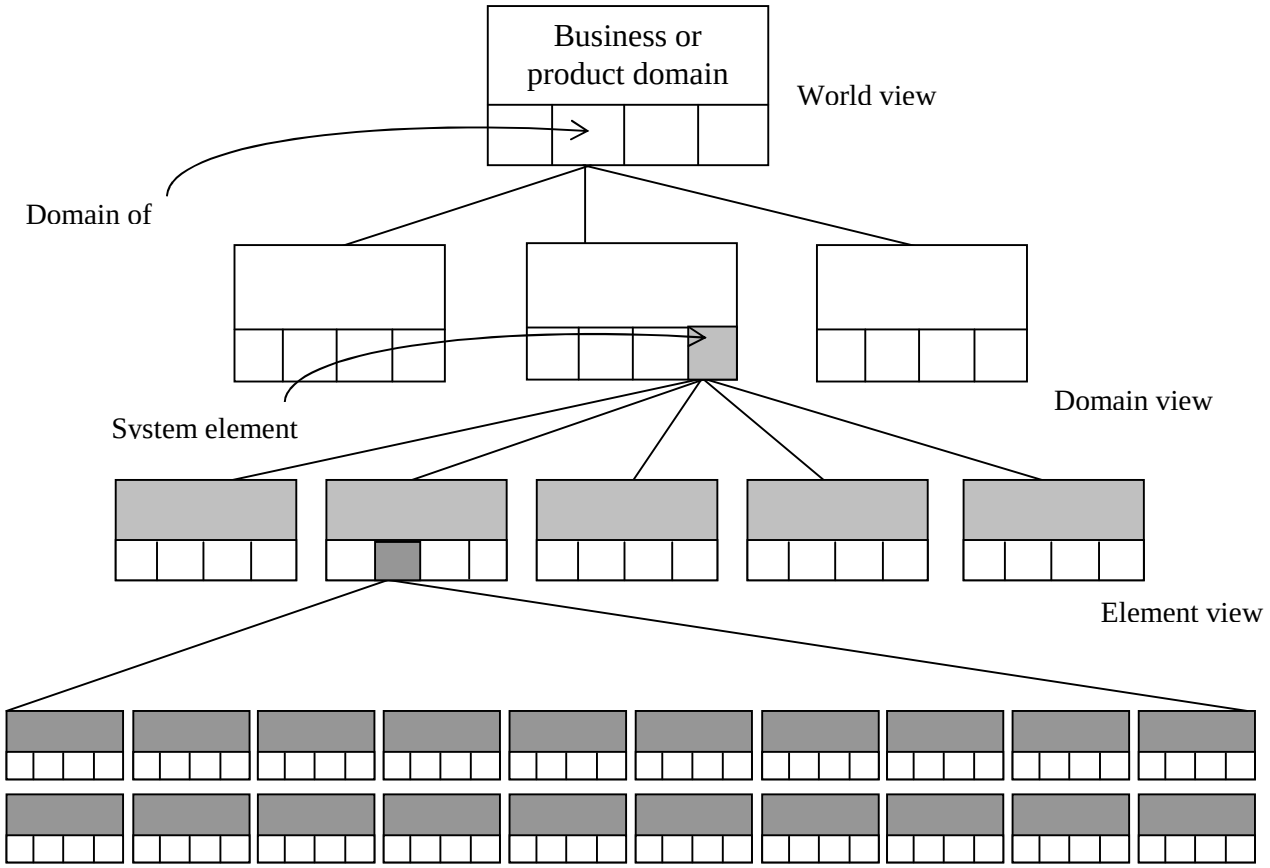

Fig: *The system engineering hierarchy*

3.3 System Modeling

- Every computer-based system can be modeled as information as information transforms using an input-processing-output template.
- To develop the system model, a system model template is used.
- The system engineer allocates system elements to each of five processing regions within the template: (1) user interface, (2) input, (3) system function and control, (4) output, and (5) maintenance and self-test.
- The system model template enables the analyst to create a hierarchy of detail.
- A system context diagram (SCD) resides at the top level of the hierarchy.
- The context diagram "establishes the information boundary between the system being implemented and the environment in which the system is to operate".
- The initial system flow diagram becomes the top node of a hierarchy of SFDs.
- Each of the SFDs for the system can be used as a starting point for subsequent engineering steps for the subsystem that has been described.
- Subsystems and the information that flow between them can be specified for subsequent engineering work.
- A narrative description of each subsystem and a definition of all data that flow between subsystems become important elements of the system specification.

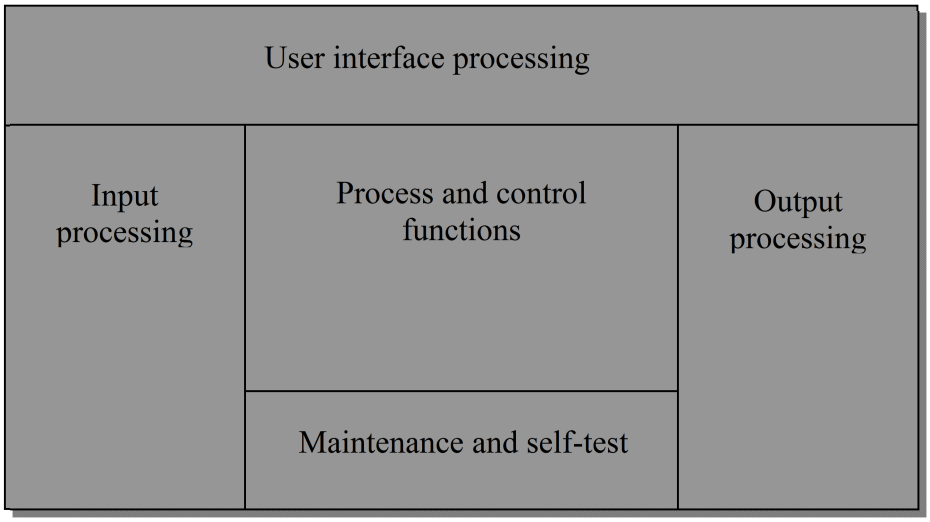

Fig: *System model template.*

CHAPTER 04

4.1 Project Planning

- Effective management of a software project depends on thoroughly planning the progress of the project.
- The project management must anticipate problems which might arise and prepare tentative solutions to those problems.
- A plan, drawn up at the start of a project, should be used as the driver for the project.
- This initial plan should be the best possible plan given the available information.
- It evolves as the project progresses and better information becomes available.
- The planning process starts with an assessment of the constraints affecting the project.
- A schedule for the project is drawn up and the activities defined in the schedule are initiated or given permission to continue.
- After sometime, progress is reviewed and discrepancies noted.
- Project managers revise the assumptions about the project as more information becomes available.
- They re-plan the project schedule.
- The objective of this review is to find some alternative approach to development which falls within the project constraint and meets the schedule.

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 of the system, maintenance costs and effort required.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

Fig:- *Types of plan*

4.1.2. The Project Plan

- The project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.
- In some organizations, the project plan is a single document including all the different types of plan.
- In other cases, the project plan is solely concerned with the development process.
- Most plan should include the following sections:

1. *Introduction* This briefly describes the objectives of the project and sets out the constraints which affect the project management.

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

4. *Hardware and Software resource requirements*. This describes the hardware and the support software required to carry out the development.

5. Work breakdown. This describes the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.

6. *Project schedule*. This describes 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 describes the management reports which should be produced, when these should be produced and the project monitoring mechanisms used.

- The project plan should be regularly revised during the project.
- A document organization which allows for the straightforward replacement of sections should be used.

4.1.2. Milestones and deliverables

- Managers need information.
- As software is intangible, this information can only be provided as documents that describe the state of the software being developed.
- Without this information, it is impossible to judge progress and cost estimates and schedules cannot be updated.
- When planning a project, a series of milestones should be established where a milestone is an end-point of a software process activity.
- At each milestone, there should be a formal output, such as a report, that can be presented to management.
- Milestones should represent the end of a distinct, logical stage in the project.
- A deliverable is a project result that is delivered to the customer.
- It is delivered at the end of some major project phase such as specification, design, etc.
- Deliverables are usually milestones but milestones need not be deliverables.
- Milestones may be internal project results that are used by the project manager to check project progress but which are not delivered to the customer.

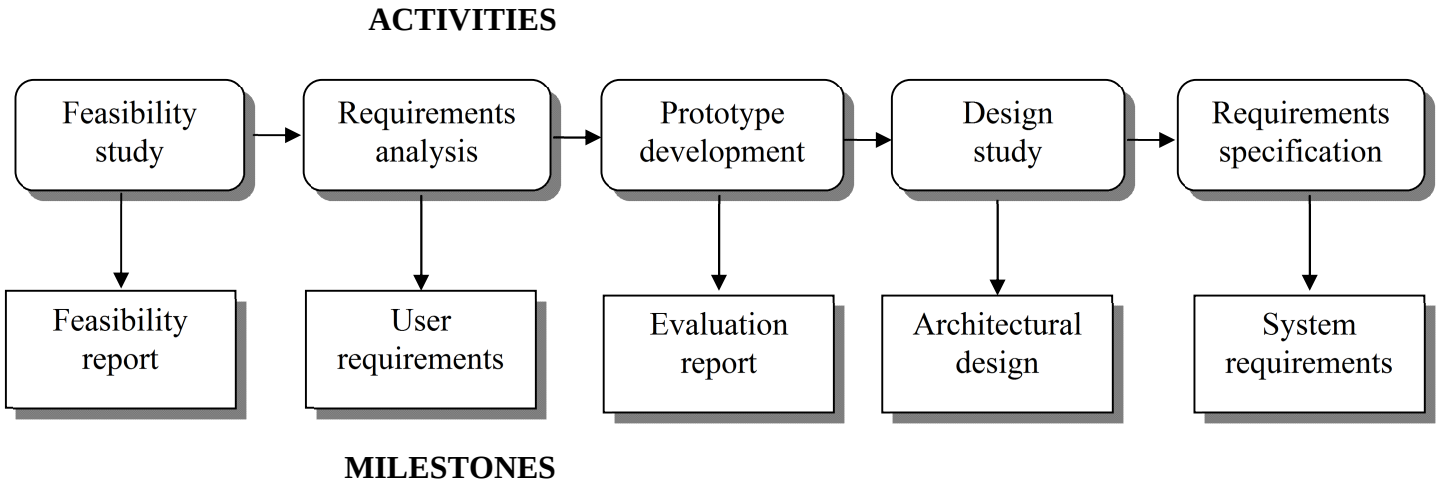

Fig: - *Milestones in the requirements process*

4.2. Project Scheduling

- Project scheduling is a particularly demanding task for software managers.
- Managers estimate the time and resources required to complete activities and organize them in a coherent sequence.
- Schedule estimation is further complicated by the fact that different project may use different design methods and implementation languages.
- Software scheduling is no different from scheduling any other type of large advanced projects.
- Schedules must be continually updated as better progress information becomes available.
- Project scheduling involves separating the total work involved in a project into separate activities and judging the time required to complete these activities.
- Projects schedulers must coordinate these parallel activities and organize the work so that the workforce is used optimally.
- They must avoid a situation where the whole project is delayed because a critical task is unfinished.
- In estimating schedules, managers should not assume that every stage of the project will be problem free.
- If the project is new and technically advanced, certain parts of it may turn out to be more difficult and take longer than originally anticipated.
- The project schedule is usually represented as a set of charts showing the work breakdown, activities dependencies and staff allocations.

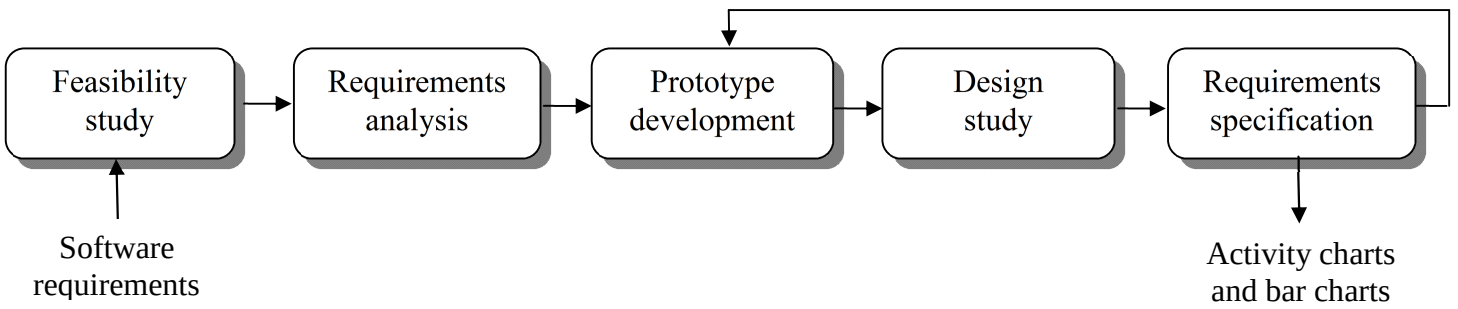

Fig:- *The project scheduling process*

4.2.1. Bar charts and activity networks

- Bar charts and activity networks are graphical notations which are used to illustrate the project schedule.
- Bar charts show who is responsible for each activity and when the activity is scheduled to begin and end.
- Activity networks show the dependencies between the different activities making up a project.
- Bar charts and activity charts can be generated automatically from a database of project information using a project management tool.

4.3. Risk Management

- An important task of a project manager is to anticipate risks which might affect the project schedule or the quality of the software being developed and to take action to avoid these risks.
- The results of the risk analysis should be documented in the project plan along with an analysis of the consequences of a risk occurring.
- Identifying risk and drawing up plans to minimize their effect on the project is called *risk management*.
- Risks may threaten the project, the software that is being developed or the organization.
- This categories of risk can be defined as follows:

    - Project risks are risks which affect the project schedule or resources.
    - Product risks are risks which affect the quality or performance of the software being developed.
    - Business risks are risks which affect the organization developing or procuring the software.

- Risk management is particularly important for software projects because of the inherent uncertainties which most projects face.
- The types of risk which may affect a project depend on the project and the organizational environment where the software is being developed.

Risk	Risk type	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities
Hardware unavailability	Project	Hardware which is essential for the project will not be delivered on schedule
Requirements change	Project & product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project & product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project & product	The size of the system of the system has been underestimated.
CASE tool underperformance	Project	CASE tools which support the project do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology
Product competition	Business	A competitive product is marketed before the system is completed.

Fig: - *Possible software risks.*

- **The process of risk management involves the several stages:**

1) **Risk identification.** Possible project, product and business risks are identified.
2) **Risk analysis.** The likelihood and consequences of these risks are assessed.
3) **Risk planning.** Plans to address the risk either by avoiding it or minimizing its effects on the project are drawn up.
4) **Risk monitoring.** The risk is constantly assessed and plans for risks mitigation are revised as more information about the risk becomes available.

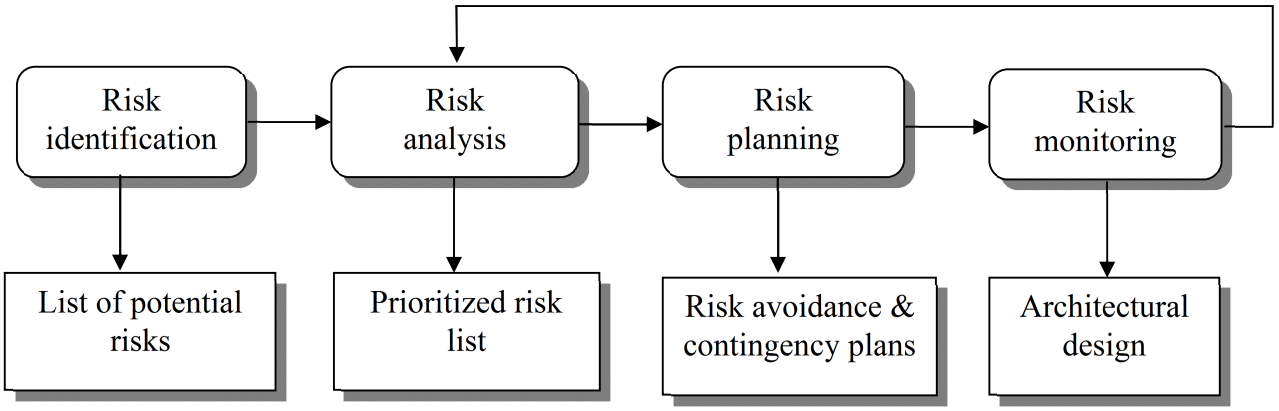

Fig: - *The risk management process*

- The risk management process, like all other project planning, is an iterative process which continuous throughout the project.
- The results of the risk management process should be documented in a risk management plan.

4.3.1. Risk Identification

- Risk identification is the first stage of risk management.
- It is concerned with discovering possible risks to the project.
- The possible types of risk used are:

  - Technology risks. Risks which derive from the software or hardware technologies which are being used as part of the system being developed.
  - People risks. Risks which associated with the people in the development team.
  - Organizational risks. Risks which derive from the organizational environment where the software is being developed.
  - Tools risks. Risks which derive from the CASE tools and other support software used to develop the system.
  - Requirements risks. Risks which derive from changes to the customer requirements and the process of managing the requirements change.
  - Estimation risks. Risks which derive from the management estimates of the system characteristics and the resources required to build the system.

4.3.2. Risk Analysis

- During the risk analysis process, each identified risk is considered in turn and a judgment made about the probability and seriousness of the risk.
- The results of this analysis process should then be tabulated with the table ordered according to seriousness of the risk.

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staffs are ill at critical times in the project.	Moderate	Serious
Software components which should be reused contain defects which limit their functionality.	Moderate	Serious
Changes to requirements which require major design rework are responsible for the project.	Moderate	Serious
The organization is restructured so that different management are responsible for the project.	High	Serious
The database used in the system cannot process as many transactions per second as expected.	Moderate	Serious
The time required to develop the software is underestimated.	High	Serious
Case tools cannot be integrated.	High	Tolerable
Customers fail to understand the impact of requirements changes.	Moderate	Tolerable
Required training for staff is not available.	Moderate	Tolerable
The rate of defect repair is underestimated.	Moderate	Tolerable
The size of the software is underestimated.	High	Tolerable

4.4.3. Risk Planning

- The risk planning process considers each of the key risks which have been identified and identifies strategies to manage the risk.
- It relies on the judgment and experience of the project manger.
- The risk management strategies falls into 3 categories:
    - Avoidance strategies. If we follow this strategy, the probability that risk will arise will be reduced.
    - Minimization strategies. If we follow this strategy, the impact of the risk will be reduced.
    - Contingency plans. If we follow this strategy, if the worst happens, you are prepared for it and have a strategy in place to deal with it.

4.3.3. Risk Monitoring

- Risk monitoring involves regularly assessing each of the identified risks to decide whether or not that risk is becoming more or less probable and whether the effects of the risk have changed.
- It should be a continuous process and, at every management progress review, each of the key risks should be considered separately and discussed by the meeting.

4.4. Management Activities

- It is impossible to write a standard job description for a software manager.
- The job varies, depending the organization and on the software product being developed.
- Most managers take responsibility at some stage for some or all of the following activities:
    - Proposal writing
    - Project planning and scheduling
    - Project costing
    - Project monitoring and reviews
    - Personnel selection and evaluation
    - Report writing and presentations
- The **first** stage in a software project may involve writing a proposal to carry out the project.
- The proposal describes the objectives of the project and how it will be carried out; usually includes cost and schedule estimates.
- The **second** is project planning.
- It is concerned with identifying the activities, milestones and deliverables produced by a project.
- A plan must then be drawn up to guide the development towards the project goals.
- The **third** is cost estimation of the project.
- It is related activity that is concerned with estimating the resources required to accomplish the project plan.
- The **fourth** is project monitoring.
- It is a continuing project activity.
- The manager must keep track of the progress of the project and compare actual and planned progress and costs.
- The **fifth** is personnel selection and evaluation.
- Informal monitoring can often predict potential project problems as they may reveal difficulties as they occur.
- The **sixth** is report writing and presentations.
- The project manager is usually responsible for reporting on the project to both the client and contractor organizations.
- Project managers must write concise, coherent documents which abstract critical information from detailed project reports.
- They must be able to present this information during progress reviews.

CHAPTER 05

- Requirement Engineering is a process that involves all of the activities required to create and maintain a system requirements document.
- There are four generic, high-level requirements engineering process activities:

1. Feasibility Study.
2. Requirement elicitation and analysis.
3. Requirements validation.
4. Requirements Management.

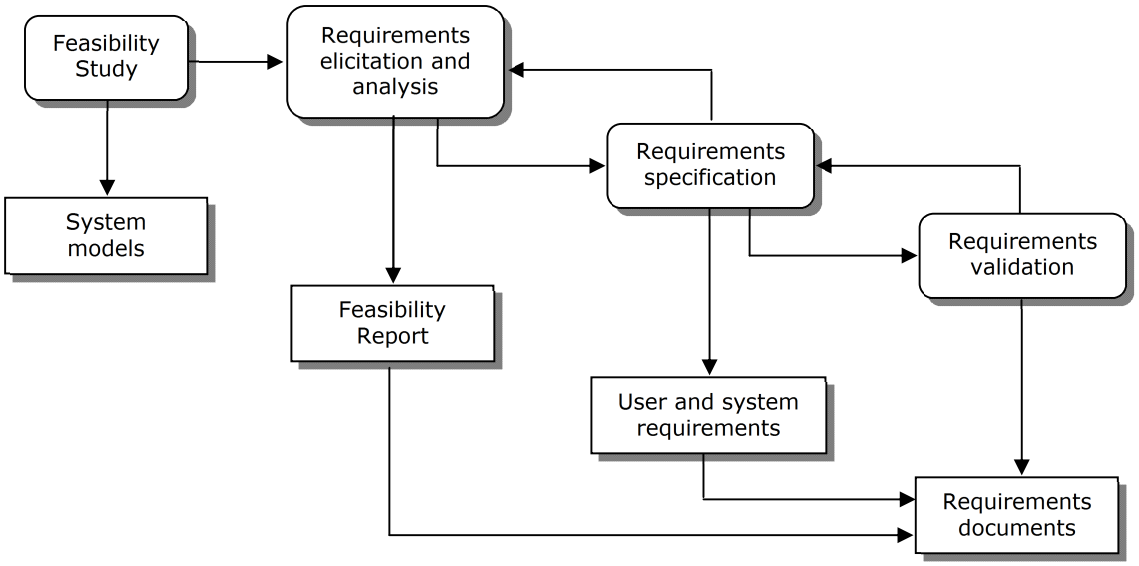

Fig: Requirement Engineering Process

- Some people consider requirements engineering to be the process of applying a structured method such as object-oriented analysis.

5.1 Feasibility Studies

- For all system, the requirements engineering process should start with a feasibility study.
- The input to the feasibility study is an outline description of the system and how it will be used within an organization.
- The result of feasibility study should be a report which recommends whether or not it is worth carrying on with the requirements engineering and system development process.
- A feasibility study focused to study the following questions:

1. Does the system contribute to the overall objectives of the organization?

2. Can the system be implemented using current technology and within given cost and schedule constraints?

3. Can the system be integrated with other systems which are already in place?

- Carrying out a feasibility study involves information assessment, information collection and report writing.
- The information assessment phase identifies the information which is required to answer the three questions set as above.
- When the information is available, the feasibility study report is prepared.
- This should make a recommendation about whether or not the system development should continue.

5.2 Requirements Elicitation and Analysis

- The second step in requirement engineering is requirements elicitation and analysis.
- In this activity, technical software development staffs work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.
- May involve a variety of different kinds of people in a organization.
- The term stakeholder is used to refer to anyone who should have some direct or indirect influence on the system requirements.
- Elicitation and analysis is a difficult process for a number of reasons:

1. Stakeholders often don't really know what they want from the computer system except in the most general terms; they may find it difficult to articulate what they want from the system.
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, must understand these requirements.
3. Different stakeholders have different requirements and they may express this in different ways.
4. Political factors may influence the requirements of the system.
5. The economic and business environment in which the analysis takes place is dynamic.

- A generic process model of the elicitation and analysis process is shown in the figure below:

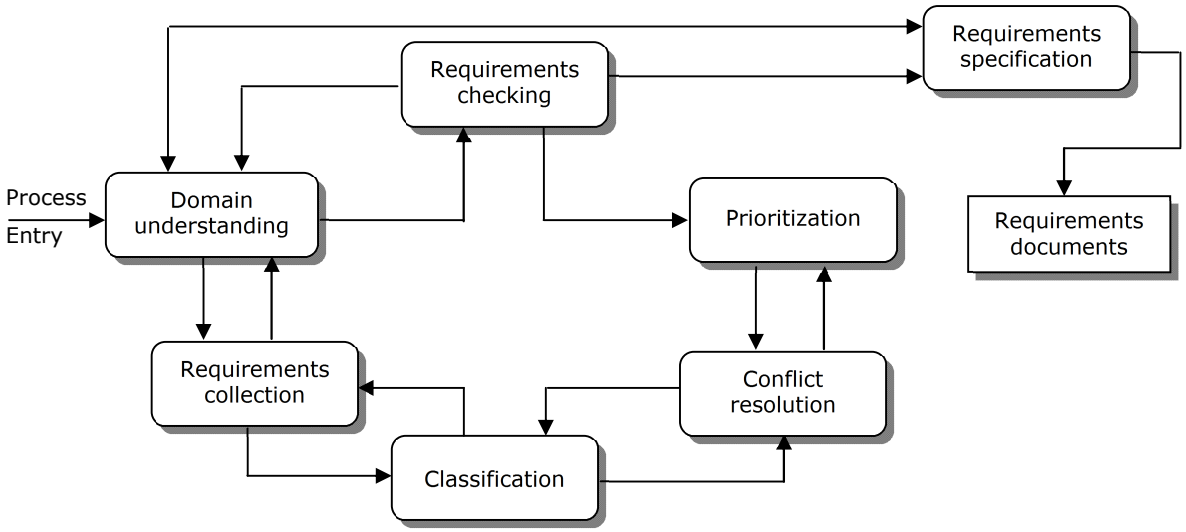

Fig: The requirement elicitation and analysis process.

- **The process activities are:**

1. **Domain Understanding**. Analysts must develop their understanding of the application domain.

2. **Requirements collection**. This is the process of interacting with stakeholders in the system to discover their requirements.

3. **Classification**. This activity takes the unstructured collection of requirements and organizes them into coherent clusters.

4. **Conflict resolution**. Where multiple stakeholders are involved, requirements will surely conflict. This activity is concerned with finding and resolving these conflicts.

5. **Prioritization**. This stage involves interaction with stakeholders to discover the most important requirements.

6. **Requirements checking**. This stage is involves the checking of requirements, if they are complete, consistent and in accordance with what stakeholders really want from the system.

- **There are three different techniques for requirements elicitation and analysis:**

5.2.1 View-point Oriented Elicitation

- For any medium-sized or large system, there are usually different types of end-user.
- Many stakeholders have some kind of interest in the system requirements.
- Different viewpoints on a problem see the problem in different ways.
- Viewpoint-oriented approaches to requirements engineering recognize these different viewpoints and use them to structure and organize both elicitation process and the requirements themselves.

- A key strength of this technique is that it recognizes the existence of multiple perspectives and provides a framework for discovering conflicts in the requirements proposed by different stakeholders.
- Different methods have different ideas of what is meant by a viewpoint.
- A viewpoint may be considered as:

1. A data source or sink. In this case, viewpoints are responsible for producing or consuming data.

2. A representation framework. In this case, a viewpoint is considered to be a particular type of system model.

3. A receiver of services. In this case, viewpoints are external to the system and receive services from the system.

- The VORD (Viewpoint-Oriented Requirements Definition) method has been designed as a service-oriented framework for requirements elicitation and analysis.

5.2.2 Scenarios

- Scenarios can be particularly useful for adding detail to an outline requirements description.
- Different forms of scenarios have been developed and they provide different types of information at different levels of detail about the system.
- The scenario starts with an outline of the interaction and, during elicitation, details and added to create a complete description of that interaction.
- At its most general, a scenario may include:

1. a system state description at the beginning of the scenario;

2. a description of the normal flow of events in the scenario;

3. a description of what can go wrong and how this is handled;

4. information about other activities which might be going on the same time;

5. a description of the state of the system after completion of the scenario.

- A structured approaches may be used, such as:

Event Scenarios:

- Are used in VORD to document the system behavior when presented with the specific events.
- Each distinct interaction event may be documented with the separate event scenario.

Use-cases:

- Are a scenario-based technique for requirements elicitation which were first introduced in the Objectory methods.
- The set of represents all of the possible interactions that will be represented in the system requirements.
- A use case encapsulates a set of scenarios where each scenario is a single thread through the use case.

5.2.3 Ethnography

- is an observational technique that can be used to understand social an organizational requirements.
- The value of ethnography is that it helps discover implicit system requirements which reflect the actual rather than the formal processes in which people are involved.
- Ethnography is particularly effective at discovering two types of requirements:

1. Requirements that are derived from the way in which people actually work rather than the way in which process definitions say they ought to work.

2. Requirements that are derived from cooperation and awareness of other people's activities.

- Ethnography may be combined with prototyping.
- The ethnography informs the development of the prototype so the fewer prototype refinement cycles are required.
- The prototyping focuses the ethnography by identifying problems and questions which can then be discussed with the ethnographer.
- Ethnographic studies can reveal critical process details which are often missed by other requirements elicitation techniques.

5.3 Requirements Validation

- is concerned with showing that the requirements actually define the system which the customer wants.
- is important because errors in a requirements document can lead to extensive rework costs when they are subsequently discovered during development or after the system is in service.
- During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document.
- These checks include:

1. Validity Checks. A user may think that a system is needed to perform certain functions. Systems have diverse users with different needs and any set of requirements is inevitably a compromise across the user community.

2. Consistency Checks. Requirements in the document should not conflict. That is, there should not be contradictory or different descriptions of the same system function.

3. Completeness Checks. The requirements document should include requirements which define all functions and constraints intended by the system user.

4. Realism Checks. Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks also take account of the budget and schedule for the system development.

5. Verifiability. To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable.

- There are number of requirements validation techniques which can be used in conjunction or individually:

1. **Requirements reviews**. The requirements are analyzed systematically by the team of reviewers.

2. **Prototyping**. In this approach to validation, an executable model of the system is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

3. **Test-case generation**. Requirements should, ideally, be testable. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered.

4. **Automated consistency analysis**. If the requirements are expressed as a system model in a structured or formal notation then CASE tools may be used to check the consistency of the model.

5.3.1 Requirements Reviews

- is a manual process which involves multiple readers from both client and contractor staff checking the requirements document for anomalies and omissions.
- Can be informal or formal.
- Informal reviews involve contractors discussing requirements with as many system stakeholders as possible.
- In a formal requirements review, the development team should 'walk' the client through the system requirements, explaining the implications of each requirement.
- Conflicts, contradictions, errors and omissions in the requirements should be pointed out during the review and formally recorded.

5.4 Requirements Management

- The requirements for large software systems are always changing.
- Because the problem cannot be fully defined, the software requirements are bound to be incomplete.
- During the software process, the developer's understanding of the problem is constantly changing and these changes feed back to the requirements.
- Requirement management is the process of understanding and controlling changes to system requirements.
- The process of requirements management is carried out in conjunction with other requirements engineering processes.

5.4.1 Requirements management planning

- Planning is an essential first stage in the requirements management process.
- During the requirements management stage, you have to decide on:
- Requirements management needs some automated support and the CASE tools used should be chosen during the planning phase.
- For small systems, it may not be necessary to use specialized requirements management tools.
- However, for larger systems, more specialized tool support is required.

5.4.2 Requirement change management

- Should be applied to all proposed changes to the requirements.
- There are 3 principal stages to a change management process:

1. **Problem analysis and change specification**. During this stage, the problem or the change proposal is analyzed to check that it valid. A more specific requirements change proposal may then be made.

2. **Change analysis and costing**. The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once the analysis is completed, a decision is made whether or not to proceed with the requirements change.

3. **Change implementation**. The requirements document and, where necessary, the system design and implementation are modified.

- If a requirements change to a system is urgently required, there is always a temptation to make that change to the system and then retrospectively modify the requirements document.

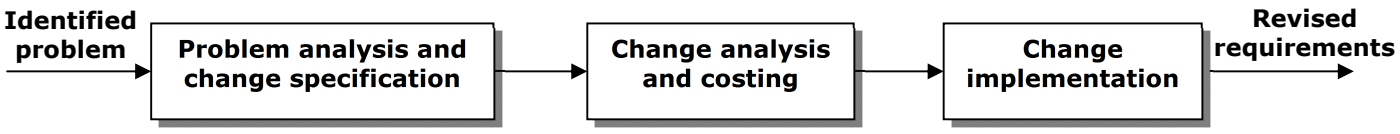

Fig: Requirements change management

CHAPTER 06 & 07

6.1 Requirement Analysis

- Is a software engineering tasks that bridges the gap between system level requirements engineering and software design.
- Allows the software engineer to refine the software allocation and build models of the data, functional and behavioral domains that will be treated by show.
- Provides software designer with a representation of information function, and behavior that can be translated to data, architectural, interface, and component level designs.
- Software requirements analysis can be divided into 5 areas of effort:

1. Problem recognition
2. Evaluation and synthesis
3. Modeling
4. Specification
5. Review

- Initially, the analyst studies the system specification and software project plan.
- It is important to understand software in a system context and to review the software scope that was used to generate planning estimates.
- Problem evaluation and solution synthesis is the next major area of effort for analysis.

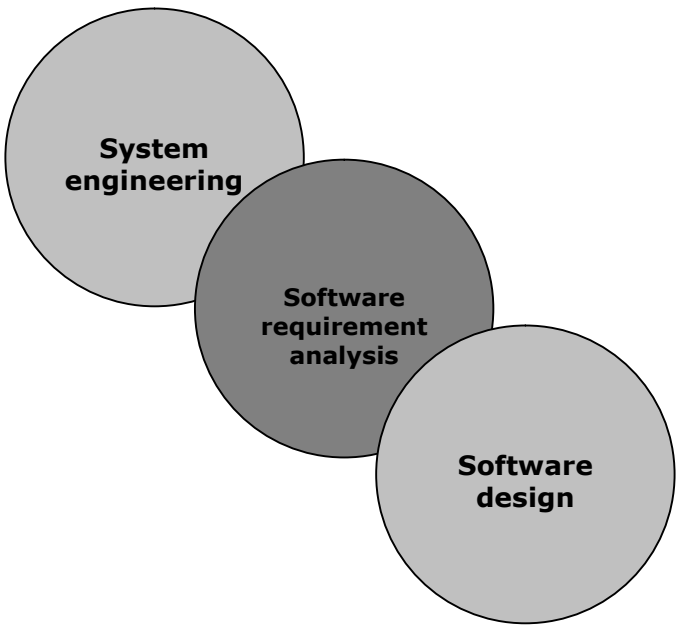

Fig: Analysis as a bridge between system engineering and software design.

6.1.1 Analysis Principles

- Numbers of analysis modeling methods have been developed.
- Each analysis method has a unique point of view.
- All analysis methods are related by a set of operational principles.

1. The information domain of a problem must be represented and understood.
2. The functions that the software is to perform must be defined.
3. The behavior of the software must be represented.
4. The models that depict information and behavior must be partitioned in a manner that uncovers detail in layered fashion.
5. The analysis process should move from essential information toward implementation detail.

- By applying these principles, the analyst approaches a problem systematically.

6.2 Viewpoint-Oriented Analysis (*Chapter 7*)

- For any medium-sized or large system, there are usually different types of end-users.
- Different people in an organization all have some kind of interest in system requirements.
- Some people have a more direct interest in the requirements of the system than others.
- For even a relatively simple system, there are many different viewpoints that should be considered.
- The most important viewpoints may be identified and used as a basis for structuring the requirements analysis.
- Viewpoint may be based on sources or sinks of data, different models of the system expressed using different notations or external interaction with the system.
- Different methods have different ideas of what is meant by a 'viewpoint'.
- A view point may be considered as:

1. *A data source or sink*. In this case, viewpoints are responsible for producing or consuming data. The analysis involves identifying all such viewpoints, identifying what data is produced or consumed and what processing is carried out.
2. *A representation framework*. In this case, a viewpoint is considered to be a particular type of system mode.
3. *A receiver of services*. In this case, viewpoints are external to the system and receive services from the system. Viewpoints may provide data for these services or control signals.

- Each of these models of a viewpoint has different strengths and weakness,
- Viewpoints as data sources or sinks and viewpoints as representations are particularly valuable for discovering detailed conflicts between requirements.
- They are less useful for structuring the requirements analysis process.
- The most effective viewpoint-oriented approach for analysis is based on external viewpoints.

- Viewpoints interact with the system by receiving services from it and providing data and control signals to the system.

- **The advantages of this type of viewpoint are:**

1. In the majority of interactive systems, it is natural to think of end-users as receivers of system services.

2. Because they are external to the system, viewpoints are a natural way to structure the requirements elicitation progress.

3. It is relatively easy to decide if some agent is or is not a valid viewpoint. If there is no interaction with the system, it is clearly not a viewpoint.

4. Viewpoints and services are a useful way of structuring non-functional requirements. Each service may have associated non-functional requirements. The same service, however, may have different non-functional requirements in different viewpoints.

CHAPTER 08

Overview of data modeling and flow diagram

SELF REVIEW – System Analysis & Design.

CHAPTER 09

- A prototype is an initial version of a software system which is used to demonstrate concepts, try out design options, and generally to find out more about the problem and its possible solutions.
- A software prototype supports two requirements engineering process activities:

    1. *Requirements elicitation*. System prototypes allow users to experiment to see how the system supports their work.

    2. *Requirements validation*. The prototype may reveal errors and omissions in the requirements which have been proposed.

- Prototyping can be used as a risk analysis and reduction technique.
- Developing system prototype may have following benefits:

    1. Misunderstanding between software developers and users may be identified as the system functions are demonstrated.

    2. Software development staff may find incomplete and/or inconsistent requirements as the prototype is developed.

    3. A working, albeit limited, system is available quickly to demonstrate the feasibility and usefulness of the application to management.

    4. The prototype may be used as a basis for writing the specification for a production-quality system.

- Once a prototype is available, it can also be used for other purposes as:

    1. *User training*. A prototype system can be used for training users before the final system has been delivered.

    2. *System testing*. Prototypes can run 'back-to-back' tests. The same test cases are submitted to the prototype and to the system under test. If both systems give the same result, the test case has not defected a fault.

- Prototyping usually increases costs in the early stages of the software but reduces later costs.

9.1 Prototyping in Software Process

- If the system is large and complex it is probably impossible to make assessments before the system is built and put into use.
- One way of tackling this difficulty is to use an evolutionary approach to systems development.
- Alternatively, a deliberate decision might be made to build a 'throw-away' prototype to help requirements analysis and validation.

9.1.1 Evolutionary prototyping

- is based on the idea of developing an initial implementation, exposing this to user comment an refining this through many stages until an adequate system has been developed.
- This approach to development was used initially for those systems which are difficult or impossible to specify.
- There are two main advantages to adopting this approach to software development:

1. Accelerated delivery of the system.

2. User engagement with the system.

9.1.2 Throw-away prototyping

- This approach extends the requirements analysis process with the intention of reducing overall life-cycle costs.
- The principal function of the prototype is to classify requirements and provide additional information for managers to assess process risk.
- After the evaluation, the prototype is thrown away.
- This approach is commonly used for hardware systems.
- Throw-away software prototype is not normally used for design validation but to help develop the system requirements.
- There are several problems to this approach:

1. Important features may have been left out of the prototype to simplify rapid implementation.

2. An implementation has no legal standing as a contract between customer and contractor.

3. Non-functional requirements such as those concerning reliability, robustness and safety cannot be adequately tested in a prototype implementation.

- Throw-away prototypes do not have to be executable software prototypes to be useful in the requirements engineering process.

9.2 Prototyping Techniques

- There are number of techniques which have been used for system prototyping. These include:

9.2.1 Executable specification language

- If a system specification is expressed in a formal, mathematical language, it may be possible to animate that specification to provide a system prototype.
- A number of executable formal specification languages have been developed.
- There are no additional costs in prototype development after the specification has been written.
- There are practical difficulties in applying this approach:

1. Graphical user interfaces cannot be prototyped using this technique.

2. Prototype development may not be particularly rapid.

3. The executable system is usually slow and inefficient. Users may get false impression of the system and compensate for this slowness during evaluation.

4. Executable specifications only test functional requirements. In many cases, the non-functional characteristics of the system are particularly important so the value of the prototype is limited.

- Some of these problems have been addressed by the developers of the functional languages which have been integrated with graphical user interface libraries and which allow rapid program development.
- A functional language is a formal language where the system is defined as a mathematical function.

9.2.2 Very high level languages

- are programming languages which include powerful data management facilities.
- These simplify program development because they reduce many problems of storage allocation and management.
- Examples of very high level languages are Lisp, Prolog, Smalltalk, APL, and SETL.
- Very high dynamic languages are normally not used for large system development because they need a large run-time support system.
- This run-time support system increases the storage needs and reduces the execution speeds of programs written in the language.
- One of the most powerful prototyping systems for interactive systems is the Smalltalk system.
- Smalltalk is an object-oriented programming language which is tightly integrated with its environment.
- There is never an ideal language for prototyping large systems as different parts of the system are so diverse.
- The advantage of mixed-language approach is that the most appropriate language for a logical part of the application can be chosen, thus speeding up prototype development.

9.2.3 Fourth-generation languages

- Evolutionary prototyping is now fairly commonly used for developing applications in the business system domain.
- There are many 4GLs and their use usually reduces the time needed for system development.
- These languages are successful because there is a great deal of commonality across data processing applications.
- At their simplest, 4GLs are database query languages such as SQL.

- These languages are used, often in conjunction with CASE tools, for the development of small to medium-sized systems.
- Using 4GLs for developing data processing systems is cost-effective in some cases, particularly for relatively small systems.
- Some CASE toolsets are closely integrated with 4GLs.
- Using such systems has the advantage that documentation is produced at the same time as the prototype system.
- 4GL-based development can be used either for evolutionary prototyping or may be used in conjunction with a method-based analysis where system models are used to generate the prototype.

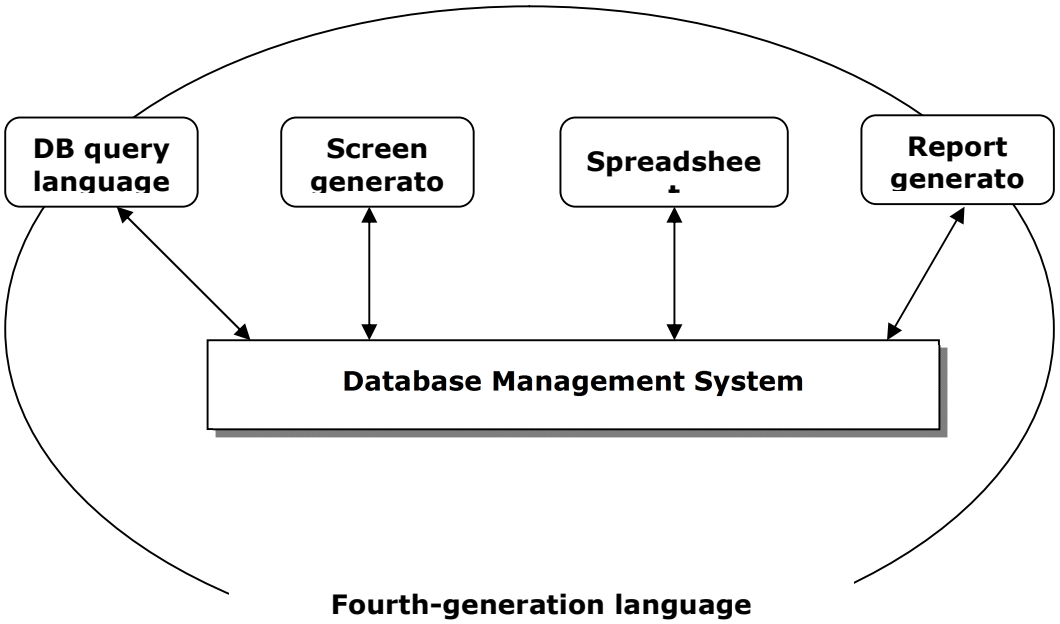

Fig: Fourth-generation languages

9.2.4 Composition of reusable components

- Prototyping with reusable components involves developing a system specification by taking account of what reusable components are available.
- These components are taken from a repository and put together to form the prototype system.
- This approach is most suitable for throw-away prototyping as the specification may not be exactly what is required.
- Prototyping using reusable components is often combined with other approaches using high level or fourth-generation languages.

- The success of Smalltalk and Lisp as prototyping languages is as much due to their reusable component libraries as to their inbuilt language facilities.

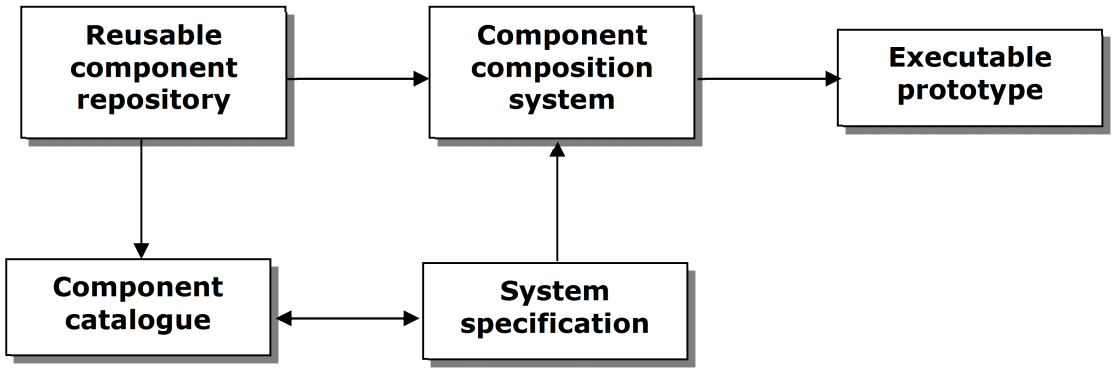

Fig: Reusable component composition

CHAPTER 10

10.1 Requirement Definition

- A software requirements definition is an abstract description of the services which the system should provide and constraints under which the system must operate.
- It should specify the external behavior of the system.
- The requirements should not be defined using an implementation model.
- The definition should be written in such a way that it is understandable by customers without knowledge of specialized notations.
- System requirements may be either functional or non-functional requirements:

1. **Functional requirements**. These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, functional requirements may also explicitly state what the system should not do.

2. **Non-functional requirements**. These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, standards and so on.

- The functional requirements definition of a system should be both complete and consistent.
- Completeness means that all services required by the user should be defined.
- Consistency means that requirements should not have contradictory definitions.
- There are three types of major problem with requirements definitions written in natural languages:

1. *Lack of clarity*. It is very difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.

2. *Requirements confusion*. Functional, non-functional requirements, system goals and design information may not be clearly distinguished.

3. *Requirements amalgamation*. Several different requirements may be expressed together as a single requirement

- Some organizations try to produce a single specification to act as both a requirements definition and a requirements specification.
- When a requirements definition is combined with a specification there is often confusion between concepts and details.

10.2 Requirements Specification

- Adds further information to the requirements definition.
- is usually presented with the system models developed during requirements analysis.
- It should include all necessary information about what the system must do and all constraints on its operation.
- Natural languages are often used to write requirements specifications.
- The natural language specification is not a particularly good basis for either a design or a contract between customer and system developer.

- **There are several reasons for this:**

    1. Natural language understanding relies on the specification readers and writers using the same words for the same concept. This leads to misunderstanding because of the inherent ambiguity of natural language words.

    2. A natural language requirements specification is over-flexible. You can say the same thing in completely different ways. It is up to the reader to find out when requirements are the same and when they are distinct.

    3. Requirements are not partitioned effectively by the language itself. It is difficult to find the related requirements.

- There are various alternatives to the use of natural language which add structure to the specification and which should reduce ambiguity. These are:

    1. *Structured natural language*. This approach depends on defining standard forms or templates to express the requirements specification.

    2. *Design description languages*. This approach relies on using a language which is like a programming language but more abstract features to specify the requirements by defining an operational model of the system.

    3. *Requirements specification languages*. Various special-purpose languages have been designed to express software requirements.

    4. *Graphical notations*. The best known graphical notation requirement is SADT.

    5. *Mathematical specifications*. These are notations based on a formal mathematical concept such as finite-state machines.

- When requirements specifications are written, it is important that related requirements should be cross-referenced.
- Traceability is the property of a requirements specification which reflects the ease of finding related requirements.
- There are some simple methods of traceability that may be applied to any requirements definition or specification:

    1. All requirements should be assigned a unique number.

    2. Requirements should explicitly identify related requirements by referring to their number.

    3. Each requirement document should contain a cross-reference matrix showing related requirements.

10.2.1 Structured language specifications

- is a restricted form of natural language for requirements specifications.
- Structured language notations may limit the terminology used and may use templates to specify system requirements.
- The advantage of this approach is that it maintains most of the expressiveness and understandability of natural language.
- It does, however, ensure that some degree of uniformity is imposed on the specification.
- Using formatted specifications removes some of the problem of natural language specification in that there is less variability in the specification and requirements are partitioned more effectively.

10.2.2 Requirements specification using a PDL

- To counter the inherent ambiguities in natural language specification, it is possible to describe requirements operationally using a program description language or PDL.
- The advantage of using a PDL is that it may be checked syntactically and semantically by the software tools.
- Requirements omissions and inconsistencies may be inferred in two situations:

1. When an operation is specified as a sequence of simpler actions and the order of execution is important. Descriptions of such sequences in natural language are sometimes confusing, particularly if nested conditions and loops are involved.

2. When hardware and software interfaces have to be specified. The PDL usually allows more detail about interface objects and types to be specified.

- If the reader of a requirements specification is familiar with the PDL used, specifying the requirements in this way can make them less ambiguous and easier to understand.
- There are also few disadvantages of this approach:

1. The language used to write the specification may not be sufficiently expressive to describe application domain concepts in an understandable way.

2. Customer staffs who are unfamiliar with programming languages may be intimidated by the notation.

3. The specification will be seen as an abstract design rather than a model to help the user understand the system.

4. Design decisions may be made too early in the software process and this can restrict the freedom of designers to meet other, non-functional, system requirements.

CHAPTER 11

11.1 Introduction to software Design

- Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used.
- Software design is the first of three technical activities: design, code generation, and test – that are required to build and verify the software.
- Using one of the numbers of design methods, the design task produce a data design, an architectural design, an interface design, and a component design.
- The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software.
- The architectural design defines the relationship between major structural elements of the software.
- The interface design describes how the software communicates within itself, with systems that incorporate with it and with humans who use it.
- The component level design transforms structural elements of the software architecture into a procedural description of the software components.
- The importance of software design can be state with a single word: **"QUALITY"**.

11.2 Software Design Process

- Software design is an iterative process through which requirements are translated into a blue-print for construction the software.
- As design iterations occur, subsequent refinements leads to design representations at much lower level of abstractions.

11.2.1 Design and Software quality

- Throughout the design process, the quality of evolving design is assessed with a series of formal technical reviews or design walkthroughs.
- McGlaugwin suggests three characteristics that serve as a guide for the evaluation of a good design.

1. The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customers.

2. The design must be readable, understandable guide for those generate code and for those who test and subsequently support the software.

3. The design should provide a complete picture of software, addressing the data, functional and behavioral domains from an implementation perspective.

- Each of these characteristics is actually a goal of the design process.
- The goal of the design process is achieved by following guidelines:

1. A design should exhibit an architectural design that –

2. A design should be modular, that is, the software should be logically partitioned into elements that perform specific functions and sub-functions.

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 objects to be implemented and are drawn from recognizable data patterns.

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 between modules and with the external environment.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirement analysis.

11.3 Design Principles

- Software design is both a process and a model.
- The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.
- The design model is equivalent of an architect's plans for a house.
- The design model that is created for software provides a variety of different views of the computer software.
- Basic design principles enable the software engineer to navigate the design process.
- Davis suggests a set of principles for the software design:

1. The design process should not suffer from "tunnel vision". A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concept presented.

2. The design should be traceable to the analysis model. Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.

3. The design should not reinvent the wheel. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention.

4. The design should minimize the intellectual distance between the software and the problem as it exists in the real world.

5. The design should exhibit uniformity and integration. A design is uniform if it appears that one person developed the entire thing. A design is integrated if care is taken in defining interfaces between design components.

6. The design should be constructed to accommodate change.

7. The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.

8. Design is not coding and coding is not design.

9. The design should be assessed for quality as it is being created, not after the fact.

10. The design should be reviewed to minimize conceptual errors. A design team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax of the design.

- When the design principles are properly applied, the software engineer creates a design that exhibits both external and internal quality.

11.4 Design Concepts

- A set of fundamental software design concepts has evolved over the past four decades.
- Each provides the software designer with a foundation from which more sophisticated design methods can be applied.
- Each helps the software engineer to answer the following questions:

1. What criteria can be used to partition software into individual components?

2. How function or data is structure detail separated from a conceptual representation to the software?

3. What uniform criteria define the technical quality of a software design?

- The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right.

11.4.1 Abstraction

- is one fundamental ways that we as human cope with complexity.
- When we consider a modular solution of any problem, many levels of abstraction can be posed.
- At the highest level of abstraction, a more procedural orientation is taken.
- Each step in software process is a refinement in a level of abstraction of the software solution.
- As we move through different levels of abstractions, we work to create procedural and data abstractions.

11.4.2 Refinement

- is actually a process of elaboration.
- Stepwise refinement is a top-down design strategy originally proposed by Niklous Wirth.
- A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached.

- Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement occurs.

11.4.3 Modularity

- Modularity is the single attribute of the software that allows a program to be intellectually manageable.
- The system may be designed modularly, even if its implementation must be “monolithic”.

11.4.4 Software Architecture

- Software architecture alludes to “overall structure of the software and the ways in which that structure provides conceptual integrity for a system.”
- One goal of the software design is derive an architectural rendering of a system.
- A set of architectural patterns enable a software engineer to reuse design-level concepts.

CHAPTER 12

The Software Design Process

a. Design Strategy, Concepts

b. Design and Software quality

CHAPTER 13

13.1 Software Architecture

- Software is divided into modules first.
- Software systems have had architectures, and programmers have been responsible for the interactions among the modules and global properties of the assemblage.
- Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

13.1.1 What is Architecture

- According to Bass, Clements, and Kazman "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."
- The architecture is not the operational software.
- It is a representation that enables a software engineer to:

    1. analyze the effectiveness of the design in meeting its stated requirements,

    2. consider architectural alternatives at a stage when making design changes is still relatively easy, and

    3. reducing the risks associated with the construction of the software.

- The software architecture considers two levels of the design pyramid – data design and architectural design.
- Data design enables us to represent the data component of the architecture.
- Architectural design focuses on the representation of the structure of software components, their properties and interactions.

13.1.2 Why is Architecture important?

- The three key reasons why software architecture is important are:

    1. Representations of software architecture are an enable for communication between all parties (stakeholders) interested in the development of a computer-based system.

    2. The architectural highlights early decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

    3. Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together."

- The architectural model provides a Gestalt view of a system, allowing the software engineer to examine is as a whole.
- The architectural design model and the architectural patterns contained within it are transferable.

13.2 Architectural styles

- The software that is built for computer-based systems exhibits one of many architectural styles:
- Each style describes a system category that encompasses:
    1. a set of components that perform a function required by a system.
    2. a set of connectors that enable "communication, co-ordination, and co-operation" among components.
    3. constraints that define how components can integrated to form the system, and
    4. semantic models that enable a designer to understand the overall properties of the system by analyzing the known properties of its constituent parts.

13.2.1 A Brief Taxonomy of Styles and Patterns

- although millions of computer-based systems have been created over the past 50 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

13.2.1.1 Data-Centered Architecture:

- A data store resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Promotes integrability i.e. existing components can be changed and new client components can be added to the architecture without concern about other clients.

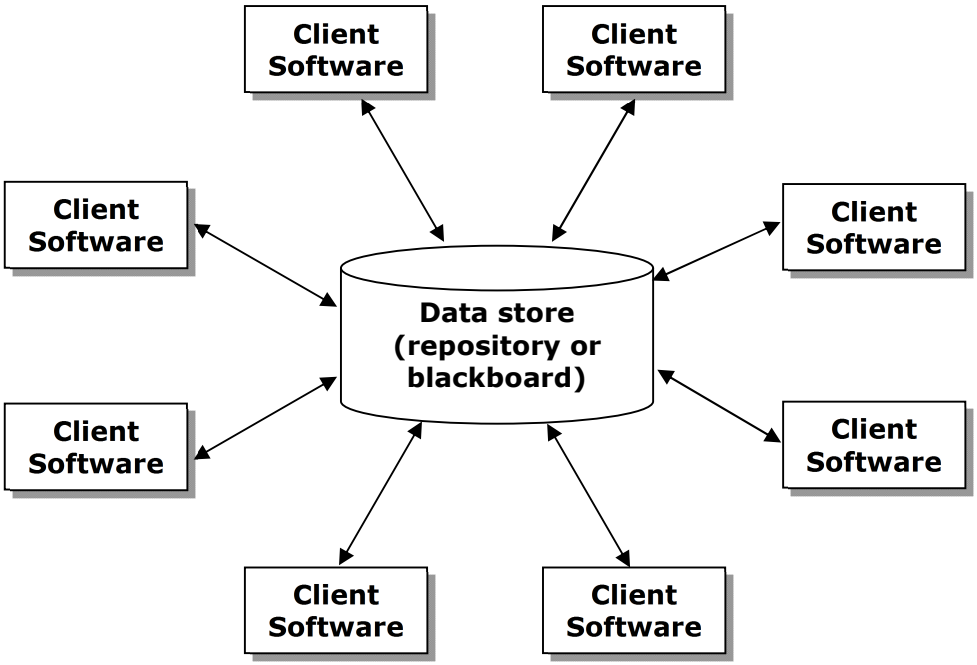

Fig:- Data-Centered Architecture

13.2.1.2 Data-Flow Architecture:

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A pipe and filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- Each filter works independently, and does not require knowledge of the working of its neighboring filters.
- If the data flow degenerates into a single line of transforms, it is termed as batch sequential.

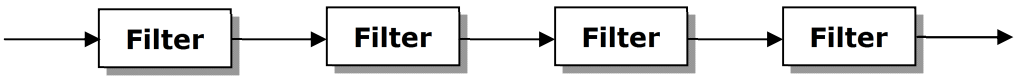

Fig:- Batch Sequential

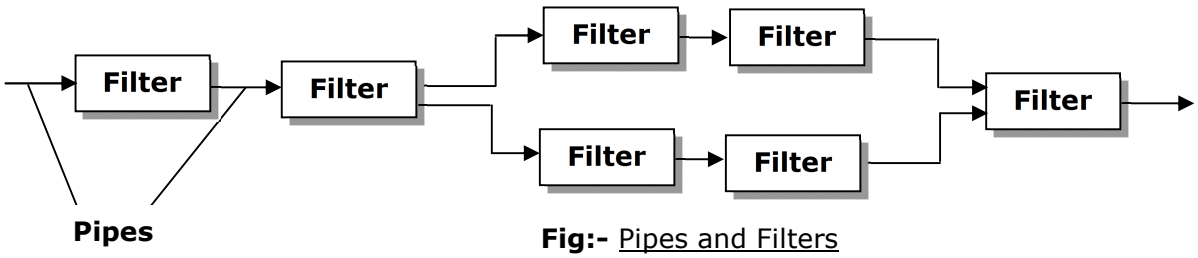

Fig:- Pipes and Filters

13.2.1.3 Call and Return Architecture:

- This architecture style enables a software designer to achieve a program structure that is relatively easy to modify and scale.
- A number of sub-styles exist within this category:

a. Main program/sub-program architecture:

This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke other components.

b. Remote procedure call architecture:

The components of a main-program/sub-program architecture are distributed across a multiple computers in a network.

13.2.1.4 Object-Oriented Architecture:

- The component of a system encapsulates data and the operations that must be applied to manipulate data.
- Communication and co-ordination between components is accomplished via message passing.

13.2.1.5 Layered Architecture:

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At inner layer, components perform operating system interfacing.
- Intermediate layer provide utility services and application software functions.

CHAPTER 14

14.1 Software Testing Fundamentals

- Testing presents an interesting anomaly for the software engineers.
- The engineer creates a series of test cases that are intended to "demolish" the software that has been built.
- Testing is the one step in software process that could be reviewed as destructive rather than constructive.
- Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered.

14.1.1 Testing Objectives

- There are number of rules stated by Glen Myers, that can serve as testing objectives:

1. Testing is the process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet-undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

- If testing is conducted successfully (according to the objectives stated), it will uncover errors in the software.
- Testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.
- Testing cannot show the absence of errors and defects, it can show only that software errors and defects are present.

14.1.2 Testing Principles

- Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing.
- Testing principles suggested by Davis are:

1. **All tests should be traceable to customer requirements**. As the objective of software testing is to overcome errors. It allows that the most severe defects (from customer's point of view) are those that cause the program to fail to meet its requirements.
2. **Tests should be planned long before testing begins**. Test planning can begin as soon as the requirements model is complete.
3. **The Pareto principle applies to software testing**. The Pareto principle implies that 80% of errors uncovered during testing will likely be traceable to 20% of all program components.
4. **Testing should begin "in the small" and progress toward testing "in the large"**. The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
5. **Exhaustive testing is not possible.** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing.
6. **To be most effective, testing should be conducted by an independent third party.** By most effective, mean testing that has the highest probability of finding errors. The software engineer who created the system is not the best person to conduct all tests for the software.

14.1.3 Testability

- Software testability is simply how easily a computer program can be tested.
- There are certainly metrics that could be used to measure testability in most of its aspects.
- Sometimes, testability is used to mean how adequately a particular set of tests will cover the product.
- "Testability" occurs as a result of good design. Data design, architecture, interfaces, and component-level detail can either facilitate testing or make it difficult.
- Kaner, Falk, and Nguyen suggest the following attributes of a "good" design.

1. A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
2. A good test is not redundant. There is not point in conducting a test that has the same purpose as another test.
3. A good test should be "best of breed". In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.
4. A good test should be neither too simple nor too complex. In general, each test should be executed separately.

14.2 Test Case Design

- The design of tests for software and other engineered products can be as challenging as the initial design of the product itself.
- A rich variety of test case design methods have evolved for software.
- This method provides the developer with a systematic approach of testing.
- More important, methods provide a mechanism that can help to ensure the components of tests and provide the highest likelihood for uncovering errors in software.
- Any engineered products can be tested in two ways:

1. Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.
2. Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh", that is, internal operations are performed according to specifications and internal components have been adequately exercised.

- The first approach is called black box testing, and the second, white box testing.
- Black box test are used to demonstrate the software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information is maintained.
- White box test of software is predicated on close examination of procedural detail.
- White box test can be designed only after a component–level design exists, the logical details of the program must be available.

14.3 Black Box Testing (*Chapter 15*)

- Also called behavioral testing.
- Focuses on the functional requirements of the software.
- It is not alternative of white box testing.
- Black box testing attempts to find errors in the following categories:

1. Incorrect or missing functions.
2. Interface errors.
3. Errors in data structures or external database access.
4. Behavior or performance errors.
5. Initialization and termination errors.

- It is applied during later stages of testing.
- Because black box testing purposely disregards control structure, attention is focused on the information domain.
- Tests are designed to answer the following questions:

1. How is functional validity tested?
2. How is system behavior and performance tested?
3. What classes of input will make good test cases?
4. Is the system particularly sensitive to certain input values?
5. How are the boundaries of a data class isolated?
6. What data rates and data volume can the system tolerate?
7. What effect will specific combinations of data have on system operation?

- By applying black box testing technique, we derive a set of test cases that satisfy the following criteria:

1. Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing.

2. Test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

14.3.1 Graph-Based Testing Methods

- The first step in black box testing is to understand the objects that are modeled in software and relationship that connect the objects.
- Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another".
- Stated another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.
- To accomplish these steps, the software engineer begins by creating a graph-a collection of nodes that represent objects, links that represents the relationships between objects, node weights that describe some characteristics of a link.

- Nodes are represented as circles connected by links that take a number of different forms.
- A directed link indicates that relationship moves in only one direction.
- A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions.
- Parallel links are used when a number of different relationships are established between graph nodes.

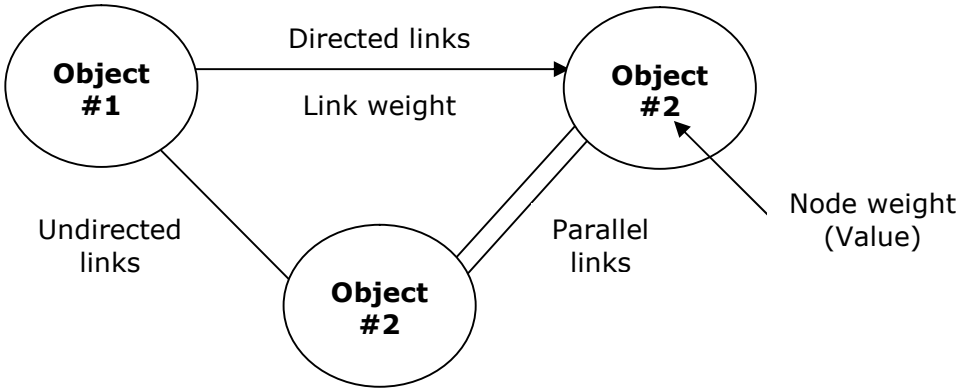

Fig: - Graph Notation

- Graph-based testing begins with the definition of all nodes and node weights i.e. objects and attributes are identified.
- The data model can be used as a starting point, but it is important to note that many nodes may be program objects, not explicitly represented in the data model.
- To provide an indication of the start and stop points for the graph, it is useful to define entry and exit nodes.
- Once nodes have been identified, links and link weights should be established.
- In general, links should be named.
- The graph model may consist loops.

14.3.2 Equivalence Partitioning

- is a black-box testing method that divides the input domain of a program into classes of data from which the test cases can be derived.
- It strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.
- Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.
- An input condition is either a specific numeric value, a range of values, a set of related values, or Boolean condition.
- Equivalence classes may be defined according to the following guidelines:

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

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

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

14.3.3 Boundary Value Analysis

- is used when a great number of errors tends to occur at the boundaries of input domain rather than in the "center".
- It leads to a selection of test cases that exercise boundary values.
- If a test case design technique that complements equivalence partitioning.
- Leads to the selection of test case at the "edges" of the class.
- Guidelines for boundary value analysis are similar in many respects to those provided for equivalence partitioning:

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

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

    3. Apply guidelines 1 and 2 to output conditions.

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

- By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

14.3.4 Comparison Testing

- There are some situations in which the reliability of software is absolutely critical.
- In such applications redundant hardware and software are often used to minimize the possibility of errors.
- When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specifications.
- In such situations, each version can be tested with the same test data to ensure that all provide the identical output.
- Then all versions are executed in parallel with the real time comparison of results to ensure consistency.
- These independent versions form basis of a black-box testing technique called comparison testing or back-to-back testing.
- If the output of each version is the same, it is assumed that all implementations are correct.
- If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference.
- Comparison testing is not foolproof.

14.3.5 Orthogonal Array Testing

- There are many applications in which the input domain is relatively limited.
- In such scenario, it is possible to consider every input permutation and exhaustively test processing of input domain.
- Orthogonal array testing can be applied to problems in which the input is relatively small but too large to accommodate exhaustive testing.
- This method is particularly useful in finding errors associated with region faults-an error category associated with faulty logic within a software component.

14.4 White Box Testing (*Chapter 16*)

- White box testing sometimes called glass-box testing; is a test case design method that uses the control structure of the procedural design to derive test cases.
- Using white box testing methods, the software engineer can derive test cases that –

1. Guarantee that all independent paths within a module have been exercised at least once.
2. Exercise all logical decisions on their true and false sides.
3. Execute all loops at their boundaries and within their operational bounds, and
4. Exercise internal data structures to ensure their validity.

Why White-box testing?

1. Logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
2. We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis.
3. Typographical errors are random; when a program is transferred into programming language source code, it is likely that some typing errors will occur.

14.4.1 Basic Path Testing

- Basic path testing is a white box testing technique first proposed by Tom McCabe.
- It enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

14.4.1.1 Flow Graph Notations:

- Before basis path method can be introduced, a simple notation for the representation of control flow, called flow graph must be introduced.
- It enables you to trace program paths more readily.

- Have to draw a flow graph when the logical control structure of a module is complex.

14.4.1.2 Cyclomatic Complexity:

- is a software metric that provides a quantitative measure of the logical complexity of a program.
- The value computed for the cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper-bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.
- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
- Cyclomatic complexity is a useful metric for predicting those modules that are likely to be error prone.
- It can be used for test planning as well as test case design.
- Cyclomatic complexity is computed in one of three ways:

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

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

   V(G) = E – N + 2

   Where E is the number of flow graph edges, N is the number of flow graph nodes

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

   V(G) = P + 1

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

14.4.1.3 Graph Matrices

- is a square matrix whose size is equal to the number of nodes on the flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- The graph matrix is nothing but a tabular representation of a flow graph.

14.4.2 Control Structure Testing

- It increases the quality of the white-box testing.

14.4.2.1 Condition Testing:

- is a test case design method that exercise the logical conditions contained in a program module.
- A simple condition is Boolean variable or a relational expression, possibly preceded with one NOT operator.
- If a condition is incorrect, then at least 1 component of the condition is incorrect, and types of errors include:

1. Boolean operator error.
2. Boolean variable error.

3. Boolean parenthesis error.
4. Relational operator error.
5. Arithmetic expression error.

- The purpose of condition error is to detect not only errors, in the conditions of the program but also other error in the program.

14.4.2.2 Data Flow Testing:

- This method selects test paths of a program according to the locations of definitions and uses of variables in the program.
- It is unrealistic to assume that data flow testing will be used extensively when testing a large system.
- are useful for selecting test paths of a program containing nested if and loop statements.

14.4.2.3 Loop Testing:

- Focuses exclusively on the validity of loop constructs.
- Four different classes of loops can be defined:

1. Simple Loops
2. Concatenated Loops
3. Nested Loops, and
4. Constructed Loops

CHAPTER 17

17.1 Software Testing Strategies

- A strategy for software testing integrates software test case design methods into a well-planned series of steps that result in successful construction of software.
- The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, how much effort, time an resources will be required.
- A software testing strategy should be:

1. Flexible enough to promote a customized testing approach.

2. Rigid enough to promote reasonable planning and management tracking all the project progresses.

- It is developed by the project manager, software engineers and testing specialists.

17.1.1 Strategic approach to Software Testing

- Testing is a set of activities that can be planned in advance and conducted systematically.
- A number of software testing strategies have been proposed, and all provide the software developer with a template for testing and all have the following characteristics:

1. Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.

CHAPTER 18

18.1 Quality concepts

- The American Heritage Dictionary defines Quality as "a character or attribute of something".
- Quality of design refers to the characteristics that designers specify for an item.
- Quality of conformance is the degree to which the design specifications are followed during manufacturing.
- The greater degree of conformance, the higher is the level of quality of conformance.
- In software development, quality of design encompasses requirements, specifications, and the design of the system.
- A product's quality is a function of how much it changes the world for the better.

18.1.1 Quality Control

- Quality control involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it.
- It includes a feedback loop to the process that created the work.
- Quality control activities may be fully automated, entirely manual, or combination of automated tools and human interaction.
- A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process.

18.1.2 Quality Assurance

- Quality assurance consists of auditing and reporting functions of management.
- The goal of quality assurance is to provide management with the data necessary to be informed about product quality.
- If the data provided through the quality assurance identify the problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

18.1.3 Cost of Quality

- The cost of quality includes all costs incurred in the pursuit of quality or in performing quality related activities.
- Quality cost may be divided into costs associated with prevention, appraisal, and failure.

18.2 Software Quality Assurance

- High quality software is an important goal.
- How do we define quality?
- Many definitions of software quality have been proposed in the literature.
- Software quality is defined as:

  "Conformance to explicitly stated functions and performance requirements explicitly documented development standards and implicit characteristics that are expected to all professionally developed software".
- The definition serves to emphasize the three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered.
3. A set of implicit requirements often goes unmentioned. If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

- The history of quality assurance in software development parallels the history of quality in hardware manufacturing.
- The people who perform Software Quality Assurance (SQA) must look at the software from the customer's point of view.

18.2.1 SQA Activities

- SQA activities is composed of variety of tasks associated with two constituencies:
  1. The software engineers, who do technical work, and
  2. An SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.
- Software engineers address quality by applying solid technical methods and measures, conducting formal technical reviews, and performing well-planned software testing.
- The task of SQA group is to assist the software team in achieving a high quality end product.

CHAPTER 19

Software Process and Projects Metrics

a. Measures, Metrics and Indicators

b. Metrics in the process and projects

c. Software Measurement

d. Metrics for software quality