Software Engineering Notes

1. The Evolving role of Software

Today, software takes on a dual role. *It is a* ***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, *by a network of computers* that are accessible by local hardware. Whether it resides within a mobile phone or operates inside a mainframe computer, ***software is information transformer— producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources***.

As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time—*information.* It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context.

Software manages business information to enhance competitiveness;
Software provides a gateway to worldwide information networks (e.g., the Internet).
Software provides the means for acquiring information in all of its forms.
Software role has undergone significant change over the last half-century.
Software industry has become a dominant factor industrialized world.

Software Crisis:

- Software crisis, the symptoms of the problem of engineering the software, began to enforce the practitioners to look into more disciplined software engineering approaches for software development.

- The software industry has progressed from the desktop PC to network-based computing to service-oriented computing nowadays.

- The development of programs and software has become complex with increasing requirements of users, technological advancements, and computer awareness among people.

- *Software crisis symptoms*
  - complexity,
  - hardware versus software cost,
  - delay and costliness,
  - poor quality,
  - unmanageable nature,
  - irresponsibility,
  - lack of planning and management practices,
  - Change, maintenance and migration,
  - etc.

What is Software Engineering?

Watts S. Humphrey is the father of Software Engineering, created the Software Process Program at Carnegie Mellon University's Institute (SEI) in the 1980s, and served as its director from 1986 through the early 1990s. This program was designed to help participants understand and manage the software development process.

- *The solution to these software crises is to introduce systematic software engineering practices for systematic software development, maintenance, operation, retirement, planning, and management of software.*

Software engineering is the practices (performs) for systematic software development, maintenance, operation,planning, and management of software.

• The systematic means the methodological and pragmatic way of development, operationand maintenance of software.

• Systematic development of software helps to understand problems and satisfy the clientneeds.

• Development means the construction of software through a series of activities, i.e., analysis, design, coding, testing, and deployment.

• Maintenance is required due to the existence of errors and faults, modification of existingfeatures, addition of new features, and technological advancements.

• Operational software must be correct, efficient, understandable, and usable for work at the client site.

• IEEE defines
The systematic approach to the development, operation, maintenance, and retirement of software.

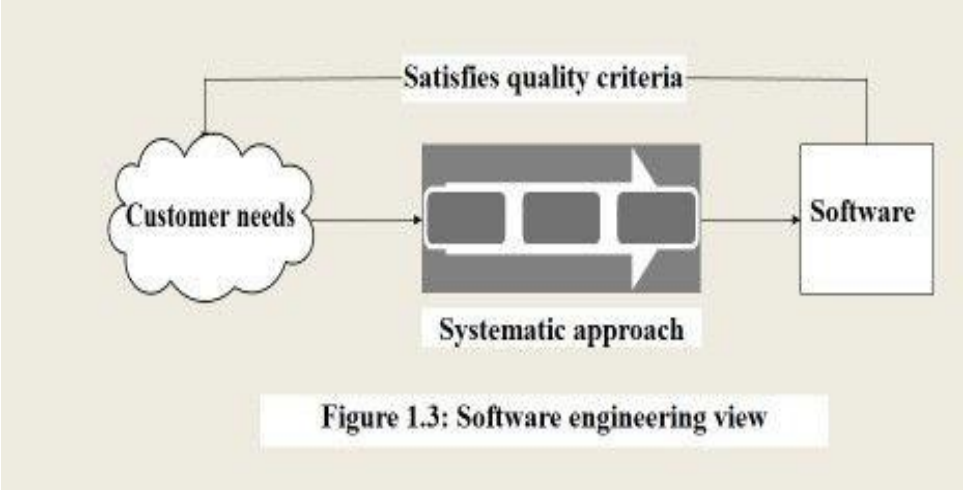


Figure 1.3: Software engineering view

Software Engineering -

• *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*

. It follows that design becomes a unique activity.

• *It follows that software should demonstrate high quality.*

• *It follows that software should be maintainable, software in all of its forms and across all of its application domains should be engineered.*

Engineering Discipline:

- Engineering is a disciplined approach with some organized steps in a managed way to construction, operation, and maintenance of software.
- Engineering of a product goes through a series of stages, i.e., planning, analysis and specification, design, construction, testing, documentation, and deployment.
- The disciplined approach may lead to better results.
- The general stages for engineering the software include feasibility study and preliminary investigation, requirement analysis and specification, design, coding,testing, deployment, operation, and maintenance.
- Development means the construction of software through a series of activities, i.e., analysis, design, coding, testing, and deployment.
- Maintenance is required due to the existence of errors and faults, modification of existing features, addition of new features, and technological advancements.
- Operational software must be correct, efficient, understandable, and usable for work at the client site.

- **IEEE defines**
  - ***S.E is the systematic approach to the development, operation,maintenance, and retirement of software.***

1.1 Defining Software

Software is:
(1) **instructions** (computer programs) that when executed provide desired features, function, and performance;
(2) **data structures** that enable the programs to adequately manipulate information, and
(3) **descriptive information** in both hard copy and virtual forms that describes the operation and use of the programs.

Software characteristics:

Software is a logical rather than a physical system element. Therefore, 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 manufacturing, 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.

Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different.

Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. Software doesn't "wear out."

Stated simply, the hardware begins to wear out. Software doesn't "wear out." But Software does deteriorate (i.e., become progressively worse)!

3. Although the industry is moving toward component-based construction, most software continues to be custom built.

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems.

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.

In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

- Software has logical properties rather than physical.
- Software is mobile to change i.e., software is flexible.
- Software is produced in an engineering manner rather than in classical sense.
- Software becomes obsolete but does not wear out or die.
- Software has a certain operating environment, end user, and customer.
- Software development is a labor-intensive task

1.2 The Changing Nature of Software:

(Application Domains i.e., Types of Software or Categories of Software)

Today, different broad categories of computer software present continuing challengesfor software engineers:

i. **System software**—a collection of programs written to service other programs.

System software processes complex, but determinate information structures.
e.g., compilers, editors, and file management utilities

Systems applications process largely indeterminate data.
(e.g., operating system components, drivers, networking software, telecommunications processors)

ii. **Application software**—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.
e.g., point-of-sale transaction processing, real-time manufacturing process control.

iii. **Engineering/scientific software**—is a special software to implement Engineering and Scientific applications. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

iv. **Embedded software**—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself.

e.g., key pad control for a microwave oven.

Provide significant function and control capability

e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems.

v. **Product-line software**—designed to provide a specific capability for use by many different customers.

e.g., inventory control products, word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications.

vi. **Web applications**—called "WebApps," this network-centric software category spans a wide array of applications. WebApps are linked with hypertext files. WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

vii. **Artificial intelligence software**— makes use of nonnumeric algorithms to solve complex problems of straightforward analysis.

Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

viii. **Open-world computing**—Software related to wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

ix. **Net sourcing**—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

x. **Open source**—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development.

The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

And other soft wares are personal, mobile, ubigititous and business

1.3 Legacy Software

Older programs —often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

> Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The maintenance of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

> Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.
> Legacy systems sometimes have inextensible designs, complicated code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long.

> legacy systems often evolve for one or more of the following reasons:

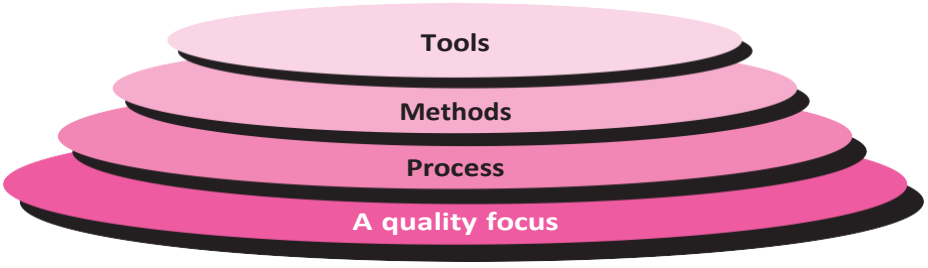
- The software must be adapted to meet the needs of new computing environments or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with other more modern systems or databases.
- The software must be re-architected to make it viable within a network environment.

2. Software Engineering- A Layered Technology:

Software engineering is a layered technology. Referring to Figure, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engi- neering. The bedrock that supports software engineering is a quality focus.

The foundation for software engineering is the *process* layer. The 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

FIGURE 1.3
Software engineering layers



that must be established for effective delivery of software engineering technology. A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created.

The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical how-to's for building soft- ware. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and sup- port. 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 information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

3. Softare Myths

- Software myths propagate false beliefs and confusion in the minds of management, users and developers.
- Software myths –false beliefs and erroneous beliefs about software and the process used to build it i.e., these are the false beliefs about the software development.
- Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike.
- Software Myths are three types

1. Managers Myths
2. Customers Myths (User Myths)
3. Practitioners Myths (Developer Myths)

i. **Management myths.** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: The book of standards may very well exist, but is it used?

Are software practitioners aware of its existence?

Does it reflect modern software engineering practice?

Is it complete?

Is it adaptable?

Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: *If we get behind schedule, we can add more programmers and catch up.*
Reality: Software development is not a mechanistic process like manufacturing. In the words of Mr. Brooks "adding people to a late software project makes it later."

At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

Myth: *If I decide to outsource the software project to a third party, I can just relax.*
Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Myth: My people (Developers) have modern software development tools, after all, we buy them the newest computers.
Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effective

ii. **Customer myths,** A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

Myth: *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*
Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and Continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*
Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can requires additional resources and major design modification.

iii. **Practitioner's myths.** Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: *Once we write the program and get it to work, our job is done.*
Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: *Until I get the program "running" I have no way of evaluate its quality.*
Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review.* Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: *The only deliverable work product for a successful project is the working program.*
Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *Software engineering will make us creates huge and unnecessary documentation and will always slow down software development.*
Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times. There by controlling its impact and cost.

4. The Software Process

- A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created.
- An *activity* attempts to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project or the complexity of the effort with which software engineering is to be applied.
- An *action* (e.g., architectural design) includes a set of tasks that produce a major work product (e.g., gathering the requirements or an architectural design model).
- A *task* focuses on a small, but well-defined objective (e.g., conducting a review of requirement analysis) that produces a real outcome.
- A Work Product an output of an action. They are individually estimated, budgeted, assigned, executed, measured and controlled.

- The goal of s/w process is always to ***deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation*** and those who will use it.

- A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of ***framework activities*** that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses (consists of) a set of ***umbrella activities*** that are applicable across the entire software process.

A generic process framework for software engineering encompasses five activities:

i. **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders). The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
i. **Planning.** A Planning activity creates a "map" defines the work by describing the tasks, risks and resources, work products and work schedule. The map—called a *software project plan*— defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
ii. **Modeling.** Modeling creates a "sketch" or "blue print" of the thing so that we will understand the requirements, how the constituent parts fit together, and many other characteristics.

   If required, we refine the sketch into greater and greater detail in an effort to better understand the problem and how we are going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
iv. **Construction.** This activity combines code generation (either manual or automated) and the testing that is required uncovering errors in the code.

v. **Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but theframework activities remain the same.

Software engineering process framework activities are balanced by a number of *umbrella activities.* In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.

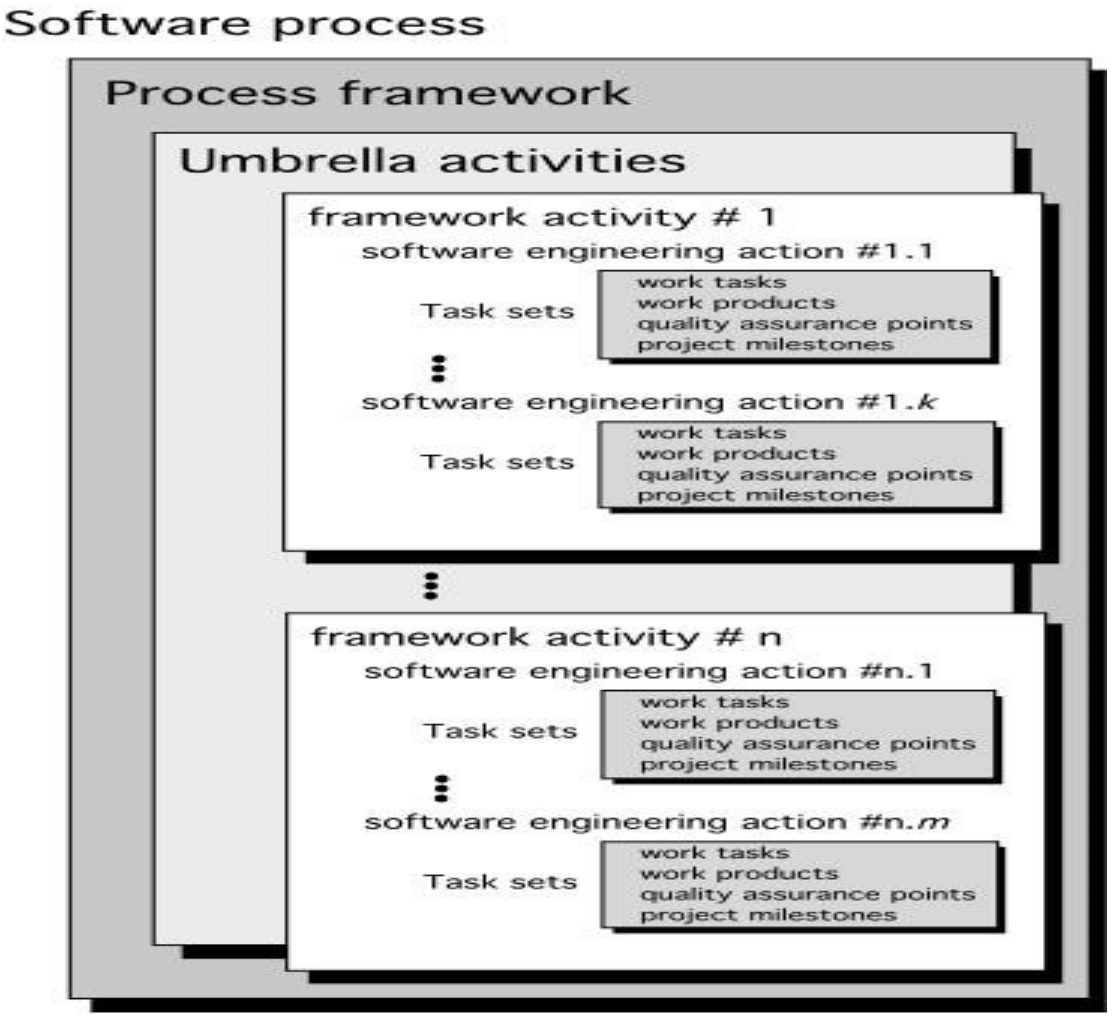
Typical umbrella activities include:

1. Software project tracking and control—allows the software team to review progress against the project plan and take any necessary action to maintain the schedule.
2. Risk management—reviews the risks that may affect the outcome of the project or the qualityof the product.
3. Software quality assurance—defines and conducts the activities required to ensure softwarequality.
4. Technical reviews— evaluates the software engineering work products in an effort to uncover and remove errors before they are transmitted to the next activity.
5. Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.
6. Software configuration management—manages the effects of change throughout the software process.
7. Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.
8. Work product preparation and production—include the activities required to create work products such as models, documents, logs, forms, and lists. Each of these umbrella activities is discussed in detail later in this book.

For many software projects, framework activities are applied iteratively as a project progresses. That is, communication, planning, modeling, construction, **and deployment** are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stake holders (customers or developers or management) with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

4.1 A GENERIC SOFT WARE PROCESS MODEL:

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.



- The software process is represented schematically in the above figure.

Defining a Framework Activity

- A generic process framework for software engineering defines five framework activities
  - **communication,**
  - **planning,**
  - **modeling,**
  - **construction,** and
  - **deployment.**

- In addition, a set of umbrella activities—**project tracking and control, risk management, quality assurance, configuration management, technical reviews,** and others—are applied throughout the process.
- Referring to the figure, each framework activity is populated by a set of software engineering actions.
- Each software engineering action is defined by a *task set* that identifies
- the work tasks that are to be completed,
- the work products that will be produced,
- the quality assurance points that will be required, and
- the milestone(Goal) that will be used to indicate progress or status of the work.

The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity.

1. **Definition phase** focuses on what. That is, during definition, 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 key requirements of the system and the software are identified.

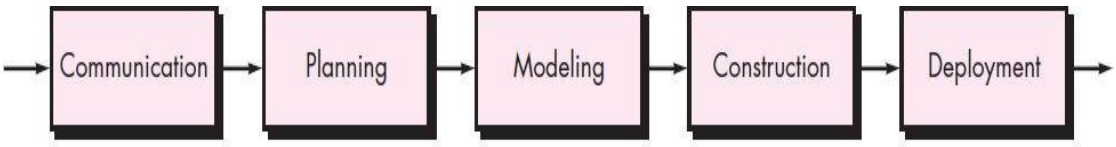
2. **Development phase** focuses on how. A software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed.

3. **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 *process flow*—describes how the framework activities, the actions and the tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in the below Figure.

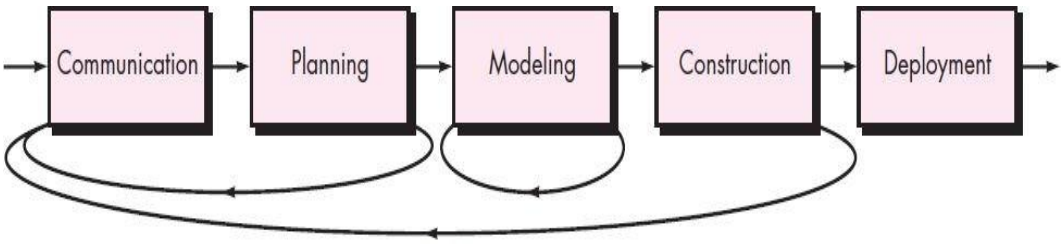
Types of Process flows:

- A **linear process flow** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment which is shown in the following Figure.



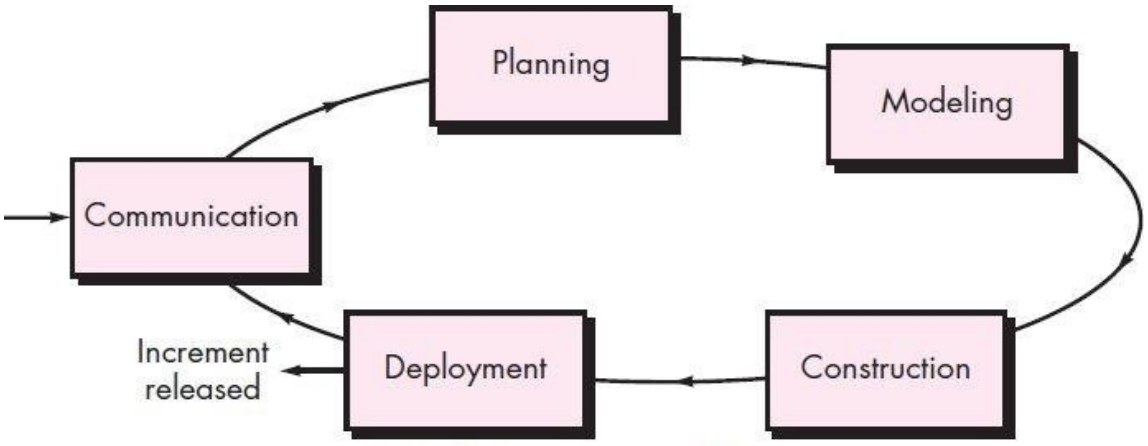
(a) Linear process flow

- An **iterative process flow** repeats one or more of the activities before proceeding to the next (shown in the following Figure).



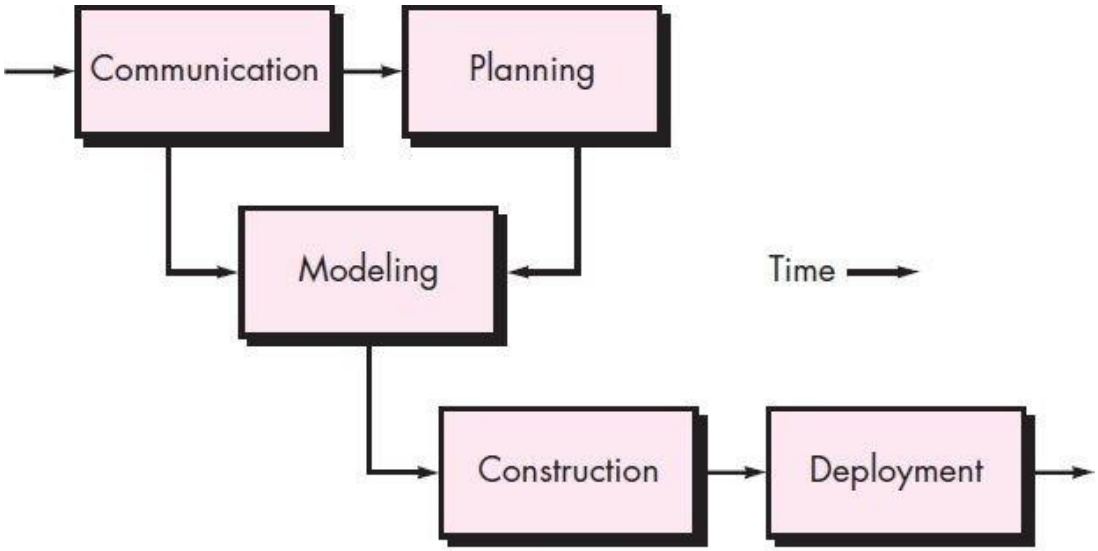
(b) Iterative process flow

- An **evolutionary process flow** executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (shown in the following Figure).



(c) Evolutionary process flow

- A **parallel process flow** executes one or more activities in parallel with other activities
  e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software. (shown in the following figure)



(d) Parallel process flow

5. CAPABILITY MATURITY MODEL INTEGRATION (CMMI)

The **Software Engineering Institute (SEI**) has developed a comprehensive model predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity.
To determine an organization's current state of process maturity, the SEI uses an assessment that results in a six point grading scheme.

Objectives of CMMI:

1. Fulfilling customer needs and expectations.
2. Value creation for investors/stockholders.
3. Market growth is increased.
4. Improved quality of products and services.
5. Enhanced reputation in Industry.

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:

1. **Process Maturity level 0 : Initial or Incomplete**
   - The process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level-1 capability.
2. **Process Maturity level 1 : Performed**
   - All of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.
3. **Process Maturity level 2 : Managed**
   - All level-1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to enough recourses to get the job done; stakeholders are actively involved in the process are as required; all work task and work products are monitored, controlled, and reviewed and are evaluated for quality.
4. **Process Maturity level 3 : Defined**
   - All level-2 criteria have been achieved. In addition the process is modified from the organization's set of standard processes according to the organization's guidelines and contributes work products, measures and other process improvement information to the organizational process resources.
5. **Process Maturity level 4 : Quantitatively managed:**
   - All level-2 criteria have been achieved. In addition the process is is controlled and improved using measurements and quantitative assessment (evaluation).
6. **Process Maturity level 5 : Optimized**
   All level-2 criteria have been achieved. In addition the process area is adapted and optimized to meet changing customer needs and to continually improve the efficacy of the process.

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals.
Specific Goals establish the characteristics that must exist if the activities Implied by a process area are to be effective.
Specific Practices refine a goal into a set of process-related activities.

For example, For Project Planning frame activity that has some **key process areas (KPA)** defined by the CMMI. **The Specific Goals (SG)** and the associated **Specific Practices (SP)** defined for project planning are

SG 1: Establish estimates
- **SP 1.1:** Estimate the scope of the project
- **SP 1.2:** Define Project life cycle
- **SP 1.3:** Determine the estimates of effort and cost

SG 2: Develop a Project Plan
- **SP 2.1:** Establish the budget and schedule
- **SP 2.1:** Identify project risks
- **SP 2.1:** Plan for data management

SP 2.1: Plan for needed knowledge and skills
SP 2.1: Establish the project plan

SG 3: Obtain commitment to the plan
SP 3.1: Review plans that affect the project

The CMMI also defines a set of generic goals and related practices for each process are. To illustrate the **generic goals (GG)** and **generic practices (GP)** for the project planning, process area are:

GG 1: Achieve specific goals
GP 1.1: Perform base practices

GG 2: Institutionalize a managed process
GP 2.1: Plan the process
GP 2.2: Provide resources
GP 2.3: assign responsibility
GP 2.4: train the people
GP 2.5: monitor and control the process

6. Process Patterns

- **Ambler** has proposed a template for describing a process pattern:
- A *Process Pattern* describes a process-related problem that is encountered (occurred) during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides with a template a consistent method for describing problem solutions within the situation of the software process.
- By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.
- Every software team encounters problems as it moves through the software process.
- It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly.
- In some cases, a pattern might be used to describe a problem and solution associated with a complete process model (e.g., prototyping).
- In other situations, patterns can be used to describe a problem and solution associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).

1. **Pattern Name.** The pattern is given a meaningful name describing it within the context of thesoftware process (e.g., **TechnicalReviews**).

2. **Environment.** The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

3. **Type.** The pattern type is specified. Ambler suggests three types:
   1. ***Stage pattern***—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity).

   An example of a stage pattern might be **EstablishingCommunication.** This pattern would incorporate the task pattern **RequirementsGathering** and others.

   2. ***Task pattern***—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).

3. ***Phase pattern***—define the sequence of framework activities that occurs within the process.

4.Initial context. Describes the conditions under which the pattern applies. Prior to theinitiation of the pattern:

(1) What organizational or team-related activities have already occurred?
(2) What is the entry state for the process?
(3) What software engineering information or project information already exists?

For example, the **Planning** pattern (a stage pattern) requires that

(1) customers and software engineers have established a collaborative communication;
(2) the project scope, basic business requirements, and project constraints are known.

5.Problem. The specific problem to be solved by the pattern.

6.Solution. Describes how to implement the pattern successfully. This section describes howthe initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern.

7.Resulting Context. Describes the conditions that will result once the pattern has beensuccessfully implemented. Upon completion of the pattern:

(1) What organizational or team-related activities must have occurred?
(2) What is the exit state for the process?
(3) What software engineering information or project information has beendeveloped?

8.Related Patterns. Provide a list of all process patterns that are directly related to other.

Conclusion on Process Patterns

- Process patterns provide an effective mechanism for addressing problems associated with any software process.

7. PROCESS ASSESSMENT AND IMPROVEMENT

- The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs.
- Process patterns must be coupled with solid software engineering practice
- In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.

A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

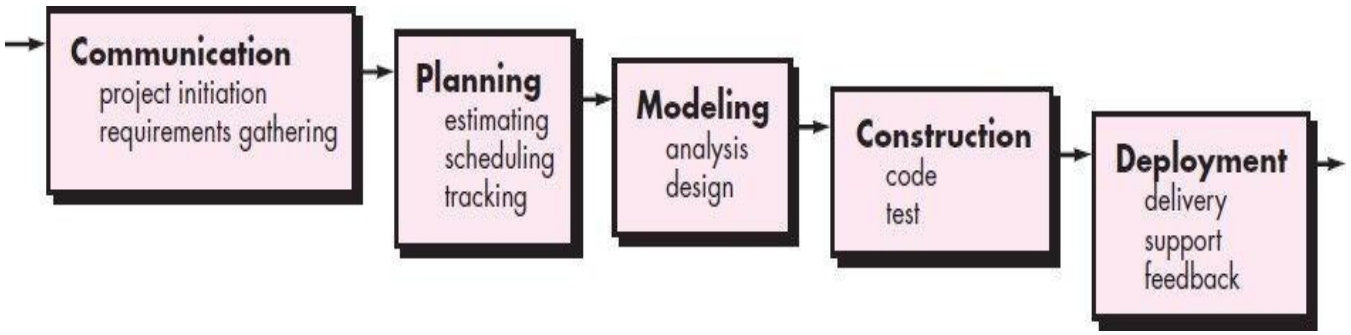
1. **Standard CMMI Assessment Method for Process Improvement (SCAMPI)—** provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.
2. **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.
3. **SPICE (ISO/IEC15504)— Software Process Improvement and Capability Determination** (**SPICE**), is a set of technical standards documents for the computer software development process and related business management functions. It is one of the joint International Organization for Standardization (ISO) and International Electro technical Commission (IEC) standards, which was developed by the ISO and IEC joint subcommittee
4. **ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

8. PROCESS MODELS

- Process model prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.
- All software process models can accommodate the five generic framework activities, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity in a different manner.

8.1 The Waterfall Model

- The waterfall model, sometimes called the *classic life cycle model*
- Winston Royce introduced the Waterfall Model in 1970.
- It suggests a systematic sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment.
- The Waterfall Model was the first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**.
- It is very simple to understand and use.
- The Waterfall model is the earliest SDLC approach that was used for software development.
- The waterfall Model illustrates the software development process in a linear sequential flow.
- This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.
- It is used when requirements are reasonably well understood.



- **Communication**: All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document. The aim of this phase is to understand the exact requirements of the customer and to document them properly. Both the customer and the software developer work together so as to document all the functions, performance, and interfacing requirement of the software. It describes the "what" of the system to be produced and not "how."In this phase, a large document called **Software Requirement Specification (SRS)** document is created which contained a detailed description of what the system will do in the common language.

- **Planning:** A Planning activity creates a "map" defines the work by describing the tasks, risks and resources, work products and work schedule. The map—called a *software project plan*— defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule. It is also used to estimating the cost and budget of the project.

- **Modeling (Analysis and design)**: The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.

- **Construction (Coding and Testing**: With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing. All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.

- **Deployment (delivery, feedback and Maintenance of system)**: Once the functional and non-functional testing is done; the product is deployed (i.e., delivered to the customer) in the customer environment or released into the market. There are some issues which come up in the client environment. To fix those issues, versions are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

Some of the major advantages of the Waterfall Model are as follows:

- Simple and easy to understand and use
- It can serve as a useful process model in situations where requirements are fixed and work is to proceed to complete in a linear manner.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

The major disadvantages of the Waterfall Model are as follows:

The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

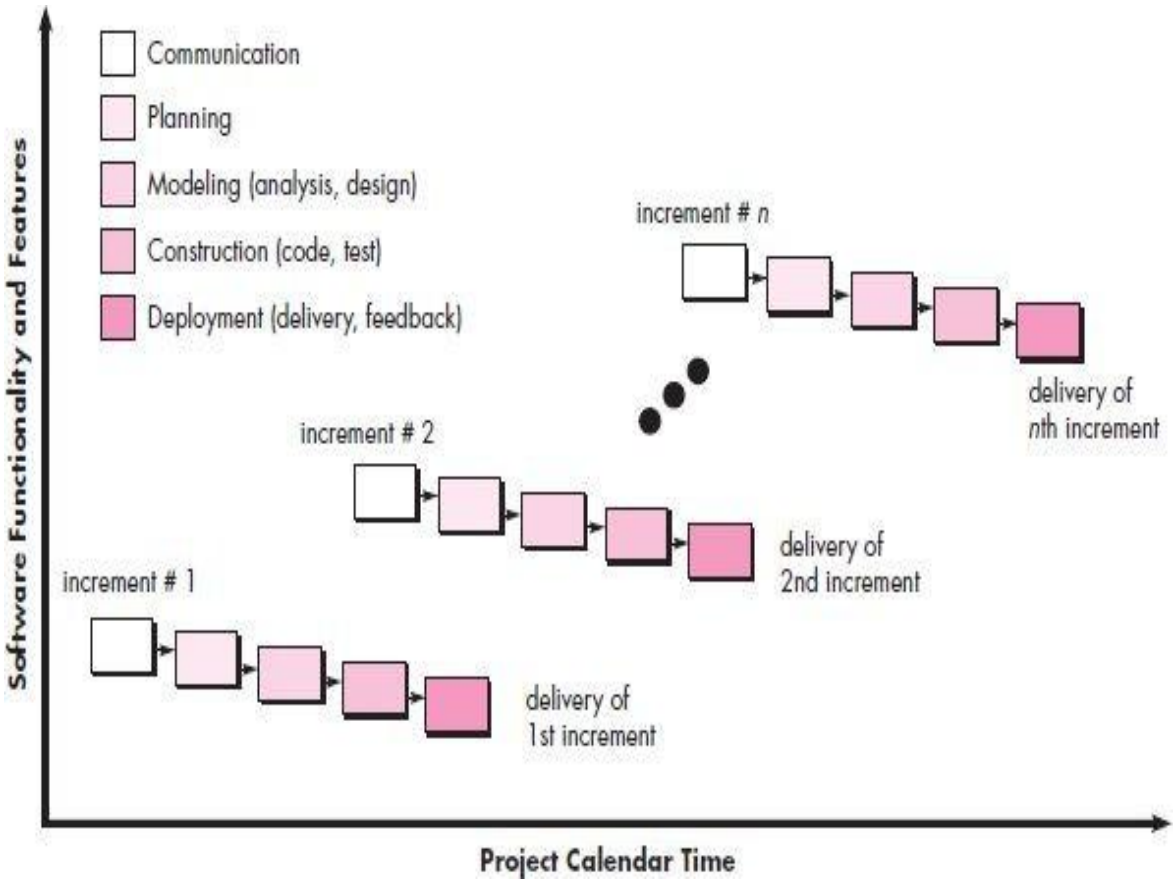
- No working software is produced until late during the life cycle.
- Not a good model for complex and object-oriented projects.
- Poor model or not suitable for long and ongoing projects (large scale projects).
- Not suitable for the projects where requirements are changing frequently and requirements are not clear or not defined properly.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- There exists blocking stages in the teams because one team is working another team may wait.

8.2 Incremental Process Models

There are many situations in which initial software requirements are reasonably well-defined, but the overall scope of the development effort may not use a purely linear process. In addition, there may be a convincing need to provide a limited set of software functionality to users quickly and then improve and expand on that functionality in later software releases. In such cases, a process model that is designed to produce the software in increments is chosen.

8.2.1. Incremental Model

- The incremental process model is also known as the Successive version model.
- First, a simple working system implementing only a few basic features is built first and then that is delivered to the customer. Then thereafter many successive iterations/ versions are implemented and delivered to the customer until the desired system is released.



8.2.1 There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort difficulty to implement linear process.

8.2.2 Need to provide a limited set of software functionality to users quickly and then refineand expand on that functionality in later software releases.

8.2.3 In such cases, we can choose a process model that is designed to produce the software in increments.

8.2.4 The ***incremental*** model combines elements of linear and parallel process flows. The above Figure shows the incremental model which applies linear sequences

8.2.5 Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.

8.2.6 For example, MS-Word software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment;

- more sophisticated editing and document production capabilities in the second increment;
- spelling and grammar checking in the third increment; and
- advanced page layout capability in the fourth increment.
- It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

8.2.7 When an incremental model is used, the first increment is often a ***core product.*** That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use evaluation, a plan is developed for the next increment.

8.2.8 The plan addresses the modification of the core product to better meet the needs of thecustomer and the delivery of additional features and functionality.

8.2.9 This process is repeated following the delivery of each increment, until the complete product is produced.

8.2.10 The incremental process model focuses on the delivery of an operational product with each increment.

8.2.11 Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

8.2.12 Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.

When we use the Incremental Model?

- When the requirements are better.
- A project has a lengthy development schedule.
- When the customer demands a quick release of the product.
- You can develop basic product or core product first.

Advantage of Incremental Model

- Errors are easy to be recognized.
- Easier to test and debug
- More flexible.
- Simple to manage risk because it handled during its iteration.
- The Client gets important functionality early.

Disadvantage of Incremental Model

- Need for good planning
- Total Cost is high.
- Well defined module interfaces are needed.

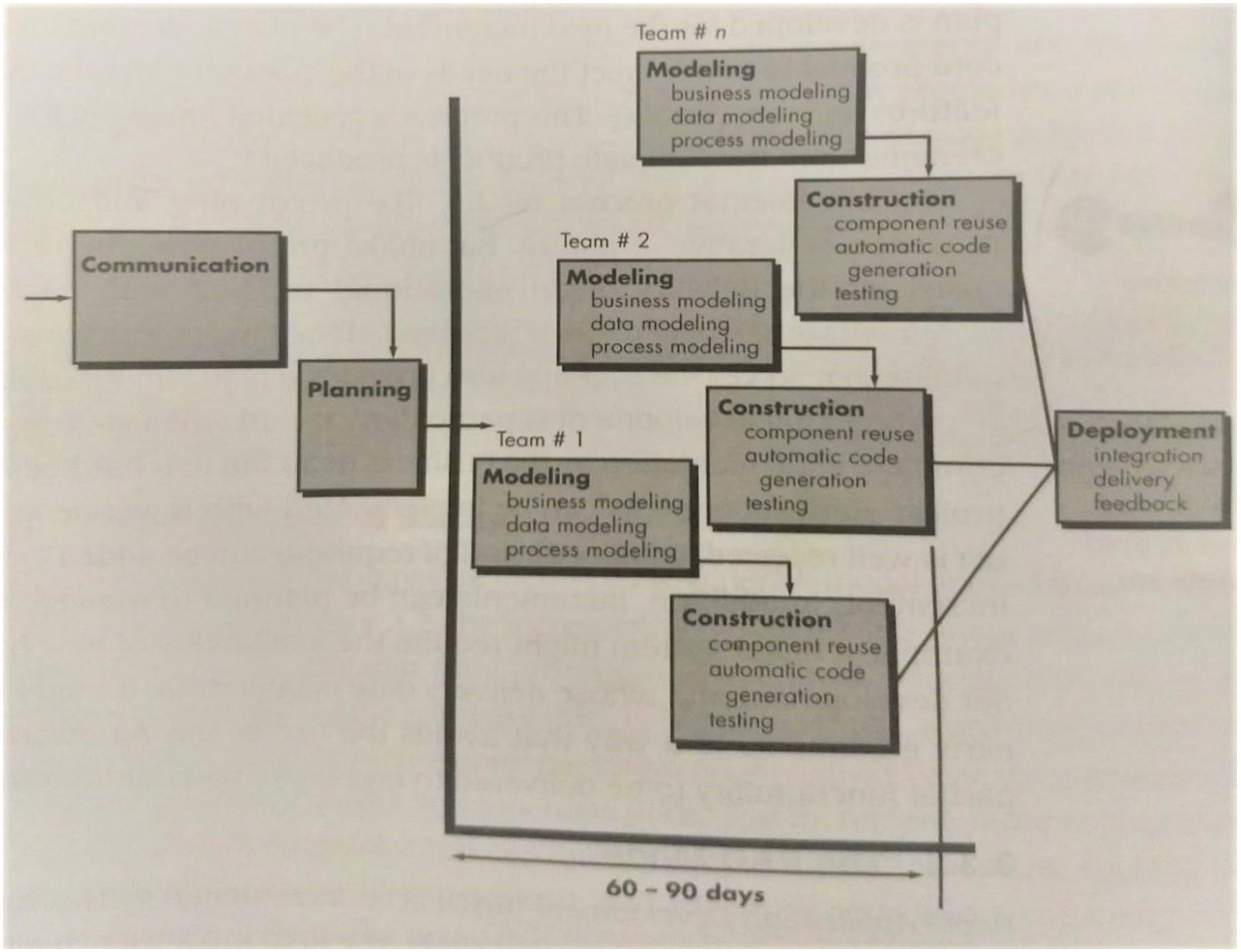
8.2.2. Rapid application development model (RAD)

-> The Rapid Application Development Model was first proposed by IBM in the 1980s.

-> The critical feature of this model is the use of powerful development tools and techniques.

-> A software project can be implemented using this model if the project can be broken down into small modules wherein each module can be assigned independently to separate teams. These modules can finally be combined to form the final product.

-> The process involves building a rapid prototype, delivering it to the customer, and taking feedback. After validation by the customer, the SRS document is developed and the design is finalized.



The Modeling Activity further contains the following three Phases:

Business Modeling:

The business model for the product under development is designed in terms of flow of information and the distribution of information between various business channels. A complete business analysis is performed to find the vital information for business, how it can be obtained, how and when is the information processed and what are the factors driving successful flow of information.

Data Modeling:

The information gathered in the Business Modeling phase is reviewed and analyzed to form sets of data objects vital for the business. The attributes of all data sets is identified and defined. The relation between these data objects are established and defined in detail in relevance to the business model.

Process Modeling:

The data object sets defined in the Data Modeling phase are converted to establish the business information flow needed to achieve specific business objectives as per the business model. The process model for any changes or enhancements to the data object sets is defined in this phase. Process descriptions for adding, deleting, retrieving or modifying a data object are given.

When to use RAD Model?

When the customer has well-known requirements, the user is involved throughout the life cycle, the project can be time boxed, the functionality delivered in increments, high performance is not required, low technical risks are involved and the system can be modularized. In these cases, we can use the RAD Model.

Advantages:

- The use of reusable components helps to reduce the cycle time of the project.
- Feedback from the customer is available at the initial stages.
- Reduced costs as fewer developers are required.
- The use of powerful development tools results in better quality products in comparatively shorter time spans.
- The progress and development of the project can be measured through the various stages.
- It is easier to accommodate changing requirements due to the short iteration time spans.

Disadvantages:

- The use of powerful and efficient tools requires highly skilled professionals.
- The absence of reusable components can lead to the failure of the project.
- The team leader must work closely with the developers and customers to close the project in time.
- The systems which cannot be modularized suitably cannot use this model.
- Customer involvement is required throughout the life cycle.
- It can't be utilized for smaller projects, unfortunately.
- It is not meant for small-scale projects as in such cases, the cost of using automated tools and techniques may exceed the entire budget of the project.

Applications:

1. This model should be used for a system with known requirements and requiring a short development time.
2. It is also suitable for projects where requirements can be modularized and reusable components are also available for development.
3. The model can also be used when already existing system components can be used in developing a new system with minimum changes.
4. This model can only be used if the teams consist of domain experts. This is because relevant knowledge and the ability to use powerful techniques are a necessity.
5. The model should be chosen when the budget permits the use of automated tools and techniques required.

8.3 Evolutionary Process Models

- **Evolutionary model** is a combination of Iterative and Incremental model of software development life cycle.
- They allow developing more complete versions of the software.
- Evolutionary process models produce an increasingly morecomplete version of the software with every iteration.

Advantages of Evolutionary Model

There are many advantages of evolutionary model, some main advantages are mentioned below;

1. The big advantage of the evolutionary model is that the user has checked every stage during the development and it is helpful in achieving customer confidence.
2. There are fewer chances of errors because all the modules are well seen.
3. It helps to reduce the risk of software projects.
4. It also reduces the cost of development.
5. Minimize serious problems during testing.

Disadvantages of Evolutionary Model

1. The **delivery of full software can be late** due to different changes by customers during development.
2. It is **difficult to divide the problem into several parts** that would be acceptable to the customer which can be incrementally implemented and delivered.

Following are the evolutionary process models.

1. The prototyping model
2. The spiral model
3. The Concurrent development model

8.3.1. The Prototyping model

- Prototype is defined as first or preliminary form using which other forms are copied or derived.
- Prototype model is a set of general objectives for software.
- It does not identify the requirements like detailed input, output.
- It is software working model of limited functionality.
- In this model, working programs are quickly produced.

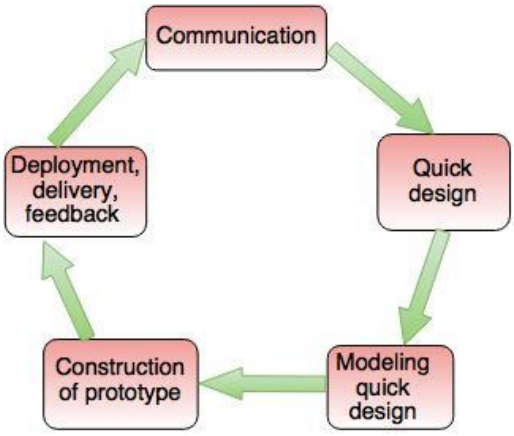


Fig. - The Prototyping Model

The different phases of Prototyping model are:

1. Communication

In this phase, developer and customer meet and discuss the overall objectives of the software.

2. Quick design

- Quick design is implemented when requirements are known.
- It includes only the important aspects like input and output format of the software.
- It focuses on those aspects which are visible to the user rather than the detailed plan.
- It helps to construct a prototype.

3. Modeling quick design

- This phase gives the clear idea about the development of software because the software is now built.
- It allows the developer to better understand the exact requirements.

4. Construction of prototype

The prototype is evaluated by the customer itself.

5. Deployment, delivery, feedback

- If the user is not satisfied with current prototype then it refines according to the requirements of the user.
- The process of refining the prototype is repeated until all the requirements of users are met.
- When the users are satisfied with the developed prototype then the system is developed on the basis of final prototype.

Advantages of Prototyping Model

- Prototype model need not know the detailed input, output, processes, adaptability of operating system and full machine interaction.
- In the development process of this model users are actively involved.
- The development process is the best platform to understand the system by the user.
- Errors are detected much earlier.
- Gives quick user feedback for better solutions.
- It identifies the missing functionality easily. It also identifies the confusing or difficult functions.

Disadvantages of Prototyping Model:

- The client involvement is more and it is not always considered by the developer.
- It is a slow process because it takes more time for development.
- Many changes can disturb the rhythm of the development team.
- It is a thrown away prototype when the users are confused with it.

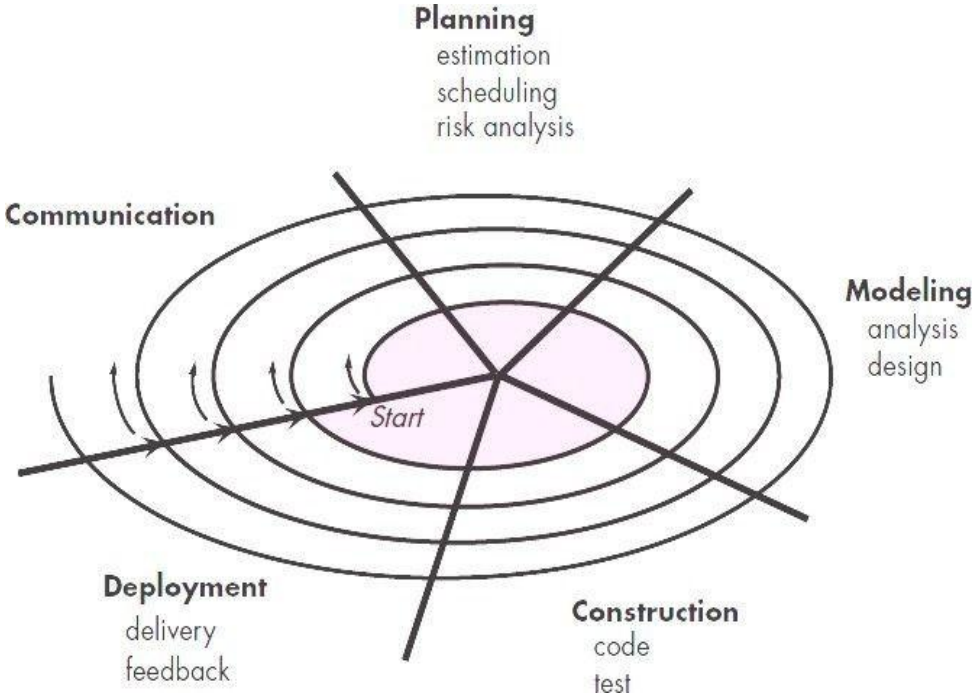
Example:

- If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.
- If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

8.3.2. The Spiral model

- Spiral model is a risk driven process model. It is used for generating the software projects.
- In spiral model, an alternate solution is provided if the risk is found in the risk analysis, then alternate solutions are suggested and implemented.
- It is a combination of prototype and sequential model or waterfall model.
- In one iteration all activities are done, for large projects the output is small.
- The Spiral model is proposed by **Barry Boehm.** The *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfallmodel.
- It provides the potential for rapid development of increasingly more complete versions of the software.
- Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype (sample). During later iterations, increasingly more complete versions of the software are produced.
- **Anchor point milestones-** is a combination of work products and conditions that are achieved along the path of the spiral and these are noted for each evolutionary pass.
- The first circuit around the spiral might result in the development of product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more refined versions of the software.
- Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.
- It maintains the systematic stepwise approach suggested by the classic life cycle butincorporates it into an iterative framework that more realistically reflects the real world.
- The first circuit around the spiral might represent a "**concept development project**" which starts at the core of the spiral and continues for multiple iterations until concept development is complete.

The framework activities of the spiral model are as shown in the following figure.



NOTE: The description of the phases of the spiral model is same as that of the other process models.

Advantages of Spiral Model

- It reduces high amount of risk.
- It is good for large and critical projects.
- It gives strong approval and documentation control.
- In spiral model, the software is produced early in the life cycle process.

Disadvantages of Spiral Model

- It can be costly to develop a software model.
- It is not used for small projects.

8.3.3. The concurrent development model

- The concurrent development model is called as concurrent process model.
- The communication activity has completed in the first iteration and exits in the awaiting changes state.
- The modeling activities completed its initial communication and then go to the under development state.
- If the customer specifies the change in the requirement, then the modeling activity moves from the under development state into the awaiting change state.
- The concurrent process model activities moving from one state to another state.

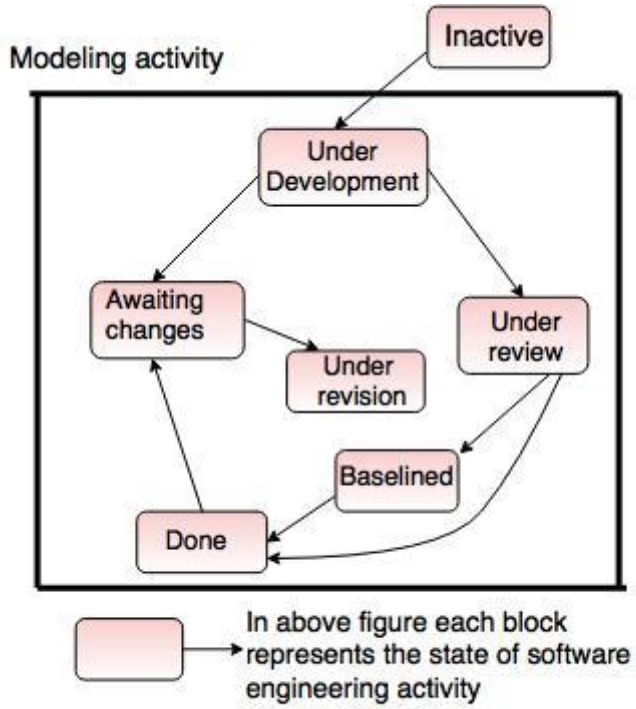


Fig. - One element of the concurrent process model

Advantages of the concurrent development model

- This model is applicable to all types of software development processes.
- It is easy for understanding and use.
- It gives immediate feedback from testing.
- It provides an accurate picture of the current state of a project.

Disadvantages of the concurrent development model

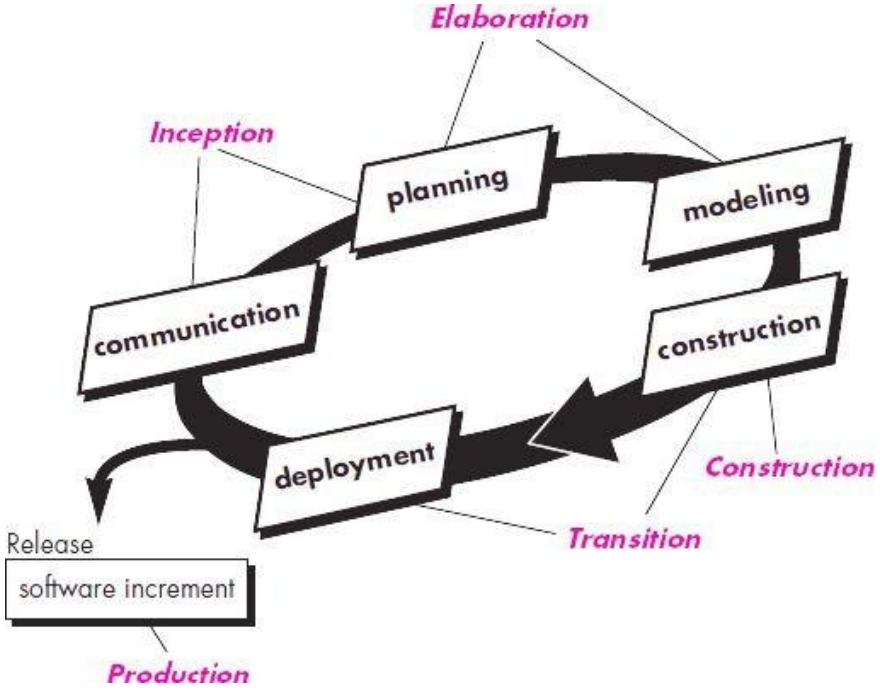
- It needs better communication between the team members. This may not be achieved all the time.
- It requires remembering the status of the different activities.

9. THE UNIFIED PROCESS

- The unified process related to "use case driven, architecture-centric, iterative and incremental" software process.
- The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models.
- The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system.
- It emphasizes the important role of software architecture and "helps the architect focus on the right goals.
- It suggests a process flow that is iterative and incremental.
- During the early 1990s James Rumbaugh, Grady Booch, and IvarJacobson began working on a "unified method".
- The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems.
- UML is used to represent both requirements and design models.
- UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework.
- Over the next few years, Jacobson, Rumbaugh, and Booch developed the *Unified Process*, a framework for object-oriented software engineering using UML.

- Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds.
- The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

9.1 Phases of the Unified Process



- The above figure represents the different phases in Unified Process.
- The ***inception phase*** of the UP includes both customer communication and planning activities.
    - By collaborating with stakeholders, business requirements for the software are identified;
    - a rough architecture for the system is proposed; and
    - a plan for the iterative, incremental nature of the ensuing project is developed.
    - Fundamental business requirements are described.
    - The architecture will be developed.
    - Planning identifies resources, reviews major risks, defines a schedule, andestablishes a basis for the phases.
- The **elaboration phase** includes the communication and modeling activities ofthe generic process model.
    - Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model.
    - In some cases, elaboration creates an "executable architectural baseline" that represents a "first cut" executable system.
    - The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system.
    - In addition, the plan is carefully reviewed.
    - Modifications to the plan are often made at this time.
- The ***construction phase*** of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.

- The elaboration phase reflect the final version of the software increment.
- All necessary and required features and functions for the software increment are then implemented in source code.
- As components are being implemented, unit tests are designed and executed for each.
- In addition, integration activities are conducted.
- Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

- The ***transition phase*** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity.
  - Software is given to end users for beta testing and user feedback reports both defects and necessary changes.
  - In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release.
  - At the conclusion of the transition phase, the software increment becomes a usable software release.
- The ***production phase*** of the UP coincides with the deployment activity of the generic process.
  - During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.
  - It is likely that at the same time the construction, transition, and production phases are being conducted.
  - Work may have already begun on the next software increment.
  - This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.
- A software engineering workflow is distributed across all UP phases.
- In the context of UP, a *workflow* is a task set
- That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks.
- It should be noted that not every task identified for a UP workflow is conducted for every software project.
- The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

10. PERSONAL AND TEAM PROCESS MODELS

- Software process model has been developed at a corporate or organizational level.
- It can be effective only if it is helpful to significant adaptation to meet the needs of the project team that is actually doing software engineering work.
- In an ideal setting, it creates a process that best fits the customer needs, and at the same time,meets the broader needs of the team and the organization.
- Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.
- It is possible to create a “personal software process” and/or a “team software process.”
- Both require hard work, training, and coordination, but both are achievable.

10.1 Personal Software Process (PSP)

- Every developer uses some process to build computer software.
- The process may be temporary; may change on a daily basis; may not be efficient, effective, or even successful; but a "process" does exist.
- Personal process, an individual must move through four phases, each requiring training and careful instrumentation.
- The *Personal Software Process* (PSP) highlights personal measurement of both the work product that is produced and the resultant quality of the work product.
- In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed.
- The PSP model defines five framework activities:

  1. Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimates (the number of defects projected for the work)is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

  2. High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

  3. High-level design review. Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

  4. Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

  5. Postmortem. Using the measures and metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.
- PSP stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.
- PSP represents a disciplined, metrics-based approach to software engineers, the resulting improvement in software engineering productivity and software quality are significant.
- However, PSP has not been widely implemented throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach.
- PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high.

10.2.Team Software Process (TSP)

- *Team Software Process* (TSP) builds a "self-directed" project team that organizes itself to produce high-quality software.
- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.
- A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality);
- TSP identifies a team process that is appropriate for the project and a strategy for implementing the process;
- TSP defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.
- TSP defines the following framework activities:
    - **project launch,**
    - **high-level design,**
    - **implementation,**
    - **integration and test,** and
    - **postmortem.**
- These activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product.
- The postmortem sets the stage for process improvements.
- TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work.
- TSP recognizes that the best software teams are self-directed.
- Team members set
    - project objectives,
    - adapt the process to meet their needs,
    - control the project schedule, and
    - analysis of the metrics collected,
    - work continually to improve the team's approach to software engineering.
- Like PSP, TSP is a exact approach to software engineering that provides distinct and quantifiable benefits in productivity and quality.
- The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

UNIT-II

Software requirements

Objectives:

The objectives of this chapter are to introduce software system requirements and to explain different ways of expressing software requirements. We will learn:

■ understand the concepts of user requirements and system requirements and why these requirements should be written in different ways;

■ understand the differences between functional and non-functional software requirements;

■ understand how requirements may be organized in a software requirements document.

Contents are:

1. Functional and non-functional requirements

2. User requirements

3. System requirements

4. Interface specifications

5. The software requirements document

The requirements:

- The requirements for a system are the descriptions or statements of the services provided by the system and its operational constraints.
- These requirements reflect the needs of customers for a system that helps solve some problem such as controlling a device, placing an order or finding information.
- IEEE defines Requirement as:
  1. A condition or capability or descriptions or statements or wants needed by a user to solve a problem or
     achieve an objective.
  2. The process of establishing the services that the customer requires from a system and theconstraints under which it operates and is developed

3. A condition or capability that must be met or overcome by a system or a system component to satisfy
   Contract (agreement), standard(regular), specification (condition) or formally(officially) imposed document
4. A documented representation of a condition or capability

Requirements engineering:

• The process of finding out, analyzing documenting and checking these services and constraints is called ***requirement engineering.***

The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

Software requirements are necessary

- To describe functional and non-functional requirements or domain requirements.
- To introduce the concepts of user and system requirements
- To explain how software requirements may be organized in a requirements document

1. FUNCTIONAL REQUIREMENTS:

- These are the statements of services the system should provide, how the system should react to particular inputs and, how the system should behave in particular situations.
- Functional requirements should describe all the required functionality or system services.
- Functional requirements describe system services or functions or Functionality or services that the system is expected to provide.
- These are the requirements that the end user specifically demands as basic facilities that the system should offer.
- The functional requirements describe the behavior of the system. It describes functionality or system services.
- It describes the functions of the software or system that must perform.
- A function is nothing but inputs, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do and it should also describe the system services in detail.

The functional requirements for **The LIBSYS system:**

- A library system that provides a single interface to a number of databases of articles in different libraries.
- Users can search a student details such as name, how many books taken, how many books due, address, branch and section using roll number.

Requirements imprecision (ambiguity)

- Problems arise when requirements are not precisely (accurately or correctly) stated.
- Ambiguous (confusing) requirements may be interpreted in different ways by developers and users.

Requirements completeness and consistency:

In principle, requirements should be both complete and consistent.

Completeness:

- They should include descriptions of all facilities

required.

Consistency:

- There should be no conflicts or contradictions in the descriptions

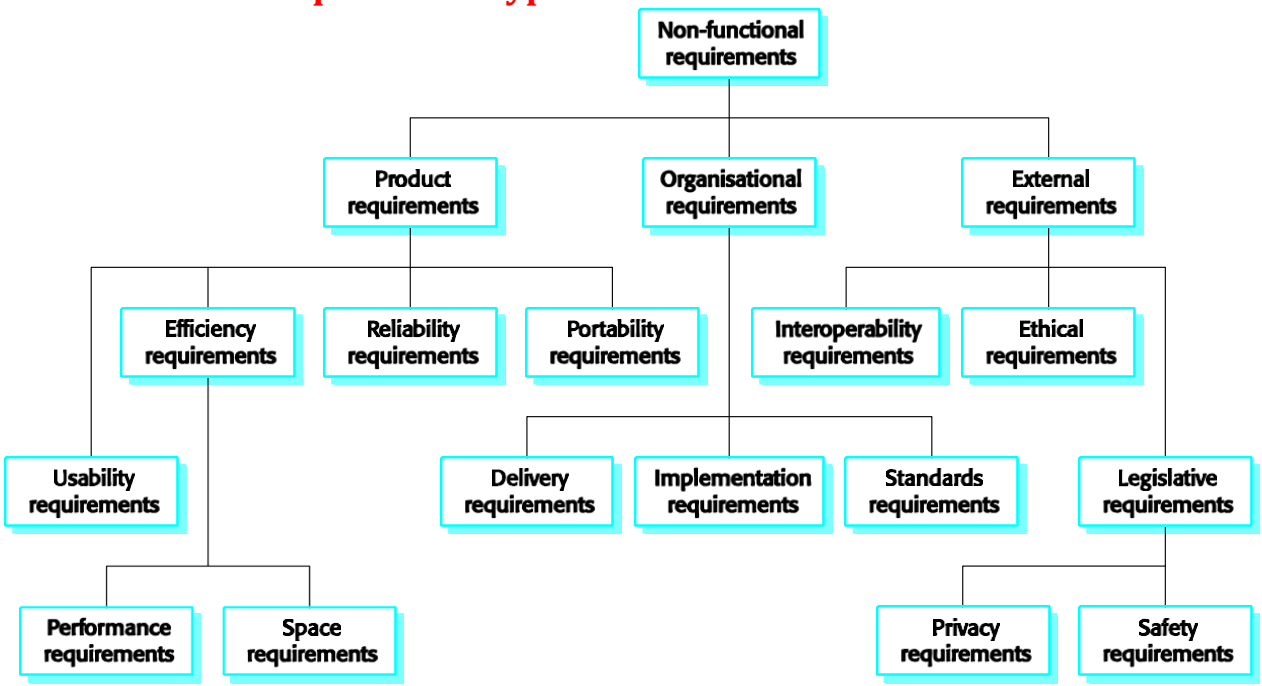
of the system facilities. In practice, it is impossible to produce a complete and consistent requirements document.

2. NON-FUNCTIONAL REQUIREMENTS

- The non functional requirements define system properties and constraints.
- These are basically the quality constraints that the system must satisfy according to the project contract. These are also called non-behavioral requirements.
- Non-functional requirements is a constraint on the system or on the developmentprocess
- Non-functional requirements specify the software's quality attribute. These are the Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Non-functional requirements ensure that the software system must follow the legal and adherence rules. The impact of the non-functional requirements is not on the functionality of the system, but they impact how it will perform. For a well-performing product, at least some of the non-functional requirements should be met.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.

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

Non-functional requirement types:



The types of non-functional requirements are:

a. **Product requirements:** These requirements specify product behaviour.

-- Examples include performance requirements on how fast (speed) the system must execute and how much memory (size) it requires; and also others such as, reliability (Mean time to failure) requirements that set out the acceptable failure rate; portability requirements; Robustness (Time to restart after failure) and usability (ease of use) requirements.

b. Organizational requirements These requirements are derived from policies and procedures in the customer's and developer's organisation.

-- Examples include process standards that must be used; implementation requirements such as the programming language or design method used; and delivery requirements that specify when the product and its documentation are to be delivered.

c. External requirements: This broad heading covers all requirements that are derived from factors external to the system and its development process.

--Examples include interoperability requirements that define how the system interacts with systems in other organisations; legislative (law) requirements that must be followed to ensure that the system operates within the law; and Ethical (moral) requirements are requirements placed on a system to ensure that it will be acceptable to its users and the general public.

3. Domain requirements:

- Domain requirements are expectations related to a particular type of software, purpose or industry.
- These are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain.
- They may be functional or non-functional requirements.
- Domain (area) requirements are the requirements which are characteristic of a particular category or domain of the projects.
- For example, in academic software that maintains records of a school or college, the functionality of being able to access the list of faculty and list of students of each branch is a domain requirement. These requirements are therefore identified from that domain model and are not user specific.

Differences between Functional and Non-functional requirements:

Functional Requirements	Non-functional requirements
Functional requirements help to understand the functions of the system.	They help to understand the system's performance.
Functional requirements are mandatory.	While non-functional requirements are not mandatory.
They are easy to define.	They are hard to define.
They describe what the product does.	They describe the working of product.
It specifies “What should the software system do?”	It places constraints on “How should the software system fulfill the functional requirements?”
It concentrates on the user's requirement.	It concentrates on the expectation and experience of the user.
Helps to verify the functionality of the software.	Helps to verify the performance of the software.
These requirements are specified by the user.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
Functional Testing like System, Integration, End to	Non-Functional Testing like Performance, Stress, Usability,

End, API testing, etc are done.	Security testing, etc are done.
These requirements are important to system operation.	These are not always the important requirements, they may be desirable.
Completion of Functional requirements allows the system to perform, irrespective of meeting the non-functional requirements.	While system will not work only with non-functional requirements.

4. USER REQUIREMENTS

- User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide and the constraints under which it must operate.
- User requirements should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge.
- User requirements are defined using natural language, tables and diagrams as these can be understood by all users.
- A user requirement refers to a function or functionality that the user requires a system to perform.
- Made through statements in natural language and diagrams of the services the system provides and its operational constraints. Written for customers.
- User requirements are set by client and confirmed before system development. – For example, in a system for a bank the user may require a function to calculate interest over a set time period.

Problems with natural language

Lack of clarity:

- It is sometimes difficult to use language in a precise (accuracy) and unambiguous way without making the document wordy and difficult to read.

Requirements confusion:

- Functional and non-functional requirements, system goals and design information may not be clearly distinguished(defined).

Requirements amalgamation (merging):

- Several different requirements may be expressed together as a single requirement.

5. SYSTEM REQUIREMENTS

- System requirements set out the system's functions, services and operational constraints in detail. The system requirements document (sometimes called a functional specification) should be defined.
- It should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.
- System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design.

- They add detail and explain how the user requirements should be provided by the system.
- A system requirement is a more technical requirement, often relating to hardware or software required for a system or software to perform function properly.
- System requirements are more commonly used by developers throughout the development life cycle.
- The system requirements should simply describe the external behaviour of the system and its operational constraints. These are more detailed specifications (or conditions) of system functions, services and constraints than user requirements.
- They should not be concerned with how the system should be designed or implemented.
- The client will usually have less interest in these lower level requirements.
- System requirements may also include validation requirements such as "File upload is limited to .xls format
- They are intended to be a basis for designing the system.
- Natural language is often used to write system requirements specifications as well as user requirements. However, because system requirements are more detailed than user requirements, natural language specifications can be confusing and hard to understand
- System requirements may be defined or illustrated using system models

6. INTERFACE SPECIFICATION

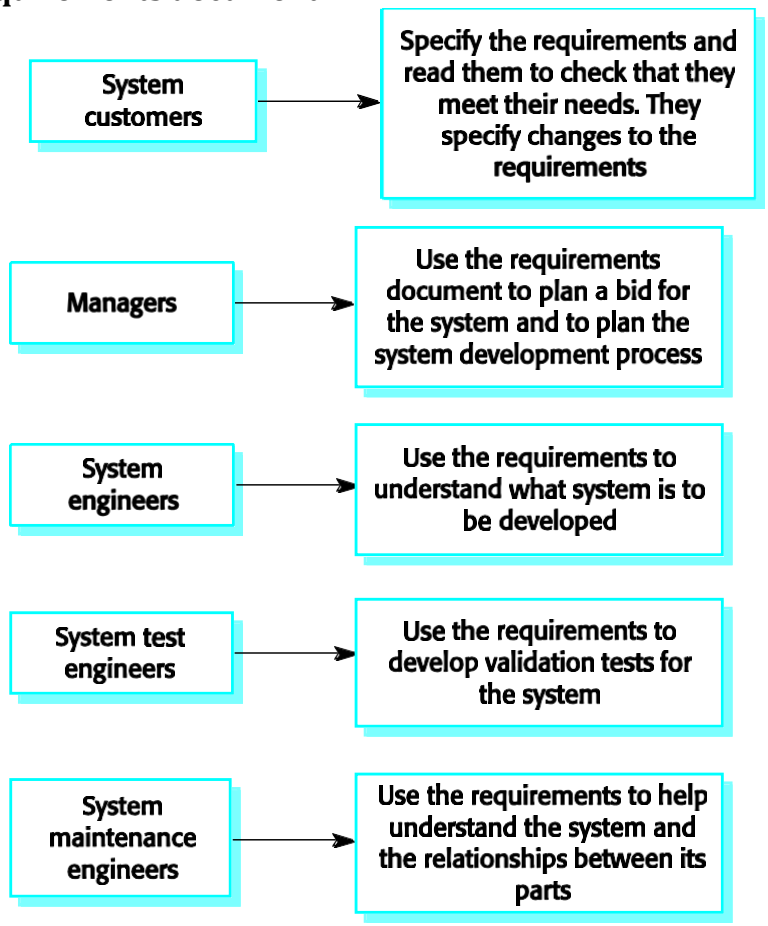
a. Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.
b. Three types of interface may have to be defined
    i. ***Procedural interfaces*** where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Application Programming Interfaces (APIs)
    ii. ***Data structures that are exchanged*** that are passed from one sub-system to another. Graphical data models are the best notations for this type of description
    iii. ***Data representations*** that have been established for an existing sub-system
c. Formal notations are an effective technique for interface specification.

7. THE SOFTWARE REQUIREMENTS DOCUMENT:

a. The software requirements document (sometimes called the software requirements specification or SRS) is the official statement of what the system developers should implement.
b. It should include both the user requirements for a system and a detailed specification of the system requirements.
c. Should provide for communication among team members
d. Should act as an information repository to be used by maintenance engineers
e. Should provide enough information to management to allow them to perform all programmanagement related activities

f. Should describe to users how to operate and administer the system
g. Specify external system behavior
h. Specify implementation constraints
i. Easy to change, modify and serve as reference tool for maintenance
j. Record forethought about the life cycle of the system i.e. predict changes
k. Characterise responses to unexpected events
l. It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it

Users of a requirements document:



IEEE requirements standard defines a generic structure for a requirements document that must be instantiated for each specific system.

1. Introduction.
   i) Purpose of the requirements document
   ii) Scope of the project
   iii) Definitions, acronyms and abbreviations
   iv) References
   v) Overview of the remainder of the document
2. General description.
   i) Product perspective
   ii) Product functions
   iii) User characteristics
   iv) General constraints
   v) Assumptions and dependencies

3. Specific requirements cover functional, non-functional and interface requirements. The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.
4. Appendices.
5. Index.

Figure 6.17 The structure of a requirements document

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe its functions and explain how it will work with other systems. It should describe how the system fits into the overall business or strategic objectives of the organisation commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	The services provided for the user and the non-functional system requirements should be described in this section. This description may use natural language, diagrams or other notations that are understandable by customers. Product and process standards which must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements, e.g. interfaces to other systems may be defined.
System models	This should set out one or more system models showing the relationships between the system components and the system and its environment. These might be object models, data-flow models and semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.
Appendices	These should provide detailed, specific information which is related to the application which is being developed. Examples of appendices that may be included are hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organisation of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, etc.

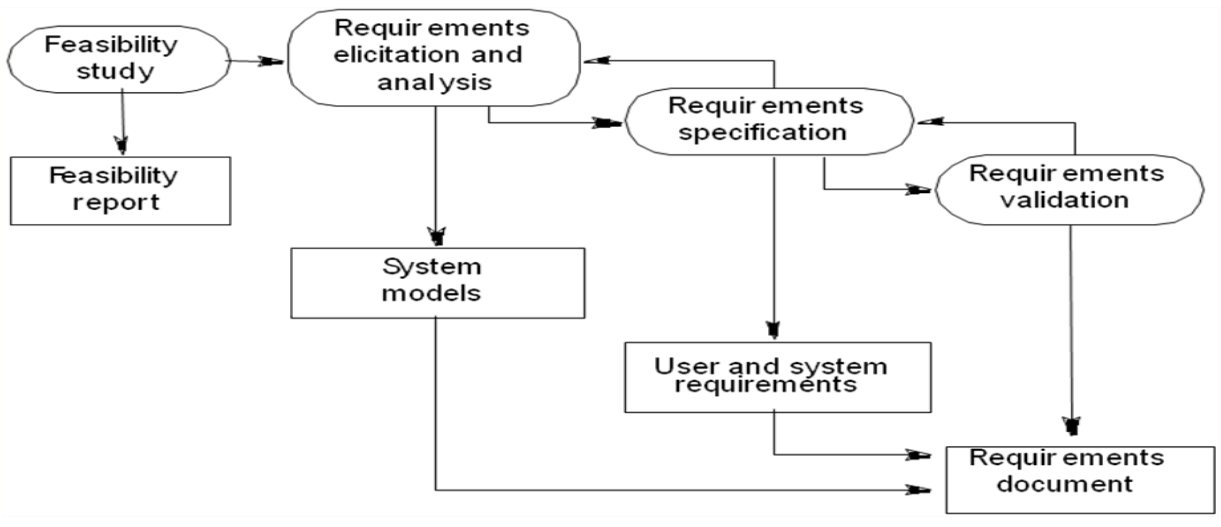
Requirements Engineering Processes (REP):

- Requirement engineering is the process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- REP helps software engineers to better understand the problem to solve.
- It is carried out by software engineers (analysts) and other project stakeholders
- It is important to understand what the customer wants before one begins to design andbuild a computer based system
- REP is a systems and software engineering process which covers all ofthe activities involved in discovering, documenting and maintaining a set of requirements for a computer-based system.
- The processes used for RE vary widely depending on the application domain, the people involved and the organization developing the requirements.

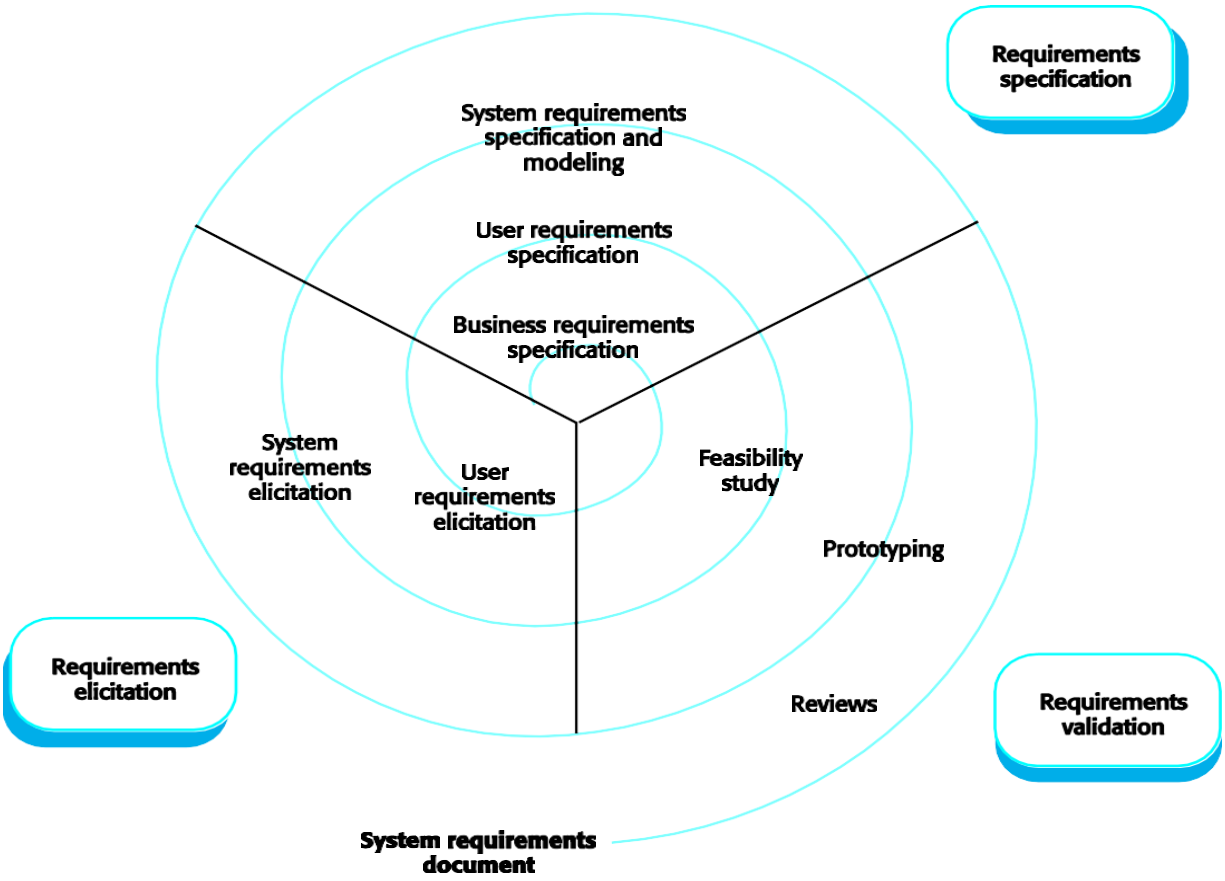
Activities within the RE process may include:

- Assessing(or evaluating or judging) whether the system is useful to the business(feasibility study)
- Requirements elicitation - discovering requirements from system stakeholders
- Requirements Analysis and negotiation - checking requirements and resolvingstakeholder conflicts
- Requirements specification (Software Requirements Specification)- documentingthe requirements in a requirements document
- System modeling - deriving models of the system, often using a notation such as theUnified Modeling Language
- Requirements validation - checking that the documented requirements and modelsare consistent and meet stakeholder needs
- Requirements management - managing changes to the requirements as the system isdeveloped and put into use

Requirements Engineering Processes:



Spiral model of requirements engineering processes



1. Feasibility studies

A feasibility study decides whether or not the proposed system is worthwhile or useful or valuable. The objective behind the feasibility study is to create the reasons for developing the software that is acceptable to users, flexible to change and conformable to established standards.

Types of Feasibility:

1. **Technical Feasibility** - Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.
2. **Operational Feasibility** - Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.
3. **Economic Feasibility** - Economic feasibility decides whether the necessary software can generate financial profits for an organization.

- The purpose of feasibility study is not to solve the problem, but to determine

whether theproblem is worth solving.

- A short focused study that checks
  - If the system contributes to organisational objectives;
  - If the system can be engineered using current technology and within budget;
  - If the system can be integrated with other systems that are used.

Based on information assessment (what is required), information collection and report writing.

- Questions for people in the organisation
  - What if the system wasn't implemented?
  - What are current process problems?
  - How will the proposed system help?
  - What will be the integration problems?
  - Is new technology needed? What skills?
  - What facilities must be supported by the proposed system?

2. Requirement Elicitation and Analysis: Requirement discovery, Interviewing,

Requirements analysis in systems engineering and software engineering, includes those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirements analysis is critical to the success of a systems or software project. The requirements should be documented, actionable, measurable, testable, traceable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

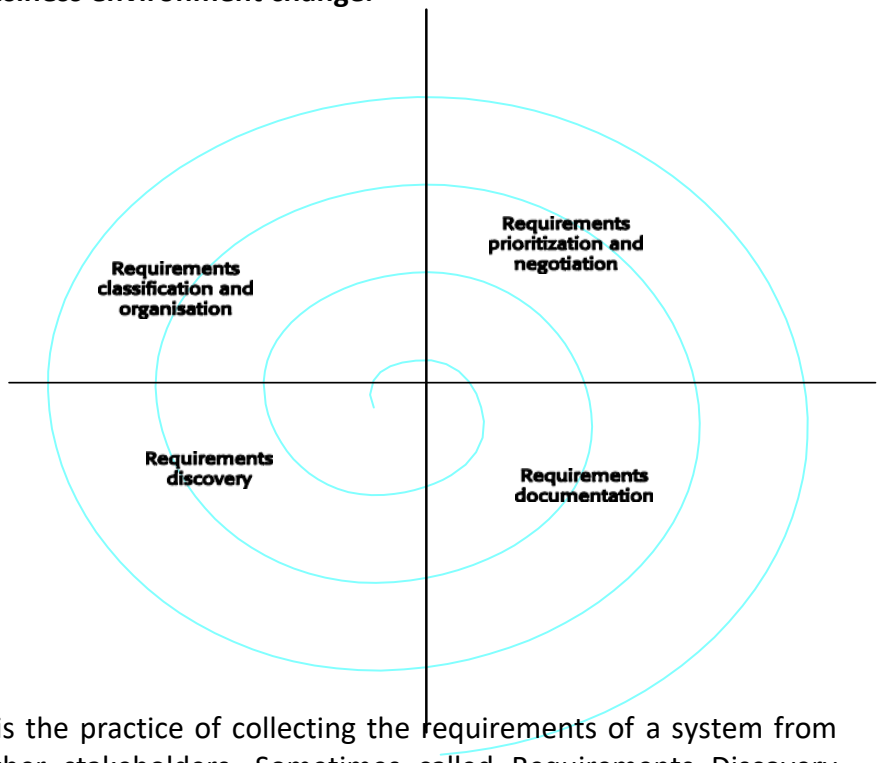
Requirements analysis includes three types of activities

- Eliciting requirements: The task of identifying the various types of requirements from various sources including project documentation, (e.g. the project charter or definition), business process documentation, and stakeholder interviews. This is sometimes also called requirements gathering.
- Analyzing requirements: determining whether the stated requirements are clear, complete, consistent and unambiguous, and resolving any apparent conflicts.
- Recording requirements: Requirements may be documented in various

forms, usually including a summary list and may include natural-language documents, use cases, user stories, or process specifications.

Problems of Requirements Analysis

- Stakeholders don‘t know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and**the business environment change.**



Requirements elicitation is the practice of collecting the requirements of a system from users, customers and other stakeholders. Sometimes called Requirements Discovery Requirements elicitation is important because one can never be sure to get all requirements from the user and customer by just asking them what the system should do Requirements elicitation practices include interviews, questionnaires, user observation, workshops, brain storming, use cases, role playing and prototyping. Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process. Requirements elicitation is a part of the requirements engineering process, usually followed by analysis and specification of the requirements.

Requirements Analysis Process activities

- **Requirements discovery**
  - Interacting with stakeholders to discover their requirements. Domain requirementsare also discovered at this stage.
- **Requirements classification and organisation**

  - Groups related requirements and organises them into consistent clusters.
- **Prioritization and negotiation**
  - Prioritizing requirements and resolving requirements conflicts.
- **Requirements documentation**
  - Requirements are documented and input into the next round of the spiral.

Requirements discovery

- The process of gathering information about the proposed and existing systems and distilling(complete separation) the user and system requirements from this information.
- Sources of information include documentation, system stakeholders and the specificationsof similar systems.

Stakeholder Identification

- Stakeholders (SH) are people or organizations (legal entities such as companies, standards bodies) that have a valid interest in the system. They may be affected by iteither directly or indirectly.

- Stakeholders are not limited to the organization employing the analyst. Other stakeholders will include:
  - Anyone who operates the system (normal and maintenance operators)
  - Anyone who benefits from the system (functional, political, financial and social beneficiaries)

Other stakeholders will include:

- Anyone involved in purchasing or procuring the system. In a mass-market product organization, product management, marketing and sometimes sales act as surrogate consumers (mass-market customers) to guide development of the product
- Organizations which regulate aspects of the system (financial, safety, and other regulators)
- People or organizations opposed to the system (negative stakeholders
- Organizations responsible for systems which interface with the system under design

Requirements Discovery Techniques:

1. Viewpoints:

- Key strength of viewpoint-oriented analysis is that it recognizes multiple perspectives and provides a framework for discovering conflicts in the requirements proposed by different stakeholders.
- Viewpoints are a way of structuring the requirements to represent the perspectives ofdifferent stakeholders. Stakeholders may be classified under

different viewpoints.

- This multi-perspective analysis is important as there is no single correct way to analyze system requirements.
- Types of viewpoint
  - Interactor viewpoints
    - People or other systems that interact directly with the system. In an ATM, the customers and the account database are interactor VPs.
  - Indirect viewpoints
    - Stakeholders who do not use the system themselves but who influence the requirements. In an ATM, management and security staff are indirect viewpoints.
  - Domain viewpoints
    - Domain characteristics and constraints that influence the requirements. In an ATM, an example would be standards for inter-bank communications.

2.Interviewing

- The interview is the primary technique for information gathering during the systems analysis phases of a development project. It is a skill which must be mastered by every analyst.
- The interviewing skills of the analyst determine what information is gathered, and the quality and depth of that information. Interviewing, observation, and research are the primary tools of the analyst.
- In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.

Goals of the Interview

- At each level, each phase, and with each interviewee, an interview may be conducted to:
  - Gather information on the company
  - Gather information on the function
  - Gather information on processes or activities
  - Uncover problems
  - Conduct a needs determination
  - Verification of previously gathered facts
  - Gather opinions or viewpoints

- Provide information
- Obtain leads for further interviews

Interviews are two types:

- Closed interviews where a pre-defined set of questions are answered.
- Open interviews where there is no pre-defined agenda and a range of issues areexplored with stakeholders.

- Normally a mix of closed and open-ended interviewing is undertaken.
- Interviews are not good for understanding domain requirements
  - Requirements engineers cannot understand specific domain terminology;
  - Some domain knowledge is so familiar that people find it hard to articulate or thinkthat it isn‘t worth articulating.
- Effective Interviewers
  - Interviewers should be open-minded, willing to listen to stakeholders and should nothave pre-conceived ideas about the requirements.
  - They should prompt the interviewee with a question or a proposal and should notsimply expect them to respond to a question such as what do you want‘.
- Information from interviews supplement other information about the system fromdocuments, user observations, and so on
- Sometimes, apart from information from documents, interviews may be the only source ofinformation about the system requirements
- It should be used alongside other requirements elicitation techniques

3. Scenarios, Use cases, Ethnography:

1. Scenarios:

- Scenarios are real-life examples of how a system can be used.
- Scenarios can be particularly useful for adding detail to an outline requirementsdescription.
- Each scenario covers one or more possible interactions
- Several forms of scenarios can be developed, each of which provides different types ofinformation at different levels of detail about the system
- Scenarios may be written as text, supplemented by diagrams, screen shots and so on.
- A scenario may include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong;
  - Information about other concurrent activities that might be going on at

the sametime
- A description of the system state when the scenario finishes.

Scenario-based elicitation involves working with stakeholders to identify scenarios and to capturedetails to be included in these scenarios. Scenarios may be written as text, supplemented by diagrams, screen shots, etc. Alternatively, a more structured approach such as event scenarios oruse cases may be used.

2. Use Cases

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- Sequence diagrams may be used to add detail to use-cases by showing the sequence of eventprocessing in the system.
- Use-case approach helps with requirements prioritization

A Use case can have high priority for
- It describes one of the business process that the system enables
- Many users will use it frequently
- A favored user class requested it
- It provides capability that's required for regularity compliance
- Other system functions depend on its presence

3. Ethnography

- Ethnography is an observational technique that can be used to understand social and organizational requirements. Ethnography means is a comparative study of people.
- Requirements obtained from working style of people.
- Requirements obtained from inter-actives performed by the people i.e., gathering the requirements based on the people's activites.
- Scientists spend a considerable time observing and analyzing how people actuallywork.
- People do not have to explain their work.
- Social and organizational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex thansuggested by simple system models.

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

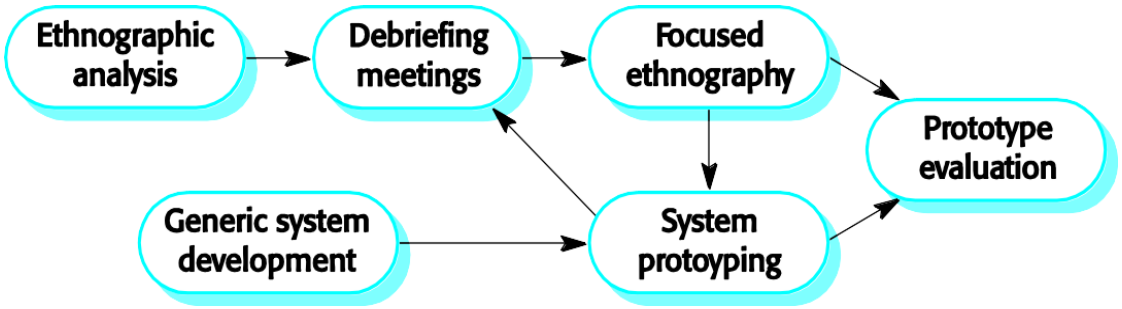
Focused ethnography

- Combines ethnography with prototyping
- Prototype development results in unanswered questions which focus the

ethnographicanalysis.

- The problem with ethnography is that it studies existing practices which may have somehistorical basis which is no longer relevant.

Ethnography and prototyping



The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problemsand questions that can then be discussed with the ethnographer.

3. Software Requirement Specification:

Software requirement specification is a kind of document which is created by a software analyst after the requirements collected from the various sources - the requirement received by the customer written in ordinary language. It is the job of the analyst to write the requirement in technical language so that they can be understood and beneficial by the development team.

The models used at this stage include ER diagrams, data flow diagrams (DFDs), function decomposition diagrams (FDDs), data dictionaries, etc.

- **Data Flow Diagrams:** Data Flow Diagrams (DFDs) are used widely for modeling the requirements. DFD shows the flow of data through a system. The system may be a company, an organization, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or bubble chart.
- **Data Dictionaries:** Data Dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developers use the same definition and terminologies.

Entity-Relationship Diagrams: Another tool for requirement specification is the entity-relationship diagram, often called an "***E-R diagram***." It is a detailed logical

representation of the data for the organization and uses three main constructs i.e. data entities, relationships, and their associated attributes

Software requirement specification is a kind of document which is created by a software analyst after the requirements collected from the various sources - the requirement received by the customer written in ordinary language. It is the job of the analyst to write the requirement in technical language so that they can be understood and beneficial by the development team.

Structured natural language (Structured language specifications)

- It is a way of writing the system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way. .
- The advantage of this approach is that it maintains most of the expressiveness and understandability of natural language but ensures that some degree of uniformity is imposed on the specification.
- A detailed software description which can serve as a basis for a design or implementation. It is written for developers.

System requirement specification(SRS) using a standard form:

1. Function
2. Description
3. Inputs
4. Source
5. Outputs
6. Destination
7. Action
8. Requires
9. Pre-condition
10. Post-condition
11. Side-effects

4. Requirement Validation

- It is concerned with demonstrating that the requirements define the system that the customerreally wants i.e., It is a process in which it is checked that whether the gathered requirements represent the same system that customer really wants.
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixingan implementation error.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

- **Validity.** Does the system provide the functions which best support the customer‘s needs?

- **Consistency.** Are there any requirements conflicts?
- **Completeness.** Are all functions required by the customer included?
- **Realism.** Can the requirements be implemented according to given available budget and technology?
- **Verifiability:** Can the requirements be checked?

Requirements Validation Techniques

- **Requirements reviews**
  - Systematic manual analysis of the requirements.
  - Regular reviews should be held while the requirements definition is being formulated.
  - Both client and contractor staff should be involved in reviews.
  - Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.
  - Don't underestimate the problems involved in requirements validation. Ultimately, it is difficult to show that a set of requirements does in fact meet a user's needs. Users need to picture the system in operation and imagine how that system would fit into their work.
- **Prototyping**
  - Using an executable model of the system to check requirements.
- **Test-case generation**
  - Developing tests for requirements to check testability.

    - **Verifiability.** Is the requirement realistically testable?
    - ***Comprehensibility*:** Is the requirement properly understood?
    - ***Traceability*:** Is the origin of the requirement clearly stated?
    - ***Adaptability*:** Can the requirement be changed without a large impact on other requirements?

5. Requirement Management

Requirements management is the process of managing the changing requirements during therequirements engineering process and system development.

Requirements are inevitably (certainty) incomplete and inconsistent

- New requirements emerge during the process as business needs change and a betterunderstanding of the system is developed;
- Different viewpoints have different requirements and these are often contradictory.

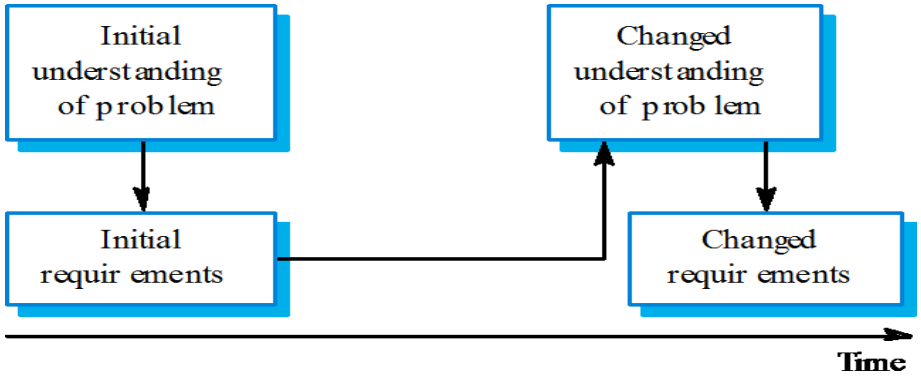
Requirements Change

- The priority of requirements from different viewpoints changes during the

developmentprocess.

- System customers may specify requirements from a business perspective that conflict withend-user requirements.
- The business and technical environment of the system changes during its development.

Requirements Evolution



Enduring (stable or permanent) requirements

- These are relatively stable requirements that derive from the core activity of theorganization
- Relate directly to the domain of the system
- These requirements may be derived from domain models that show the entities andrelations which characterize an application domain
- For example, in a hospital there will always be requirements concerned with patients,doctors, nurses, treatments, etc

Volatile requirements

- These are requirements that are likely to change during the system development process orafter the system have been become operational.
- Examples of volatile requirements are requirements resulting from government health-carepolicies or healthcare charging mechanisms.

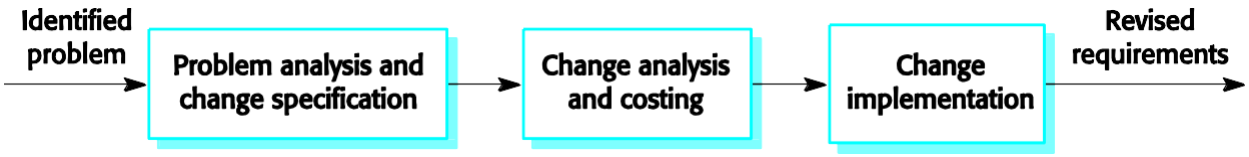
Traceability

- Traceability is concerned with the relationships between requirements, their sources and the system design
- Source traceability
  - Links from requirements to stakeholders who proposed these requirements;
- Requirements traceability
  - Links between dependent requirements;
- Design traceability
  - Links from the requirements to the design;
- Requirements storage
  - Requirements should be managed in a secure, with in a managed data store.

- Change management
  - The process of change management is a workflow process whose stages can bedefined and information flow between these stages partially automated.
- Traceability management
  - Automated retrieval of the links between requirements.

Requirements Management Planning

- During the requirements engineering process, one has to plan:
  - Requirements identification
    - How requirements are individually identified;
  - A change management process
    - The process followed when analyzing a requirements change;
  - Traceability policies
    - The amount of information about requirements relationships that ismaintained;
  - CASE tool support
    - The tool support required to help manage requirements change;
- Should apply to all proposed changes to the requirements.
- Principal stages
  - Problem analysis. Discuss requirements problem and propose change;
  - Change analysis and costing. Assess effects of change on other requirements;
  - Change implementation. Modify requirements document and other documents toreflect change.



System models

- Modeling consists of building an abstraction of reality. Abstractions are simplifications because: They ignore irrelevant details and they only represent the relevant details. What is relevant or irrelevant depends on the purpose of the model.
- System modeling is the process of developing abstract models of a system. Each model presenting a different view or perspective of that system from different perspectives
- System modeling helps the analyst to understand the functionality of the system and models are used to communicate with customers

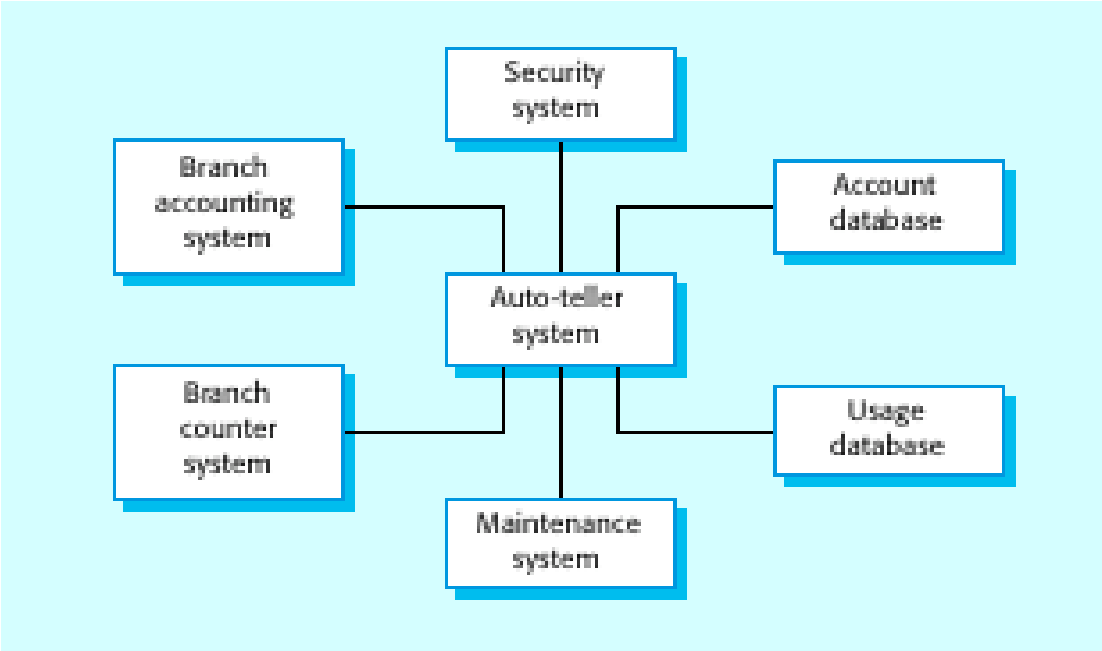
Types of System models are

1. **Context models**
2. **Behavioral models**
3. **Data models**
4. **Object models**

1. Context models

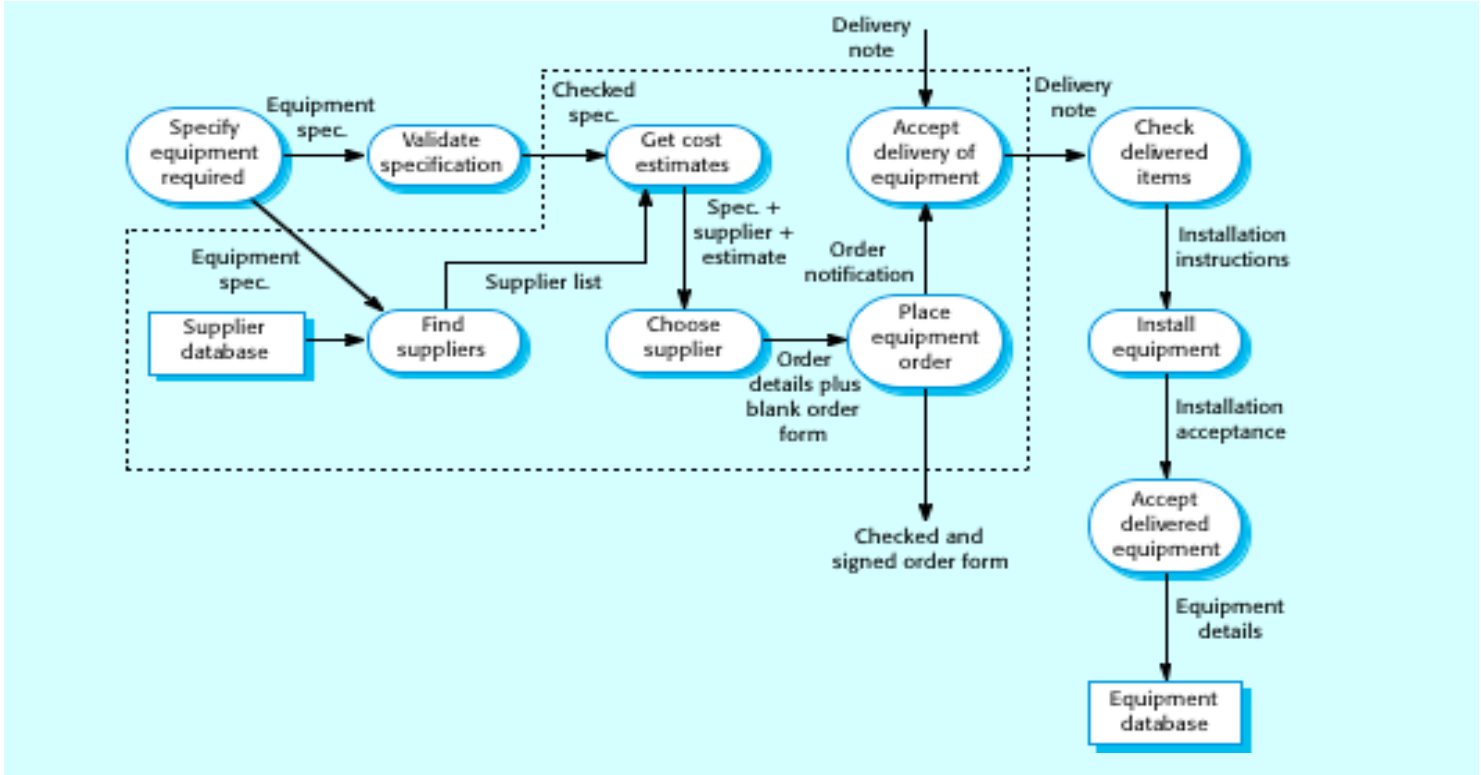
- Defines the physical scope of the system: i.e., what is part of the system (under your control) and what is external to the system.
- It is an external perspective i.e., Model the context or environment of the system.
- Context models are used to illustrate the **operational context** of a system - They show what lies outside the system boundaries.
- **Context models** simply **show** the other systems in the environment, not **how** the system being developed is used in that environment.
- System boundaries are established to define **what is inside and what is outside the system.**
- It shows other systems that are used or depend on the system being developed.

The context of an ATM system



- **Data flow models or Activity Diagrams may be used to show the processes and the flow of information fromone process to another**

Equipment procurement process



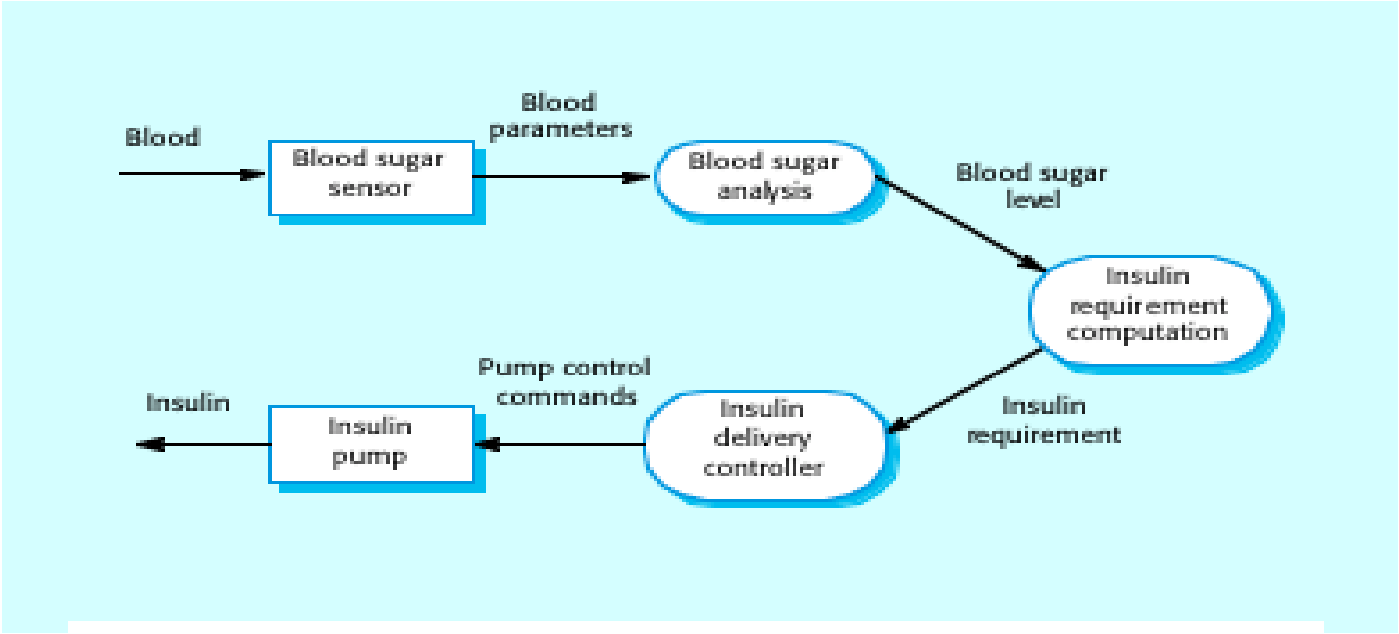
2. Behavioral Model

- Behavioral models are used to describe the overall behavior of a system.
- **Behavioral Model** is specially designed to make us understand behavior and factors that influence behavior of a System.

i. Data Flow Diagrams (DFD)

- Data flow diagrams (DFDs) may be used to model the system's data processing.
- Data flow models may be used to show the processes and the flow of information from one process to another.
- In Software engineering DFD (data flow diagram) can be drawn to represent the system of different levels of abstraction.
- Higher-level DFDs are partitioned into low levels-hacking more information and functional elements.
- It is used to show how data is processed as it moves through the system.
- These show the processing steps as data flows through a system.
- DFDs model the system from a functionalperspective.
- Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system.
- Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment.

Insulin pump DFD



Levels in DFD

Levels in DFD are numbered 0, 1, 2 or beyond. Here, we will see mainly 3 levels in the data flow diagram, which are: 0-level DFD, 1-level DFD, and 2-level DFD.

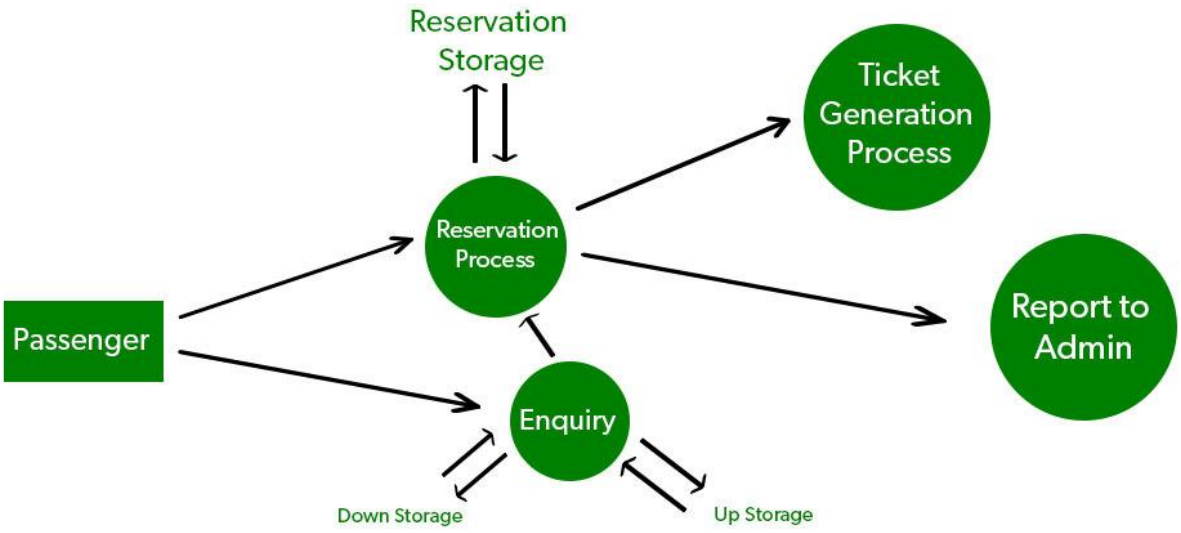
0-level DFD:

It is also known as a context diagram. It's designed to be an abstraction view, showing the system as a single process with its relationship to external entities. It represents the entire system as a single bubble with input and output data indicated by incoming/outgoing arrows.

0-LEVEL DFD

1-level DFD:

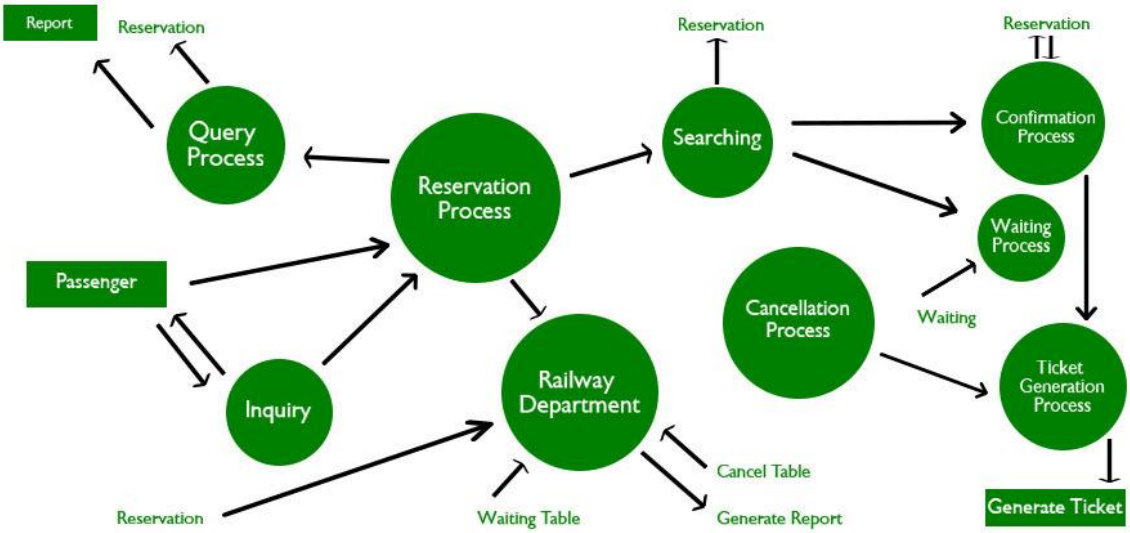
In 1-level DFD, the context diagram is decomposed into multiple bubbles/processes. In this level, we highlight the main functions of the system and breakdown the high-level process of 0-level DFD into sub processes.



1-LEVEL DFD

2-level DFD:

2-level DFD goes one step deeper into parts of 1-level DFD. It can be used to plan or record the specific/necessary detail about the system's functioning.

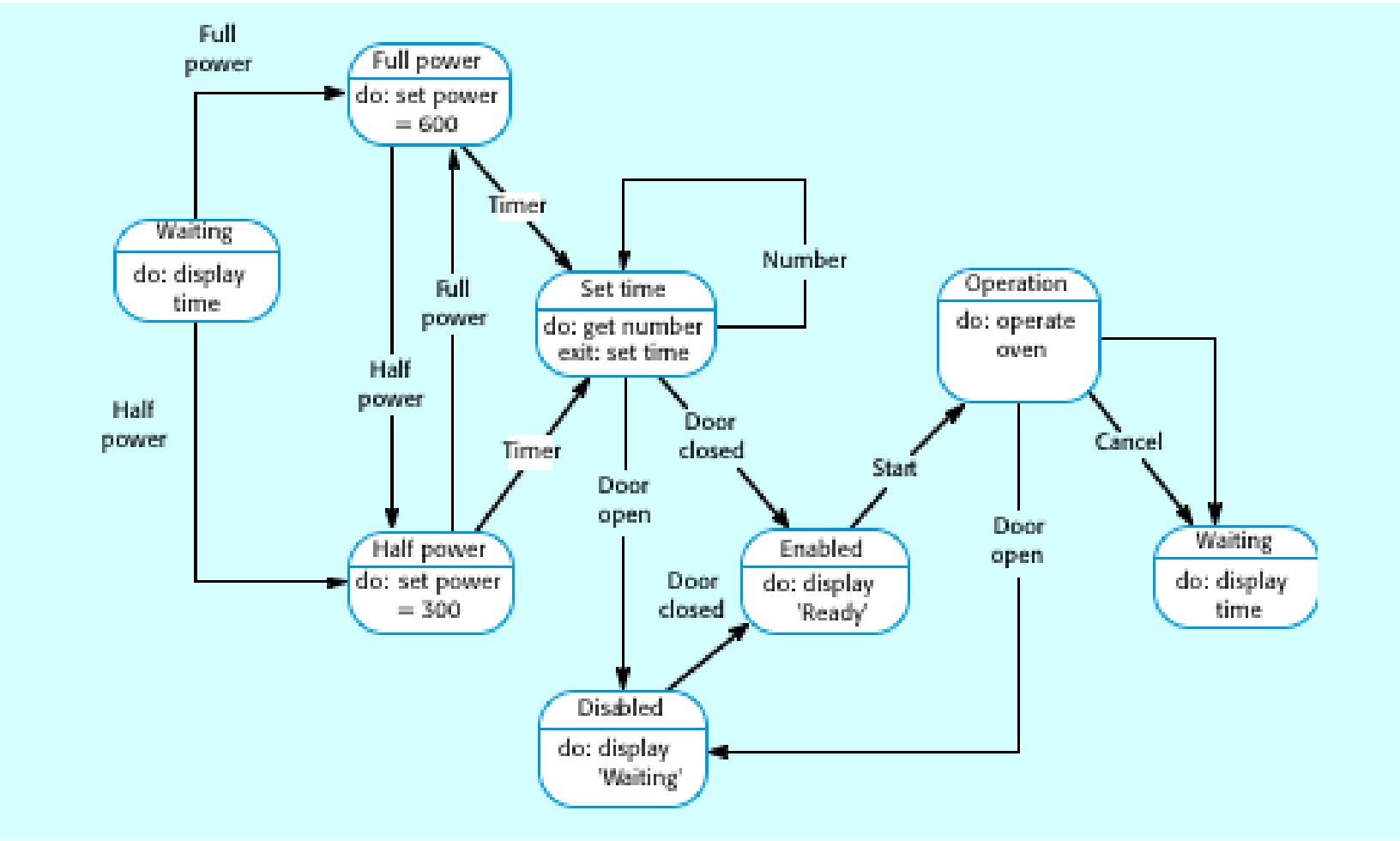


2-LEVEL DFD

ii. State Transition Diagram or state chart or state machine diagram

- Behavior of a system is explained and represented with the help of a diagram.
- This diagram is known as State Transition Diagram or state chart or state machine diagram. It is a collection of states and events.
- It usually describes overall states that a system can have and events which are responsible for a change in state of a system.
- It models the behaviour of the system inresponse to external and internal events.
- They show the system's responses to stimulus (incentive or motivate) so are often used for modeling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- State charts are an integral part of the UML and are used to represent state machine models.
- So, on some occurrence of a particular event, an action is taken and what action needs to be taken is represented by State Transition Diagram.
- Behavioral perspective - Model the dynamic behavior of the system and how it responds to events - Behavioral models, State machine models
- Allow the decomposition of a model into sub-models (see following slide).
- A brief description of the actions is includedfollowing the 'do' in each state.
- Can be complemented by tables describingthe states and the stimuli.

Example-1:Microwave oven model



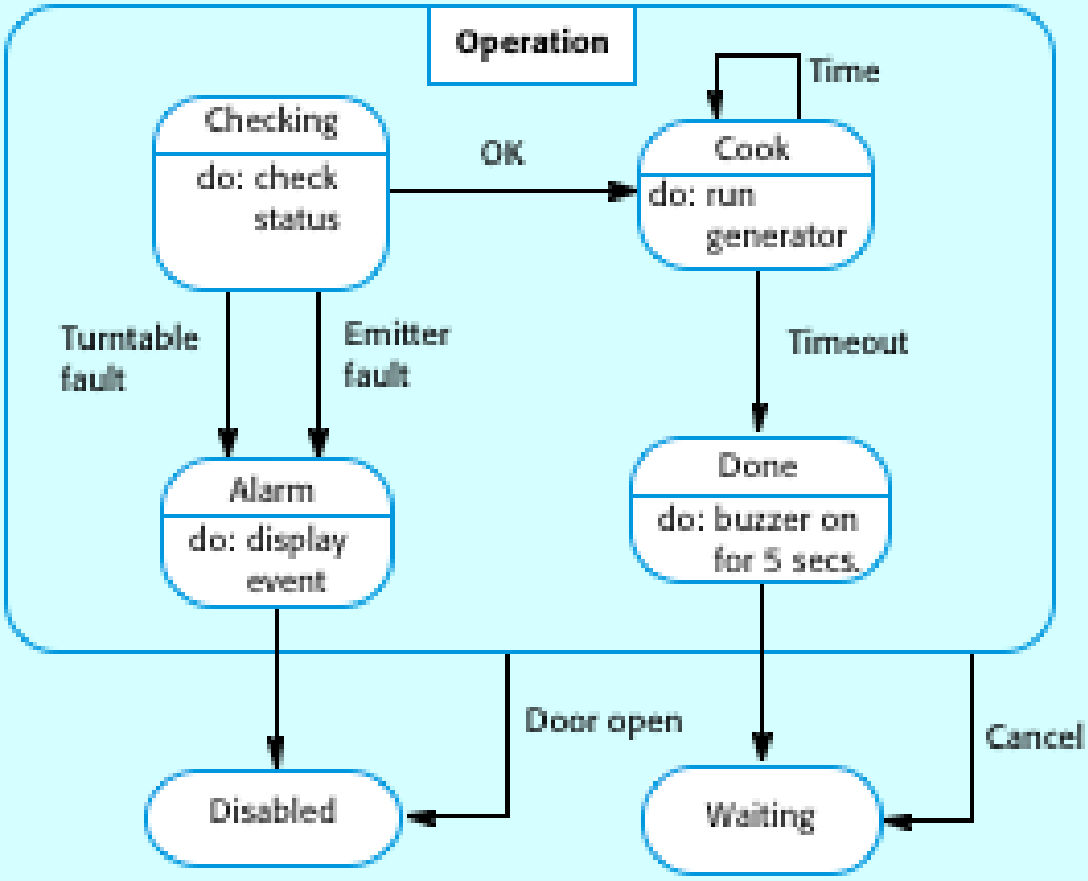
Microwave oven state description

State	Description
Waiting	The oven is waiting for input. The display shows the current time. The oven
Half power	power is set to 300 watts. The display shows ‘Half power’.The oven power is
Full power	set to 600 watts. The display shows ‘Full power’.
Set time	The cooking time is set to the user’s input value. The display shows the cooking timeselected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows ‘Notready’.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows ‘Ready to cook’.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 s econds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding.

Microwave oven stimulus (motivation)

Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	

Microwave oven operation



Example2:

Consider an Elevator. This elevator is for n number of floors and has n number of buttons one for each floor.

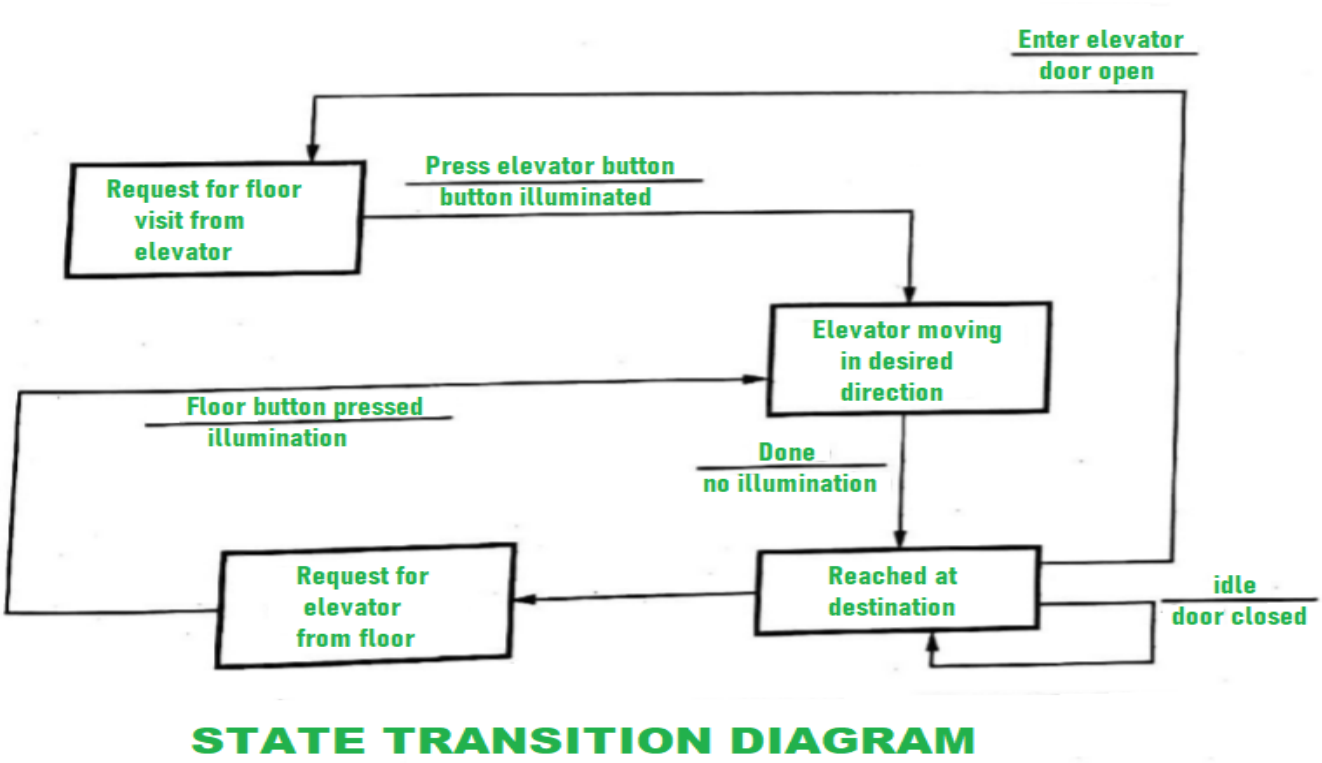
Elevator's working can be explained as follows:

1. **Elevator buttons** are type of set of buttons which is there on elevator. For reaching a particular floor you want to visit, "elevator buttons" for that particular floor is pressed. Pressing, will cause illumination and elevator will start moving towards that particular floor for which you pressed "elevator buttons". As soon as elevator reaches that particular floor,
   illumination gets canceled.
2. **Floor buttons** are another type of set of buttons on elevator. If a person is on a particular floor and he wants to go on another floor, then elevator button for that floor is pressed.

Then, process will be same as given above. Pressing, will cause illumination and elevator to start moving, and when it reaches on desired floor, illumination gets canceled.

3. When there is no request for elevator, it remains closed on current floor.

State Transition Diagram for an elevator system is shown below –



STATE TRANSITION DIAGRAM

Advantages :

- Behavior and working of a system can easily be understood without any effort.
- Results are more accurate by using this model.
- This model requires less cost for development as cost of resources can be minimal.
- It focuses on behavior of a system rather than theories.

Disadvantages :

- This model does not have any theory, so trainee is not able to fully understand basic principle and major concept of modeling.
- This modeling cannot be fully automated.
- Sometimes, it's not easy to understand overall result.
- Does not achieve maximum productivity due to some technical issues or any errors.

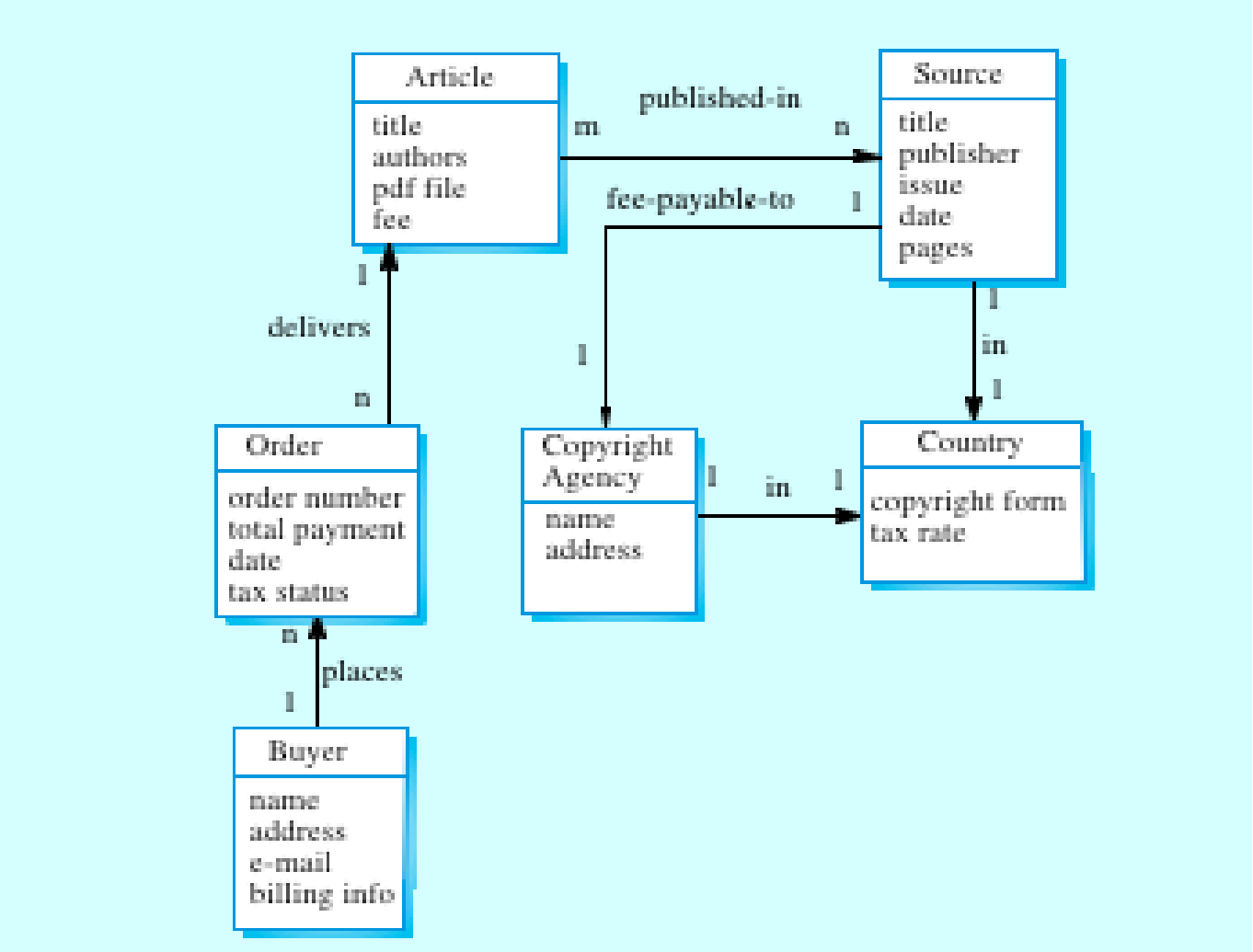
3.Data Models

Data Modeling in software engineering is the process of **simplifying the diagram or data model of a software system** by applying certain formal techniques. It involves expressing data and information through text and symbols. The data model provides the blueprint for building a new database or reengineering legacy applications.

i. Semantic data models

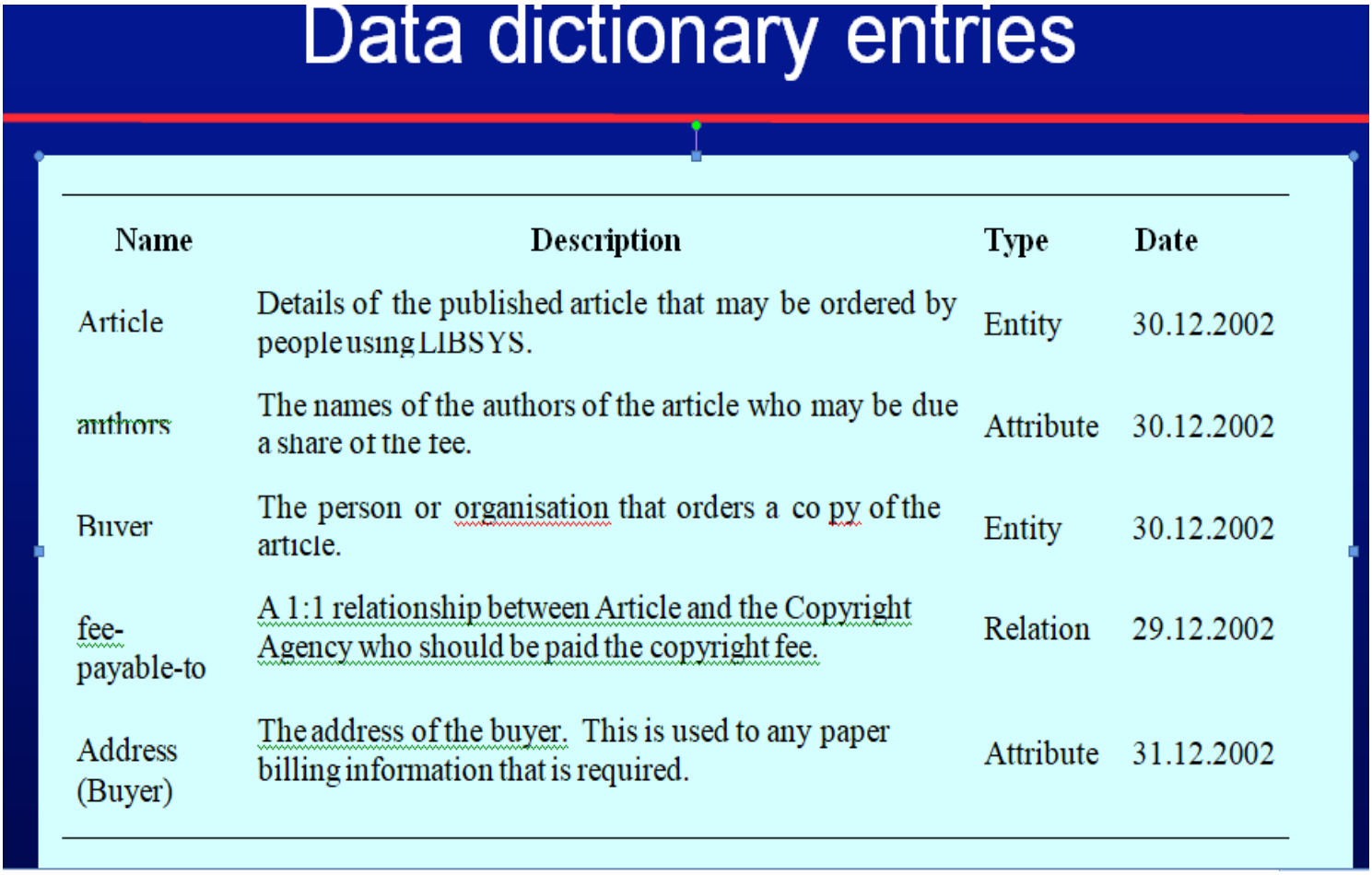
- Used to describe the logical structure of data processed by the system.
- An entity-relation-attribute model sets out the entities in the system, the relationships betweenthese entities and the entity attributes
- Widely used in database design. Can readily beimplemented using relational databases.
- No specific notation provided in the UML but objectsand associations can be used.

Library semantic model



Data dictionaries

- Data dictionaries are lists of all of the names used inthe system models. Descriptions of the entities, relationships and attributes are also included.
- Advantages
  - Support name management and avoid duplication;
  - Store of organisational knowledge linking analysis, designand implementation;



4. Object Models

- Object models describe the system in terms of object classes and their associations.
- An object class is an abstraction over a set of objects with common attributes and the services(operations) provided by each object.
- Natural ways of reflecting the real-worldentities manipulated by the system
- More abstract entities are more difficult tomodel using this approach
- Object class identification is recognized as adifficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities arereusable across systems
- Various object models may be produced
    - Inheritance models
    - Aggregation models
    - Object behavior models or Interaction models

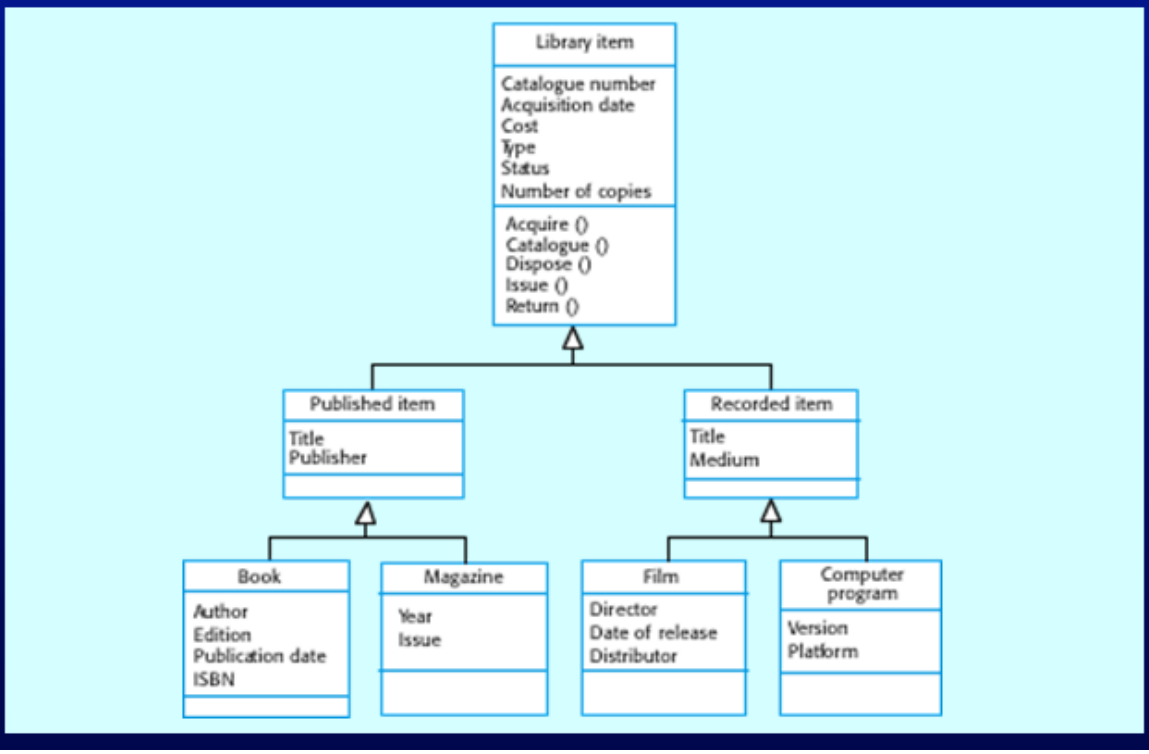
i. Inheritance models

- Organize the domain object classes into a hierarchy.
- Classes at the top of the hierarchy reflect the common features of all classes.
- Object classes inherit their attributes and services from one or more super-classes. These may then be specialized as necessary.
- Class hierarchy design can be a difficult process ifduplication in different branches is to be avoided.

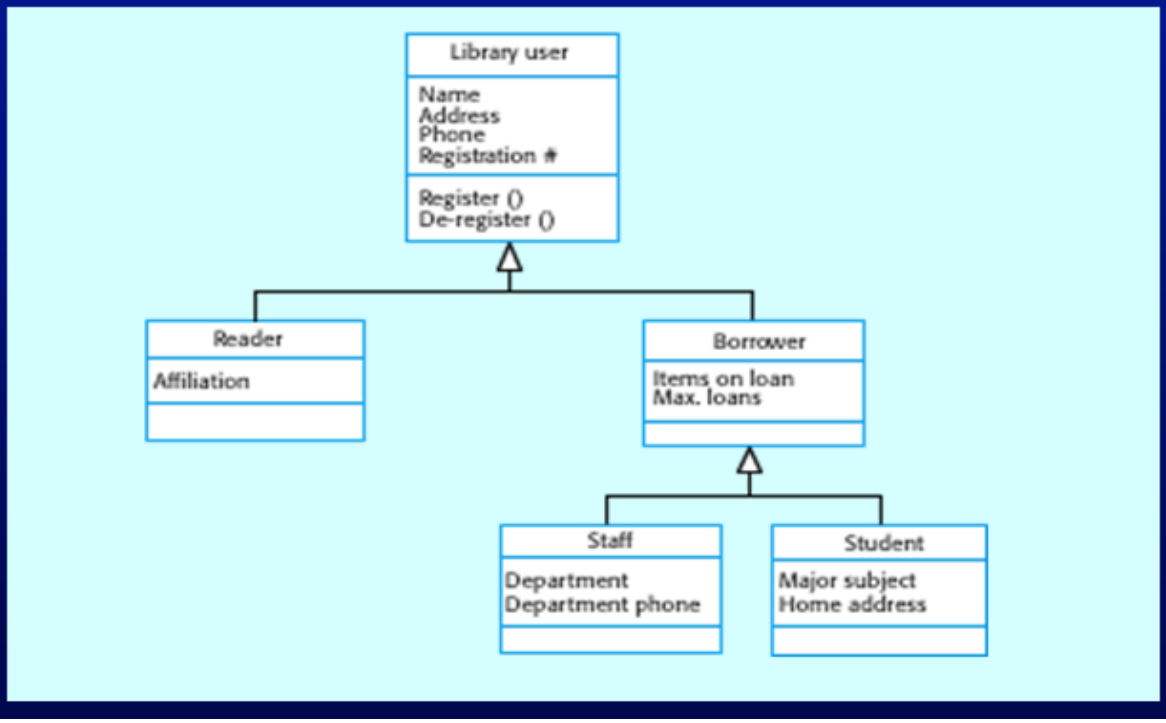
Object models and the UML

- The UML is a standard representation devised bythe developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modeling.
- Notation
    - Object classes are rectangles with the name at the top,attributes in the middle section and operations in the bottom section;
    - Relationships between object classes (known asassociations) are shown as lines linking objects;
    - Inheritance is referred to as generalization and is shown‘upwards’ rather than ‘downwards’ in a hierarchy.

Library class hierarchy



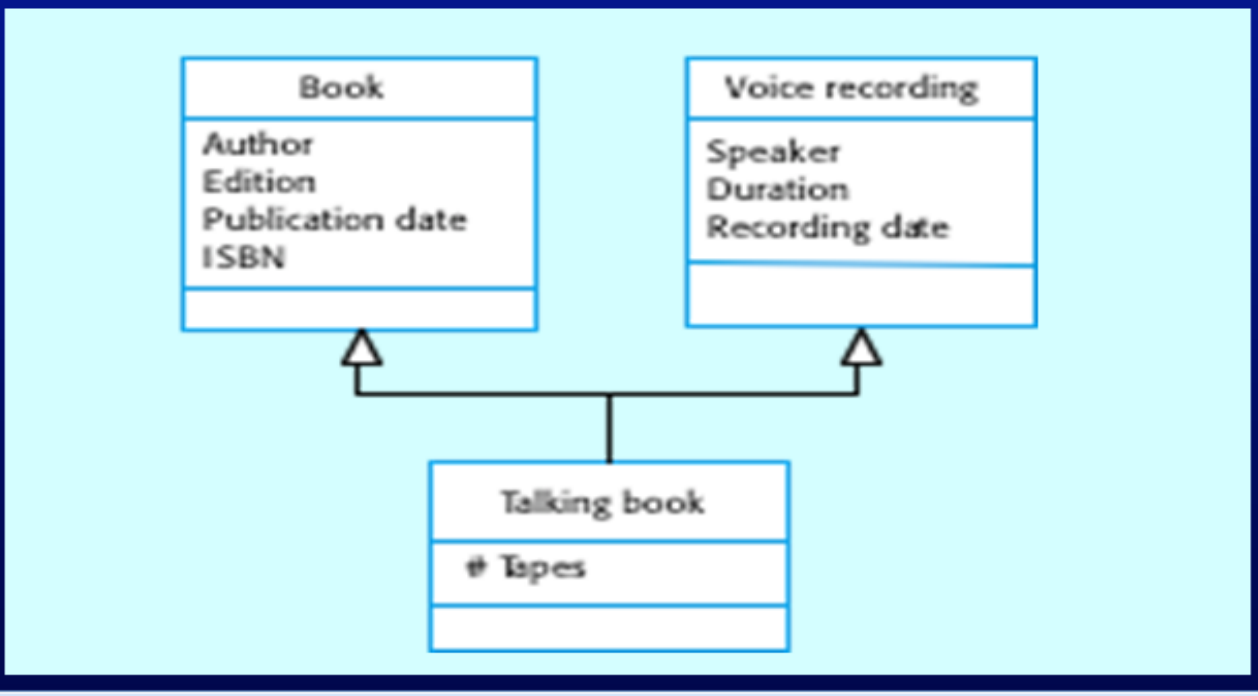
User class hierarchy



Multiple inheritance

- Rather than inheriting the attributes and services from a single parent class, a system which supports multiple inheritance allows object classes to inherit from several super-classes.
- This can lead to semantic conflicts where attributes/services with the same name in different super-classes have different semantics.
- Multiple inheritance makes class hierarchy reorganisation more complex.

Multiple inheritance

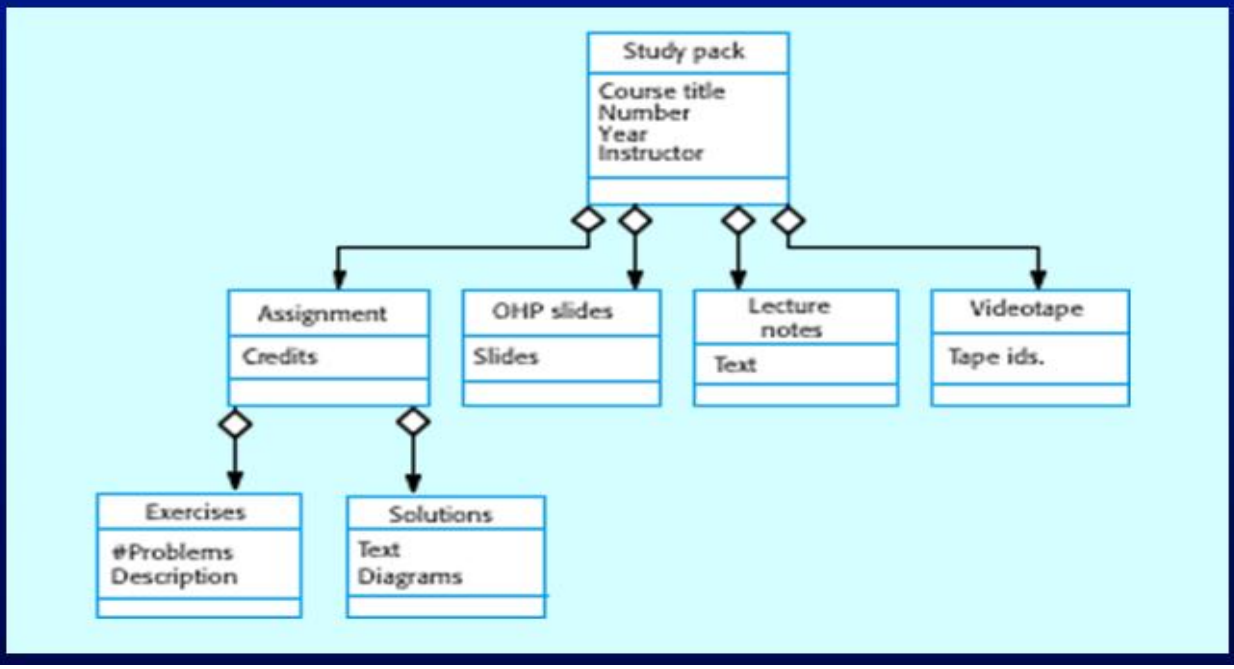


ii.Object Aggregation:

Object aggregation

- An aggregation model shows how classes that are collections are composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.

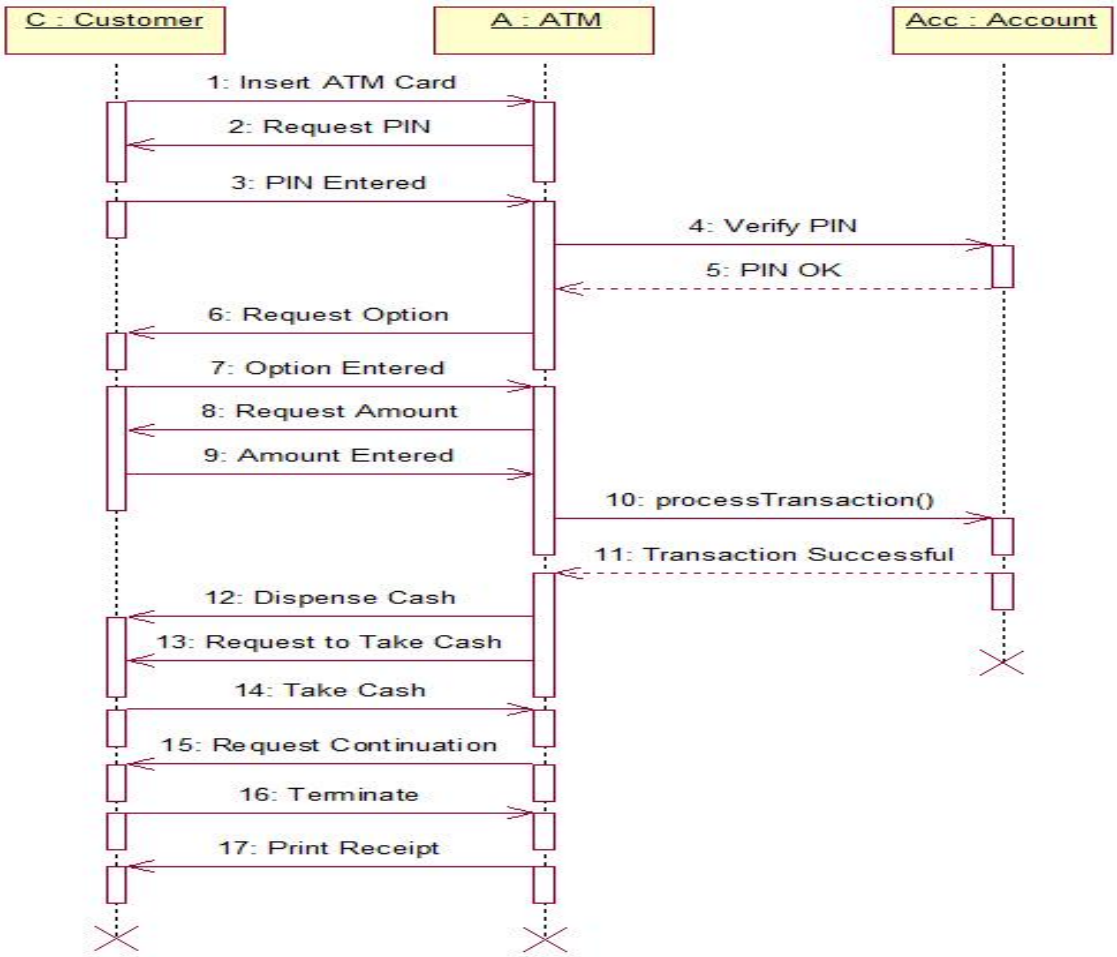
Object aggregation



iii. Object behavior modelling:

Object behaviour modelling

- A behavioural model shows the interactions between objects to produce some particular system behaviour that is specified as a use-case.
- Sequence diagrams (or collaboration diagrams) in the UML are used to model interaction between objects.



III -UNIT

Design Process , Design Quality and Design Principles:

7.2 Design Process and Design Quality

7.2.1 Design Process

- Design Process is a sequence of steps carried through which the user requirements are translated into a system or software model.
- The design is represented at high level of abstraction.

Following figure shows the design activities -

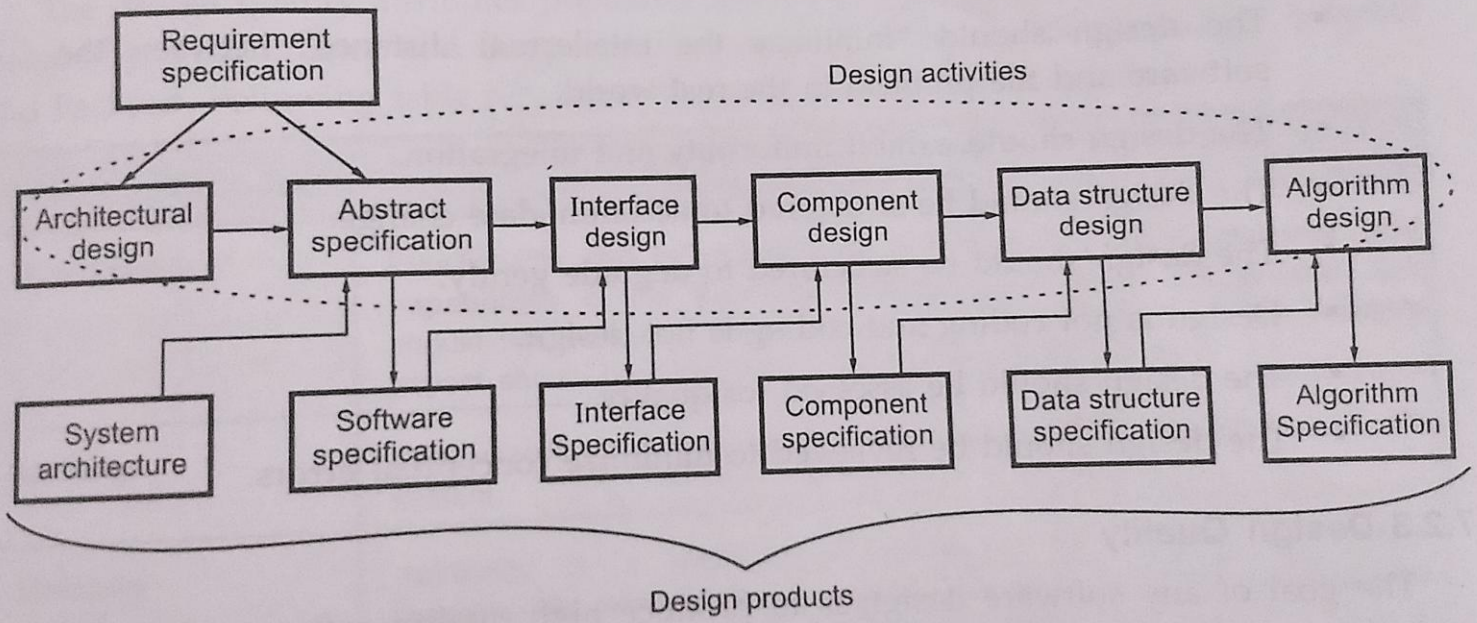


Fig. 7.2 Design Process

1. From requirements specification the architectural design and abstract specification is created. In architectural design the subsystem components can be identified. And the abstract specification is used to specify the subsystems.
2. Then the interfaces between the subsystems are designed which is called interface design.
4. In component design of subsystems components is done.
5. Once the components of the system are identified and designed the decision on use of particular data structures is taken. And the data structure is designed to hold the data.

6. For performing the required functionality, the appropriate algorithm is designed. These algorithms should suit the data structures selected for software product.

The design process occurs in iterations so that subsequent refinement is possible. At each design step the corresponding specifications are created. Throughout the design process the quality of evolving design is assessed with the help of formal technical reviews and design walkthroughs.

7.2.2 Design Principles

Davis suggested a set of principles for software design as:

- The design process should not suffer from "tunnel vision".
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance" between the software and the problem in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently.
- Design is not coding and coding is not design.
- The design should be assessed for quality.
- The design should be reviewed to minimize conceptual errors.

7.2.3 Design Quality

The goal of any software design is to produce high quality software. In order to evaluate quality of software there should be some predefined rules or criteria that need to be used to assess the software product. Such criteria serve as characteristics for good design. The quality guidelines are as follows –

1. The design architecture should be created using following issues –
   - The design should be created using **architectural styles and patterns.**
   - Each component of design should posses **good design characteristics**
   - The implementation of design should be **evolutionary**, so that testing can be performed at each phase of implementation.
2. In the design the data, architecture, interfaces and components should be clearly represented.
3. The design should be **modular**. That means the subsystems in the design should be logically partitioned.

4. The **data structure** should be appropriately chosen for the design of specific problem.
5. The **components** should be used in the design so that functional independency can be achieved in the design.
6. Using the information obtained in software **requirement analysis** the design should be created.
7. The **interfaces** in the design should be such that the complexity between the connected components of the system gets reduced. Similarly interface of the system with external interface should be simplified one.
8. Every design of the software system should convey its **meaning** appropriately and effectively.

7.2.4 Design Quality Attributes

The design quality attributes popularly known as **FURPS** (**F**unctionality, **U**sability, **R**eliability, **P**erformance and **S**upportability) is a set of criteria developed by Hewlett and Packard. Following table represents meaning of each quality attribute

Quality Attribute	Meaning
Functionality	Functionality can be checked by assessing the set of **features and capabilities** of the functions. The functions should be **general** and should not work only for particular set of inputs. Similarly the **security** aspect should be considered while designing the function.
Usability	The usability can be assessed by knowing the **usefulness** of the system.
Reliability	Reliability is a measure of frequency and severity of **failure**. Repeatability refers to the consistency and repeatability of the measures. The mean time to failure (**MTTF**) is a metric that is widely used to measure the product's performance and reliability.
Performance	It is a measure that represents the response of the system. Measuring the performance means measuring the processing speed, memory usage, response time and efficiency.
Supportability	It is also called maintainability. It is the ability to adopt the enhancement or changes made in the software. It also means the ability to withstand in a given environment.

Design Concepts:

7.3 Design Concepts

The software design concept provides a framework for implementing the right software.

Following issues are considered while designing the software –

1. Abstraction –

The abstraction means an ability to cope with the complexity. At each stage of software design process levels of abstractions should be applied to refine the software solution. At the higher level of abstraction, the solution should be stated in broad terms and in the lower level more detailed description of the solution is given.

While moving through different levels of abstraction the procedural abstraction and data abstraction are created.

The procedural abstraction gives the named sequence of instructions in the specific function. That means the functionality of procedure is mentioned by its implementation details are hidden. For example: *Search the Record* is a procedural abstraction in which implementation details are hidden(i.e. Enter the name, compare each name of the record against the entered one, if a match is found then declare success!! Otherwise declare 'name not found')

In data abstraction the collection of data objects is represented. For example for the procedure *search* the data abstraction will be *Record*.The record consists of various attributes such as Record ID, name, address and designation.

2. Modularity –

- The software is divided into separately named and addressable components that called as modules.
- Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

Modular decomposability : A design method provides a systematic mechanism for decomposing the problem into sub-problems. This reduces the complexity of the problem and the modularity can be achieved.

Modular composability : A design method enables existing design components to be assembled into a new system.

Modular understandability: A module can be understood as a stand alone unit. It will be easier to build and easier to change.

Modular continuity: Small changes to the system requirements result in changes to individual modules, rather than system-wide changes.

Modular protection: An aberrant condition occurs within a module and its effects are constrained within the module.

3. Architecture –

Architecture means representation of **overall structure** of an integrated system. In architecture various components interact and the data of the structure is used by various components. These components are called **system elements**. Architecture provides the basic framework for the software system so that important framework activities can be conducted in systematic manner.

In architectural design various system models can be used and these are

Model	Functioning
Structural model	Overall architecture of the system can be represented using this model
Framework model	This model shows the architectural framework and corresponding applicability.
Dynamic model	This model shows the reflection of changes on the system due to external events.
Process model	The sequence of processes and their functioning is represented in this model
Functional model	The functional hierarchy occurring in the system is represented by this model.

4. Refinement –

- Refinement is actually a process of elaboration.
- Stepwise refinement is a top-down design strategy proposed by Niklaus WIRTH.
- The architecture of a program is developed by successively refining levels of procedural detail.
- The process of program refinement is analogous to the process of refinement and partitioning that is used during requirements analysis.
- Abstraction and refinement are complementary concepts. The major difference is that - in the abstraction low-level details are suppressed. Refinement helps the designer to elaborate low-level details.

5. Pattern –

According to **Brad Appleton** the design pattern can be defined as – It is a named nugget(something valuable) of insight which conveys the essence of proven solution to a recurring problem within a certain context.

In other words, design pattern acts as a design solution for a particular problem occurring in specific domain. Using design pattern designer can determine whether-

- Pattern can be **reusable**
- Pattern can be used for **current work**
- Pattern can be used to **solve similar kind of problem** with different functionality.

6. Information hiding –

It is the characteristics of module in which major design decisions can be hidden from all others. Such hiding is necessary because information of one module cannot be accessed by another module. The advantage of information hiding is that modifications during testing and maintenance can be made independently without affecting the functionality of other modules.

7. Functional independence –

Functional independence can be achieved by modularity, abstraction and information hiding. Functional independence obtained by creating each module that is performing only one specific task. And such a module should be interconnected to other components by a simple interface. Effective modularity helps in achieving the functional independence. In short, Functional independence is a key to good design and good design leads a quality in software product. The functional independence is assessed using two factors cohesion and coupling.

Cohesion

- With the help of cohesion the information hiding can be done.
- A cohesive module performs only "one task" in software procedure with little interaction with other modules. In other words cohesive module performs only one thing.
- Different types of cohesion are :

1. **Coincidentally cohesive** – The modules in which the set of tasks are related with each other loosely then such modules are called coincidentally cohesive.
2. **Logically cohesive** – A module that performs the tasks that are logically related with each other is called logically cohesive.

3. **Temporal cohesion** – The module in which the tasks need to be executed in some specific time span is called temporal cohesive.
4. **Procedural cohesion** – When processing elements of a module are related with one another and must be executed in some specific order then such module is called procedural cohesive.
5. **Communicational cohesion** – When the processing elements of a module share the data then such module is communicational cohesive.

- The goal is to achieve high cohesion for modules in the system.

Coupling

- Coupling effectively represents how the modules can be "connected" with other module or with the outside world.
- Coupling is a measure of interconnection among modules in a program structure.
- Coupling depends on the interface complexity between modules.
- The goal is to strive for lowest possible coupling among modules in software design.
- The property of good coupling is that it should reduce or avoid change impact and ripple effects. It should also reduce the cost in program changes, testing, and maintenance.

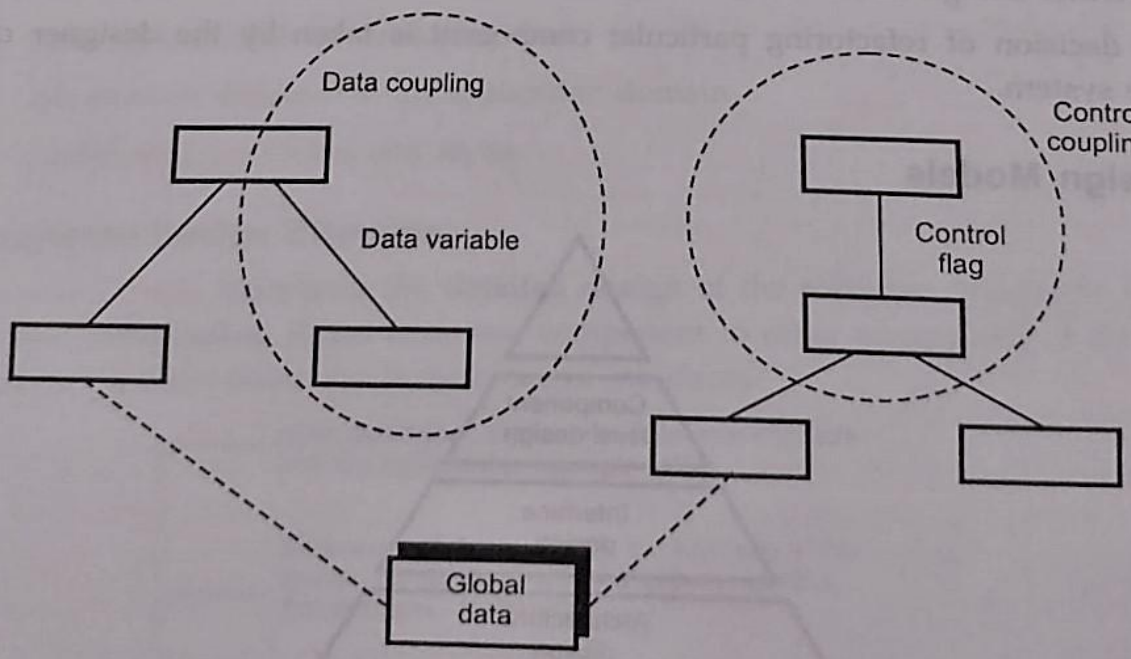


Fig. 7.3 Coupling

- Various types of coupling are :
  i) **Data coupling** – The data coupling is possible by parameter passing or data interaction.
  ii) **Control coupling** – The modules share related control data in control coupling.
  iii) **Common coupling** – In common coupling common data or a global data is shared among the modules.
  iv) **Content coupling** – Content coupling occurs when one module makes use of data or control information maintained in another module.

8. Refactoring –

Refactoring is necessary for simplifying the design without changing the function or behaviour. Fowler has defined refactoring as *the process of changing a software system in such a way that the external behaviour of the design do not get changed, however the internal structure gets improved.*

Benefits of refactoring are –

- The **redundancy** can be achieved.
- **Inefficient algorithms** can be eliminated or can be replaced by efficient one.
- Poorly constructed or **inaccurate data structures** can be removed or replaced.
- Other **design failures** can be rectified.

The decision of refactoring particular component is taken by the designer of the software system.

Design Models:

7.4 Design Models

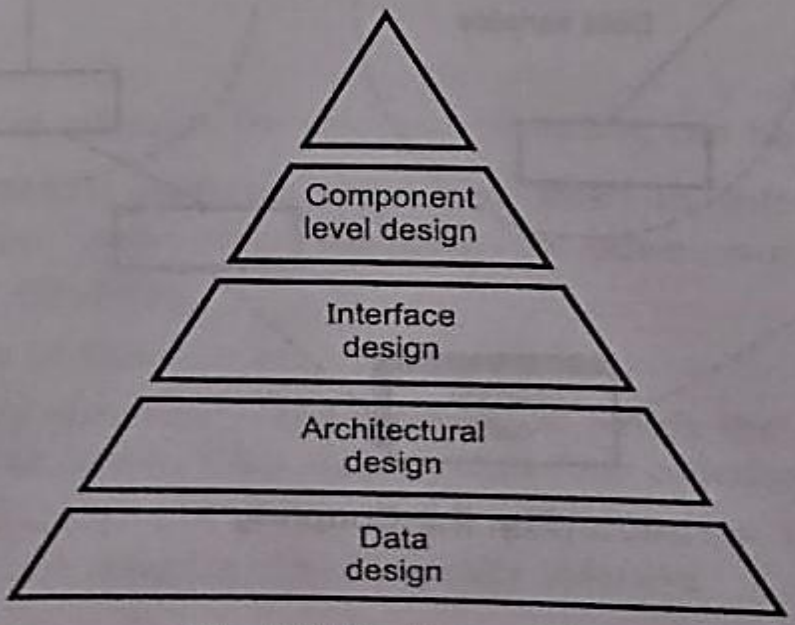


Fig. 7.4 Design model

- The design model is represented as pyramid. The pyramid is a stable object. Representing design model in this way means that the software design should be stable.
- The design model has broad foundation of data design, stable mid-region with architectural and interface design and the sharp point to for component level design.
- The design model represents that the software which we create should be stable such that any changes should not make it collapsed. And from such a stable design a high quality software should be generated.

7.4.1 Data Design Element

The data design represents the **high level of abstraction**. This data represented at data design level is **refined gradually** for implementing the computer based system. The data has great impact on the architecture of software systems. Hence structure of data is very important factor in software design. Data appears in the form of **data structures and algorithms** at the program **component level**. At the **application level** it appears as the **database** and at the **business level** it appears as **data warehouse and data mining**. Thus data plays an important role in software design.

7.4.2 Architectural Design Element

The architectural design gives the layout for overall view of the software. Architectural model can be built using following sources –

- Data flow models or class diagrams
- Information obtained from application domain
- Architectural patterns and styles.

7.4.3 Interface Design Elements

Interface Design represents the **detailed design** of the software system. In interface design how **information flows** from one component to other component of the system is depicted. Typically there are three types of interfaces-

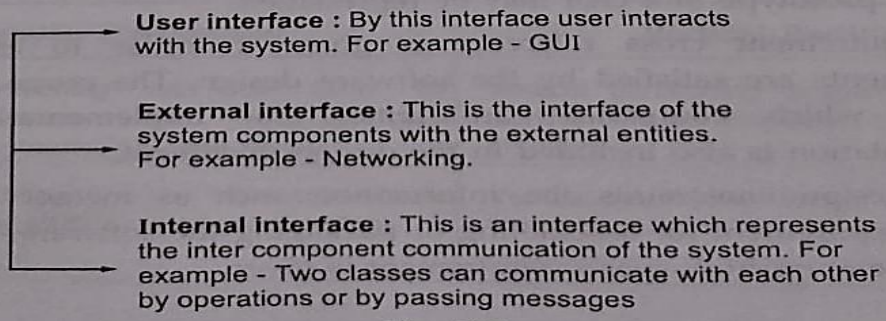


Fig. 7.5

7.4.4 Component Level Design Elements

The component level design is more detailed design of the software system along with the specifications. The component level design elements describe the internal details of the component. In component level design all the local data objects, required data structures and algorithmic details and procedural details are exposed.

8.2 Software Architecture

The architectural design is the design process for identifying the subsystems making up the system and framework for subsystem control and communication.

The goal of architectural design is to establish the overall structure of software system. Architectural design represents the link between design specification and actual design process.

Software Architecture is a structure of systems which consists of various components, externally visible ***properties*** *of these components and the inter-****relationship*** *among these components*

Importance of Software Architecture

There are three reasons why the software architecture is so important?

1. Software architecture gives the representation of the computer based system that is to be built. Using this system model even the **stakeholders** can take **active part** in the software development process. This helps in clear specification/understanding of requirements.
2. Some **early design decisions** can be taken using software architecture and hence system performance and operations remain under control.
3. The software architecture gives a clear cut **idea** about the computer based system which is to be built.

8.2.1 Structural Partitioning

The program structure can be partitioned horizontally or vertically.

Horizontal partitioning

Horizontal partitioning defines separate branches of the modular hierarchy for each major program function.

Horizontal partitioning can be done by partitioning system into : input, data transformation (processing) and output.

In horizontal partitioning the design making modules are at the top of the architecture.

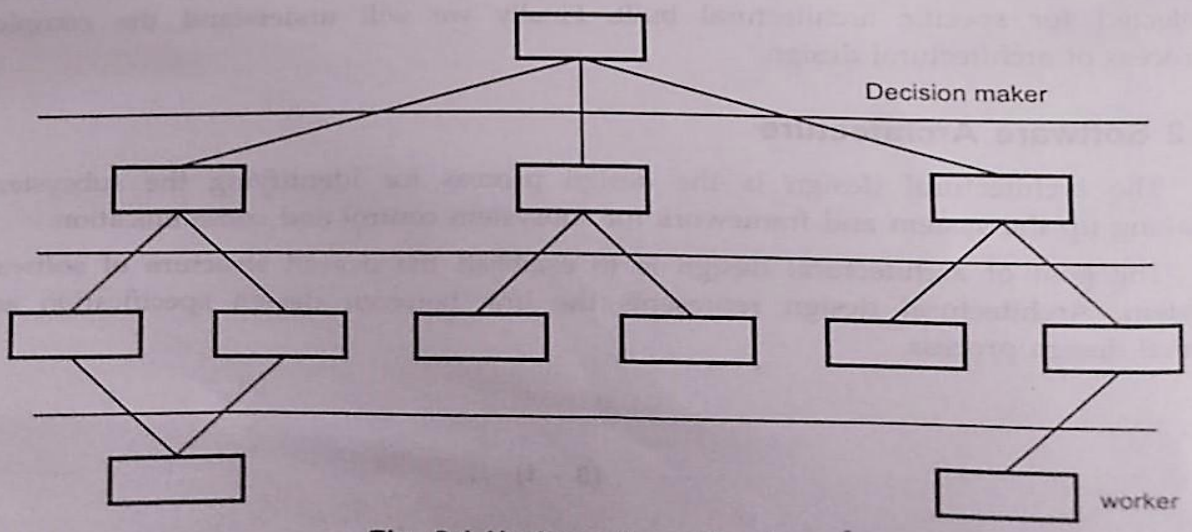


Fig. 8.1 Horizontal partitioning

Advantages of horizontal partition

1. These are easy to test, maintain and extend.
2. They have fewer side effects in change propagation or error propagation.

Disadvantage of horizontal partition

More data has to be passed across module interfaces which complicate the overall control of program flow.

Vertical partitioning

Vertical partitioning suggests the control and work should be distributed top-down in program structure.

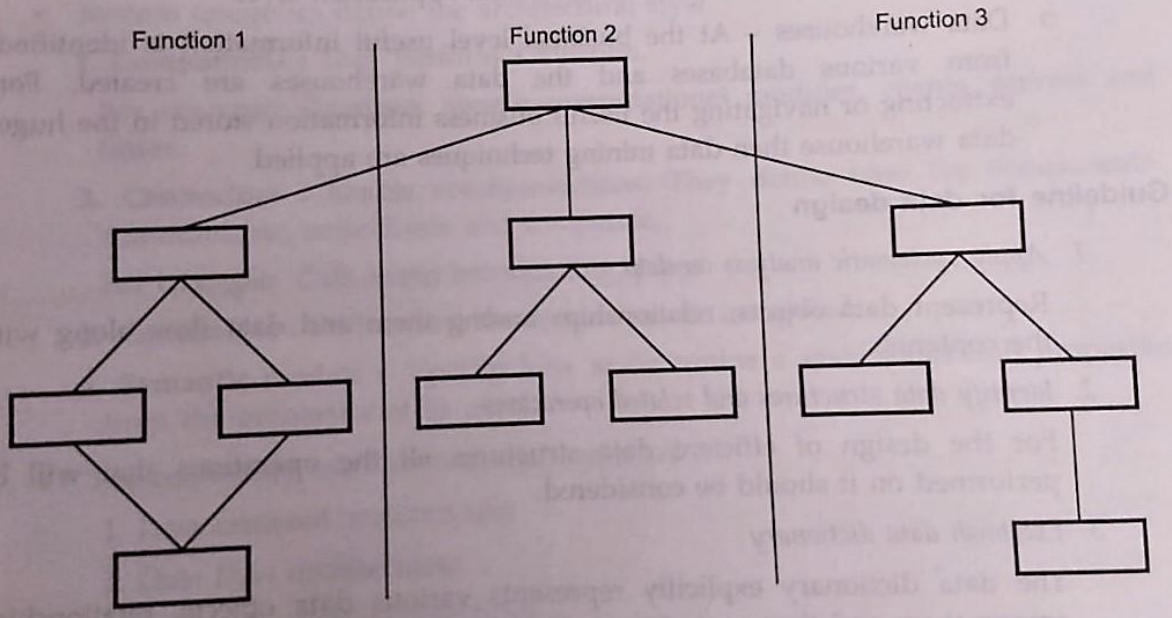


Fig. 8.2 Vertical partitioning

In vertical partitioning

- Define separate branches of the module hierarchy for each major function.
- Use control modules to co-ordinate communication between functions.

Advantages of vertical partition

1. These are easy to maintain the changes.
2. They reduce the change impact and error propagation.

8.3 Data Design

- Data design is basically the model of data that is represented at the high level of abstraction.
- The data design is then progressively refined to create implementation specific representations.
- Various elements of data design are
  - Data object – The data objects are identified and relationship among various data objects can be represented using entity relationship diagrams or data dictionaries.
  - Databases – Using software design model, the data models are translated into data structures and databases at the application level.
  - Data warehouses – At the business level useful information is identified from various databases and the data warehouses are created. For extracting or navigating the useful business information stored in the huge data warehouse then data mining techniques are applied.

Guideline for data design

1. *Apply systematic analysis on data*

   Represent data objects, relationships among them and data flow along with the contents.

2. *Identify data structures and related operations*

   For the design of efficient data structures all the operations that will be performed on it should be considered.

3. *Establish data dictionary*

   The data dictionary explicitly represents various data objects, relationships among them and the constraints on the elements of data structures.

4. *Defer the low-level design decisions until late in the design process*

   Major structural attributes are designed first to establish an architecture of data. And then low-level design attributes are established.

5. *Use information hiding in the design of data structures*

   The use of information hiding helps in improving quality of software design. It also helps in separating the logical and physical views.

6. *Apply a library of useful data structures and operations*

   The data structures can be designed for reusability. A use of library of data structure templates (called as abstract data types) reduces the specification and design efforts for data.

7. *Use a software design and programming language to support data specification and abstraction*

   The implementation of data structures can be done by effective software design and by choosing suitable programming language.

8.4 Architectural Styles and Pattern

8.4.1 Architectural Styles

- The **architectural model or style** is a pattern for creating the system architecture for given problem. However, most of the large systems are heterogeneous and do not follow single architectural style.
- System categories define the architectural style

  1. Components : They perform a function.

  For example: Database, simple computational modules, clients, servers and filters.

  2. Connectors : Enable communications. They define how the components communicate, co-ordinate and co-operate.

  For example Call, event broadcasting, pipes

  3. Constraints : Define how the system can be integrated.

  4. Semantic models : Specify how to determine a system's overall properties from the properties of its parts.
- The commonly used architectural styles are
  1. Data centered architectures
  2. Data flow architectures
  3. Call and return architectures
  4. Object oriented architectures
  5. Layered architectures

8.4.1.1 Data Centered Architectures

In this architecture the data store lies at the centre of the architecture and other components frequently access it by performing add, delete and modify operations. The client software requests for the data to central repository. Sometime the client software accesses the data from the central repository without any change in data or without any change in actions of software actions.

Data centered architecture posses the property of interchangeability. Interchangeability means any component from the architecture can be replaced by a new component without affecting the working of other components.

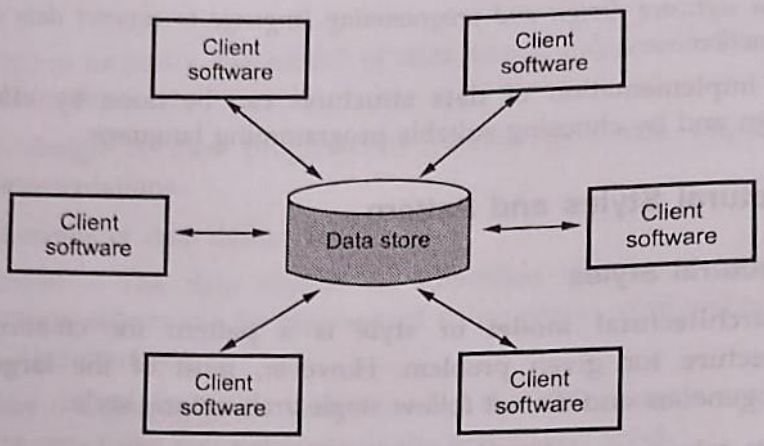


Fig. 8.3 Data centered architecture

In data centered architecture the data can be passed among the components.

In data centered architecture

Components are : Database elements such as tables, queries.

Communication are : By relationships

Constraints are : Client software has to request central data store for information.

8.4.1.2 Data Flow Architectures

In this architecture series of transformations are applied to produce the output data. The set of components called filters are connected by pipes to transform the data from one component to another. These filters work independently without a bothering about the working of neighbouring filter.

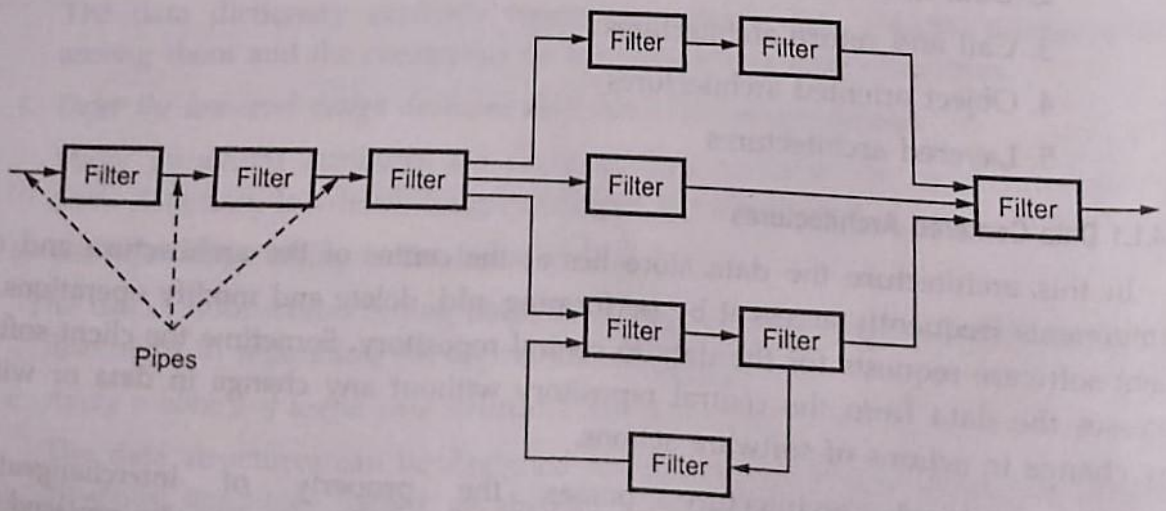


Fig. 8.4 Pipes and Filters

If the data flow degenerates into a single line of transforms, it is termed as batch sequential.

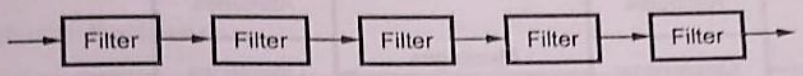


Fig. 8.5 Batch sequential

In this pattern the transformation is applied on the batch of data.

8.4.1.3 Call and Return Architecture

The program structure can be easily modified or scaled. The program structure is organized into modules within the program. In this architecture how modules call each other. The program structure decomposes the function into control hierarchy where a main program invokes number of program components.

In this architecture the hierarchical control for call and return is represented.

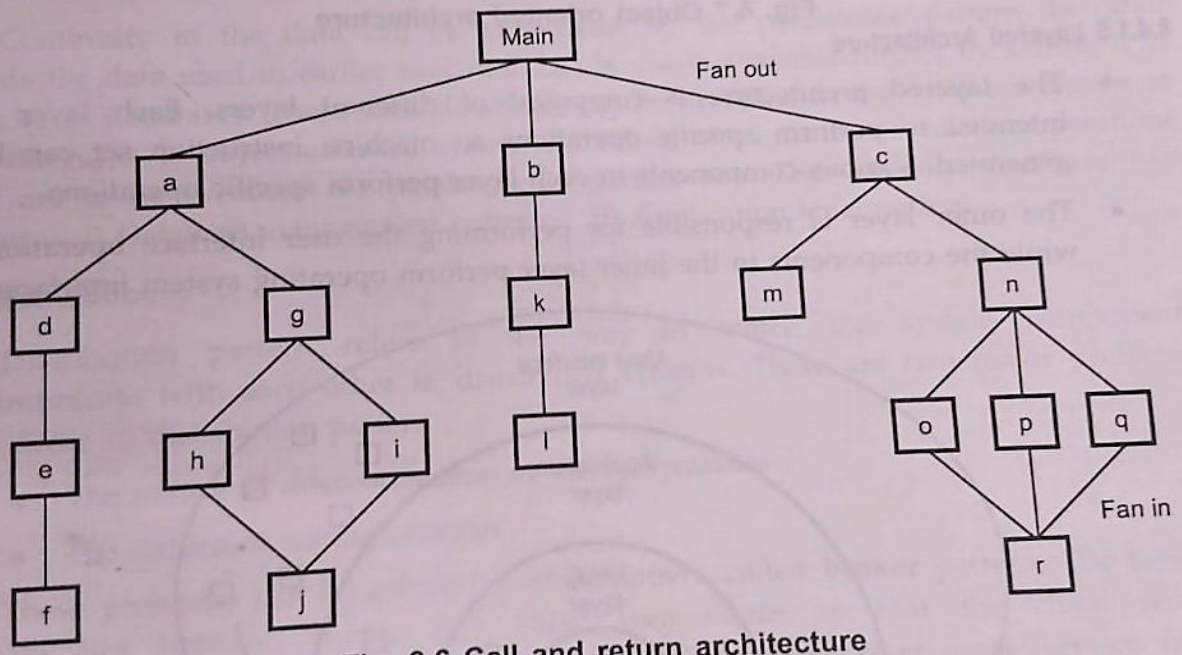


Fig. 8.6 Call and return architecture

8.4.1.4 Object Oriented Architecture

In this architecture the system is decomposed into number of interacting objects.

These objects encapsulate data and the corresponding operations that must be applied to manipulate the data.

The object oriented decomposition is concerned with identifying objects classes, their attributes and the corresponding operations. There is some control models used to co-ordinate the object operations.

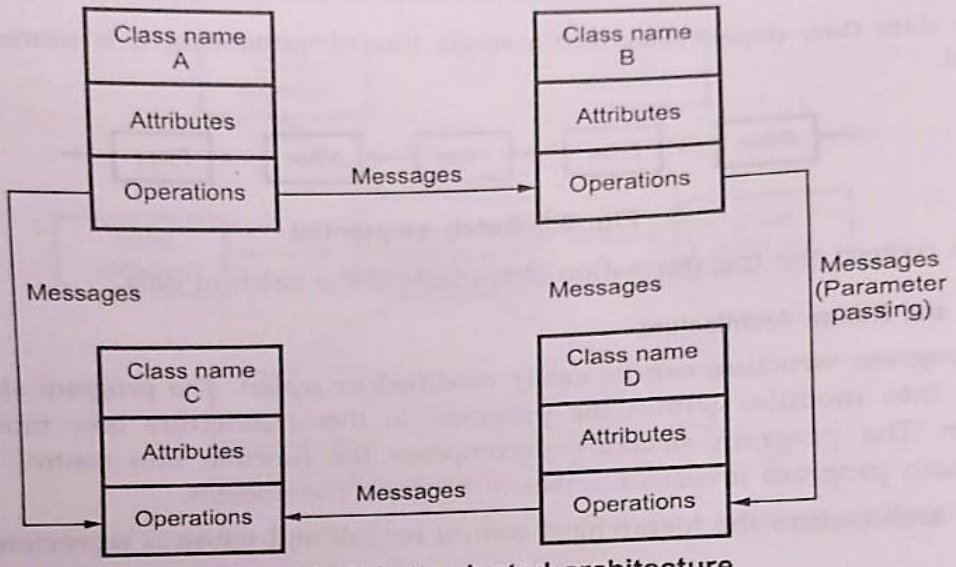


Fig. 8.7 Object oriented architecture

8.4.1.5 Layered Architecture

- The layered architecture is composed of different layers. Each layer is intended to perform specific operations so machine instruction set can be generated. Various components in each layer perform specific operations.
- The outer layer is responsible for performing the user interface operations while the components in the inner layer perform operating system interfaces.

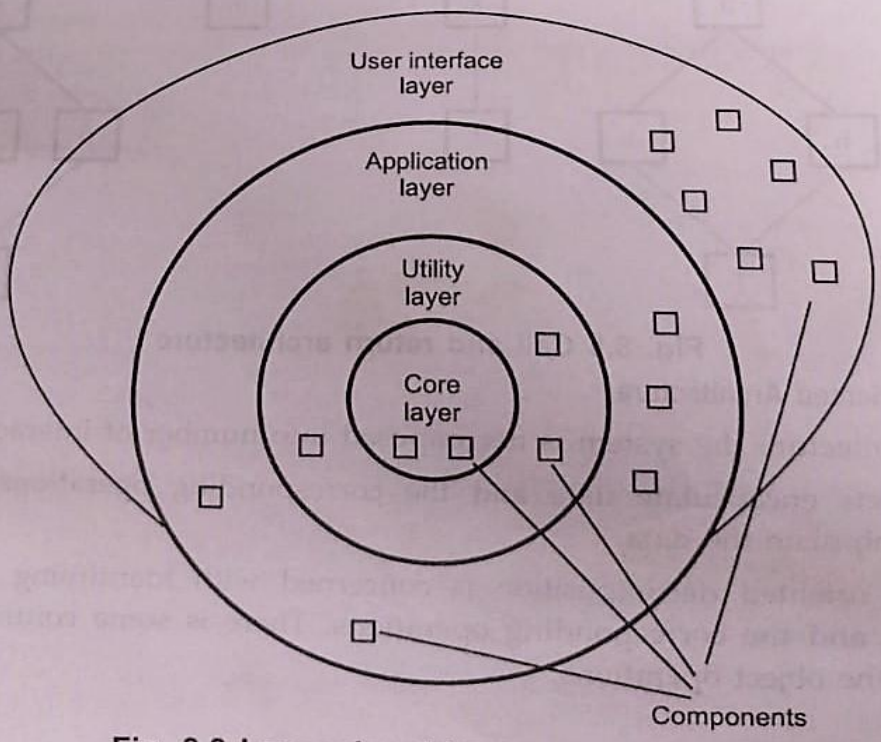


Fig. 8.8 Layered architecture components

- The components in intermediate layer perform utility services and application software functions.

8.4.2 Architectural Patterns

In this section we will understand *what are architectural patterns* ? The architectural pattern is basically an approach for handling behavioural characteristics of software systems. Following are the architectural pattern domains

1. Concurrency

Concurrency means handling multiple tasks in parallel. For example in operating system, **multiple tasks** are executed in parallel. Hence concurrency is a pattern which represents that the system components can interact with each other in parallel. The benefit of this pattern is that system **efficiency** can be achieved.

2. Persistence

Continuity in the data can be maintained by the persistence pattern. In other words the data used in earlier execution can be made available further by storing it in files or in **databases.** These files/databases can be modified in the software system as per the need. In **object oriented** system the values of all attributes various operations that are to be executed are persistent for further use. Thus broadly there are two patterns. i) Database management pattern ii) Application level pattern.

3. Distribution

Distribution pattern refers to the way in which the system components communicate with each other in distributed systems. There are two major problems that occur in distribution pattern

- The nature of interconnection of the components
- The nature of communication

These problems can be solved by other pattern called **broker pattern.** The broker pattern lies between server and client components so that the client server communication can be established properly. When client want some service from server, it first sends message to broker. The broker then conveys this message to server and completes the connection. Typical example is CORBA. The CORBA is a distributed architecture in which broker pattern is used.

8.5 Architectural Design

In architectural design at the initial stage a **context model** is prepared. This model defines the external entities that interact with the software. Along with this model the nature of software interaction with external entities is also described. The context model is prepared by using information obtained from analysis model and requirement specification. After that the designer creates **structure of the system** by defining and refining software components. Thus process of creations of context model and structural model of the system is iteratively carried out until and unless complete architectural model of the system gets created. Let us discuss how an architectural design gets generated using some simple representations.

8.5.1 Representing System in Context

We have already discussed in chapter 6, how to create a context model for the given software system. As per our discussion, context model is a graphical model in which the environment of the system is defined by showing the external entities that interact with the software system.

In architectural design the Architectural Context Diagram(ACD) is created. The difference between context model and architectural context diagram is that in ACD the nature of interaction is clearly described. Following are the basic terminologies associated with architectural context diagram.

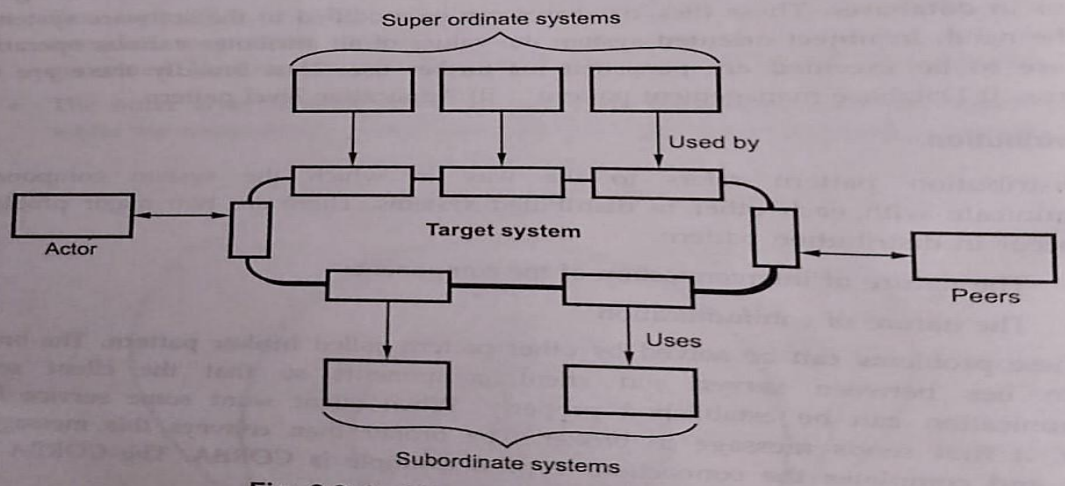


Fig. 8.9 Architectural context diagram

Target system : The target system is a computer based system for which the architectural context diagram has to be prepared.

Super ordinate systems : These systems are created at higher level of processing when the target system is being developed.

Sub ordinate systems : These systems are used by the target system for processing of the data that are necessary to complete target system functionality

Actors : These are the systems or entities that interact with the target system for producing or consuming information of the target system.

Peer-level systems: These are the systems that interact on peer to peer basis.

For example

Consider the *Inventory Control System* for which the Architectural Context Diagram can be prepared as below

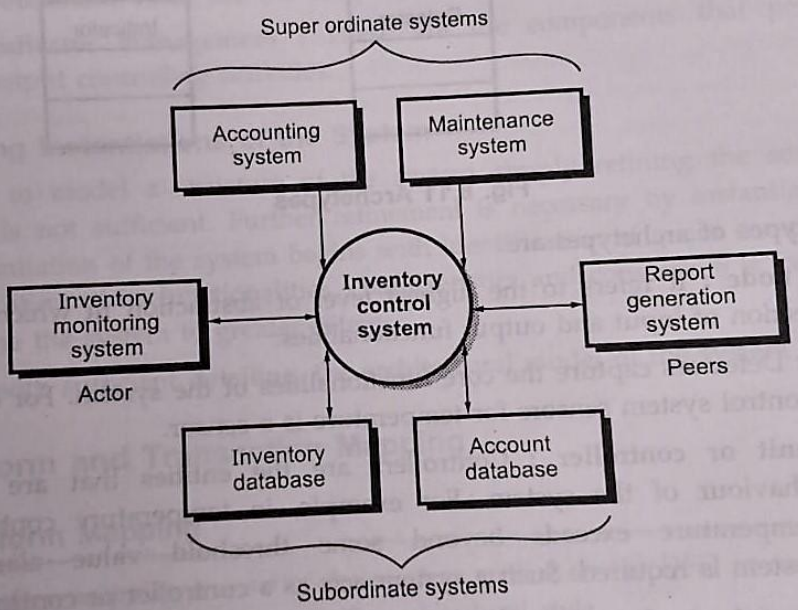


Fig. 8.10 ACD for inventory control system

8.5.2 Defining Archetypes

Defining archetype is a **basic step** in architectural design, more precisely in functionality based design. Archetype is a **core abstraction** using which the system can be structured. Using archetypes, a **small set** of entities that describe the major part of system behaviour can be described. Typically archetypes are the stable elements and they do not change even though system undergo through various changes. Identifying archetypes is a critical task and it requires well experienced architect. Following figure represents various archetypes.

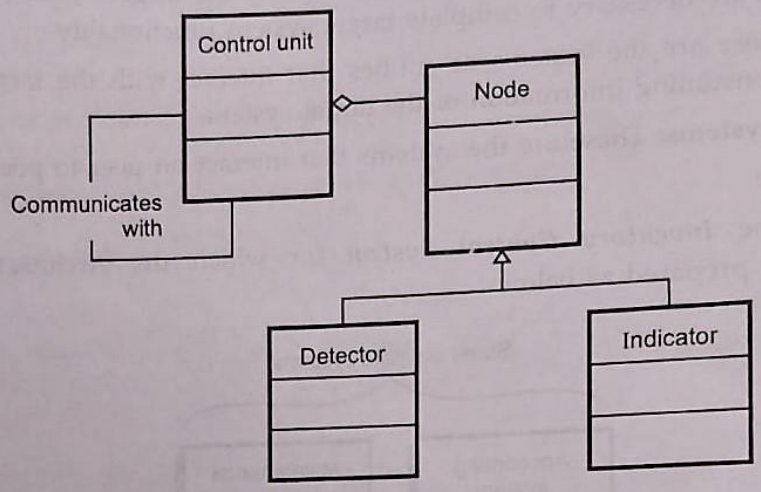


Fig. 8.11 Archetypes

Various types of archetypes are

Point or node : It refers to the highest level of abstraction in which there is a cohesive collection of input and output functionalities.

Detector : Detectors capture the core functionalities of the system. For example, in temperature control system sensors for temperature is a sensor.

Control unit or controller : Controllers are the entities that are useful for controlling behaviour of the system. For example, in temperature control system. When the temperature exceeds beyond some threshold value alarming and dis-alarming system is required. Such a system acts as a controller or control unit.

Indicator or output : It represents the generic output functionalities. For example, monitoring system of any computer based system acts as an indicator.

In software engineering archetype is a number of **major components** that are used to describe the system which we want to build.

8.5.3 Refining Architecture into Components

To create full structure of the system it is required to refine the software architecture into components. Hence it is necessary to identify the components of the system. The components can be identified from **application domain** or from **infrastructure domain**. The architectural designer has to identify these components from these domains. There are two methods by which the components can be identified.

1. The **data flow diagram** is drawn from which the specialized components can be identified. Such components are the components that process the data flow across the interfaces.
2. The components can be the entities that follow following **functionalities** –

   External communication : The components that take part in the communication with external entities are the communication components.

   Control panel processing : These components perform all control panel management activities.

   Detection : These are the components that perform detection activities.

   Indicator management : These are the components that perform the output controlling activities.

8.5.4 Defining Instantiations of the System

In order to model a structure of the system simply refining the software into components is not sufficient. Further refinement is necessary by instantiation of the system. Instantiation of the system begins with identification of major components and then identification of its functionalities, characteristics and constraints is carried out in order to refine the system to greater extent.

Finally with sufficient detailing, the architectural model of the system gets ready.

Unified Modeling Language(UML):

Unified Modeling Language (UML) is a graphical language, used in object-oriented development that includes a several types of system model and provides different views of a system.

Unified Modeling Language (UML) is **a standardized modeling language**. It helps software developers visualize, construct, and document new software systems and blueprints.

UML is a visual modeling language. “A picture is worth a thousand words.”

UML is a standard language for software blueprints.”

UML combined the best from object-oriented software modeling methodologies that were in existence during the early 1990’s. – Grady Booch, James Rumbaugh, and Ivor Jacobson are the primary contributors to UML.

1. Model:

A model is a simplification of reality.

It is a representation of a subject or objects.

It captures a set of ideas *(known as abstractions or concepts)* about its subject.

We build models so that we can better understand the system we are developing.

Through modeling, we achieve four aims

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

2. Unified:

It is to bring together the information systems and technology industry's best engineering practices.

These practices involve applying techniques that allow us to successfully develop systems.

3. Language:

It enables us to communicate about a subject which includes the requirements and the system.

It is difficult to communicate and collaborate for a team to successfully develop a system without a language.

Usages of UML:

UML is used to:

i. UML is a language for documenting design – Provides a record of what has been built. – Useful for bringing new programmers up to speed.

ii. Represent different views/aspects of design – visualize and construct designs -----static / dynamic / deployment / modular aspect.

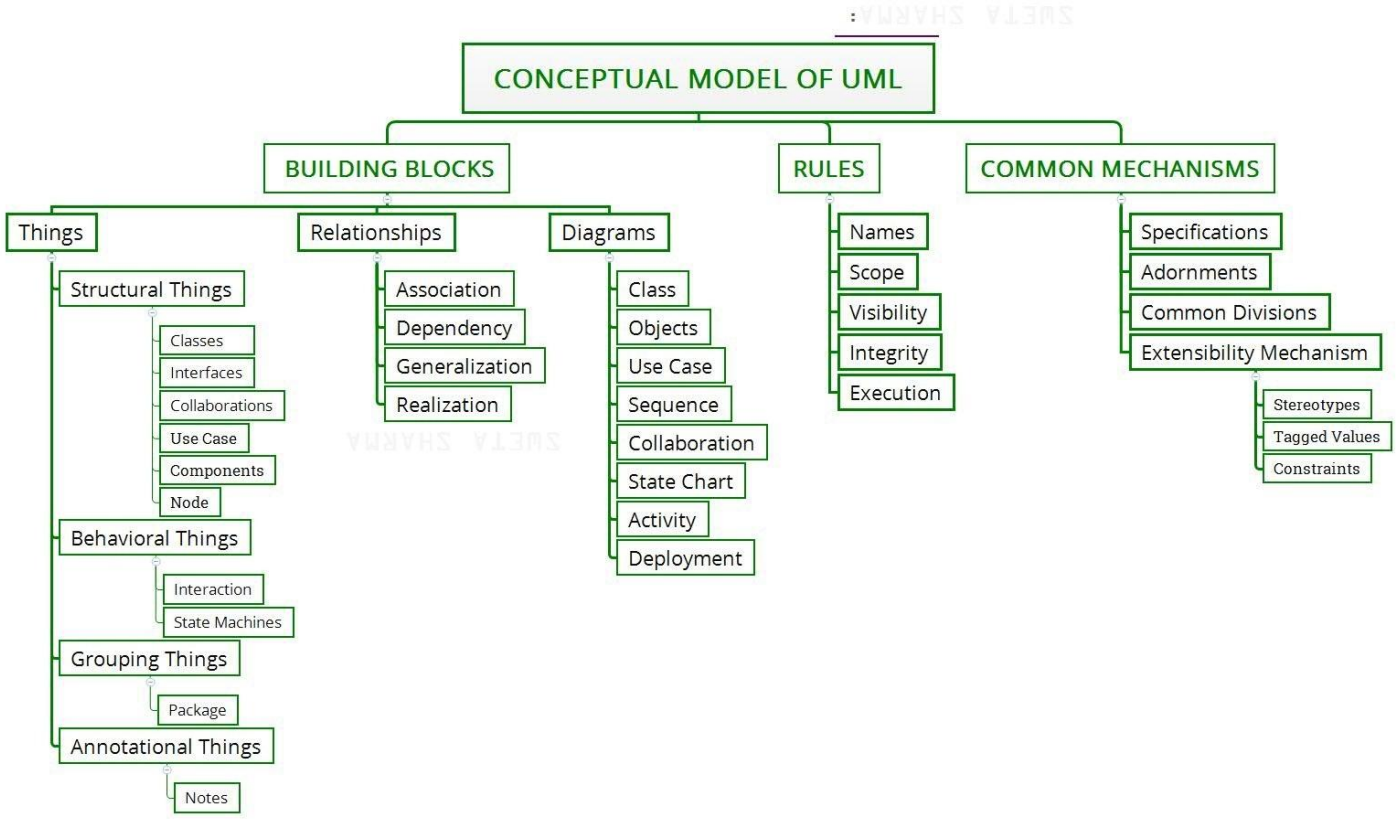
iii. Provide a next-to-precise, common, language –specify visually

iv. To present a simplified view of reality in order to facilitate the design and implementation of object-oriented software systems

A Conceptual Model of UML:

A conceptual model of the language underlines the three major elements:

- The Building Blocks
- The Rules
- Some Common Mechanisms



I). Building blocks of UML (Conceptual model of UML)

These are the fundamental elements in UML. Every diagram can be represented using these building blocks. The building block of UML contains three types of elements. They are:

1) Things (object oriented parts of uml)I

2) Relationships (relational parts of uml)

3) Diagrams

1.Things

Things are the abstractions. A diagram can be viewed as a graph containing vertices and edges. In UML, vertices are replaced by things, and the edges are replaced by relationships. There are four types of things in UML. They are:

a) Structural things (nouns of uml – static parts)

b) Behavioral things (verbs of uml – dynamic parts)

c) Grouping things (organizational parts)

d) Annotational things (explanatory parts)

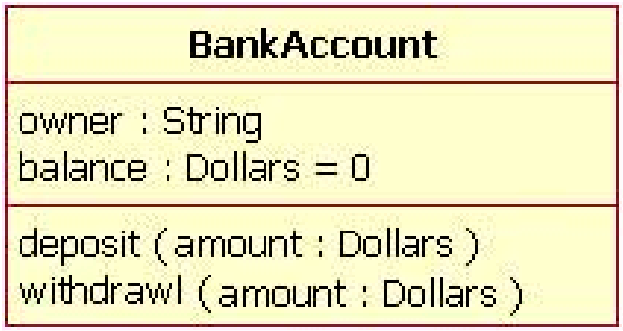
a)Structural things

Represents the static aspects of a software system. There are seven structural things in UML. They are:

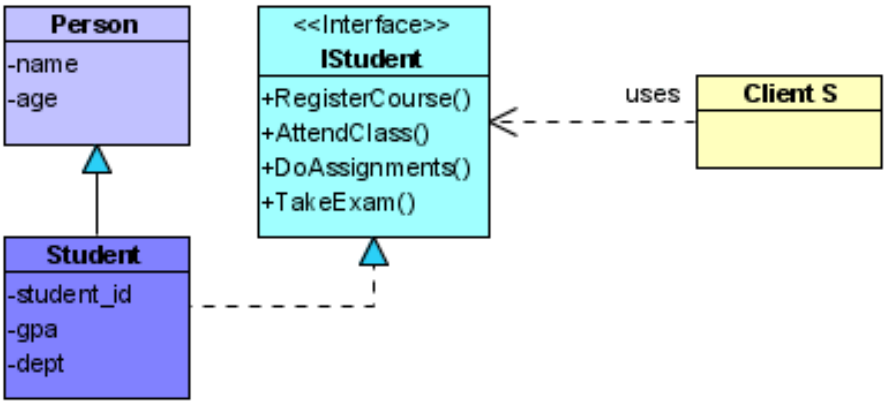
Class: A class is a collection of similar objects having similar attributes, behavior, relationships and semantics. Graphically class is represented as a rectangle with three compartments.

Graphical representation:

Example:



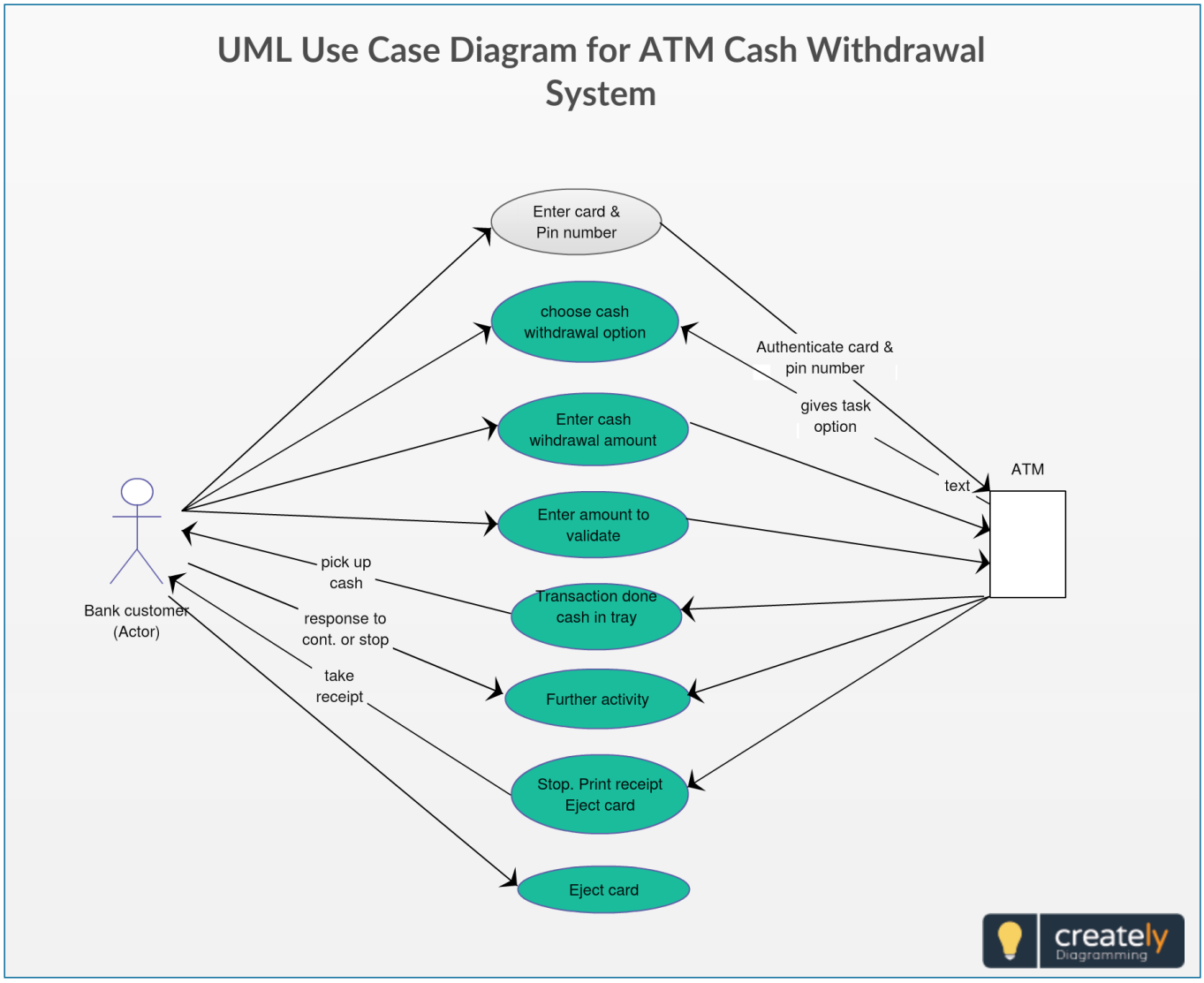
Interface: An interface is a collection of operation signatures and/or attribute definitions that ideally define a cohesive set of behavior. Graphically interface is represented as a circle or a class symbol stereotyped with interface.



Use Case: A use case is a collection of actions, defining the interactions between a role (actor) and the system. Graphically use case is represented as a solid ellipse with its name written inside or below the ellipse.

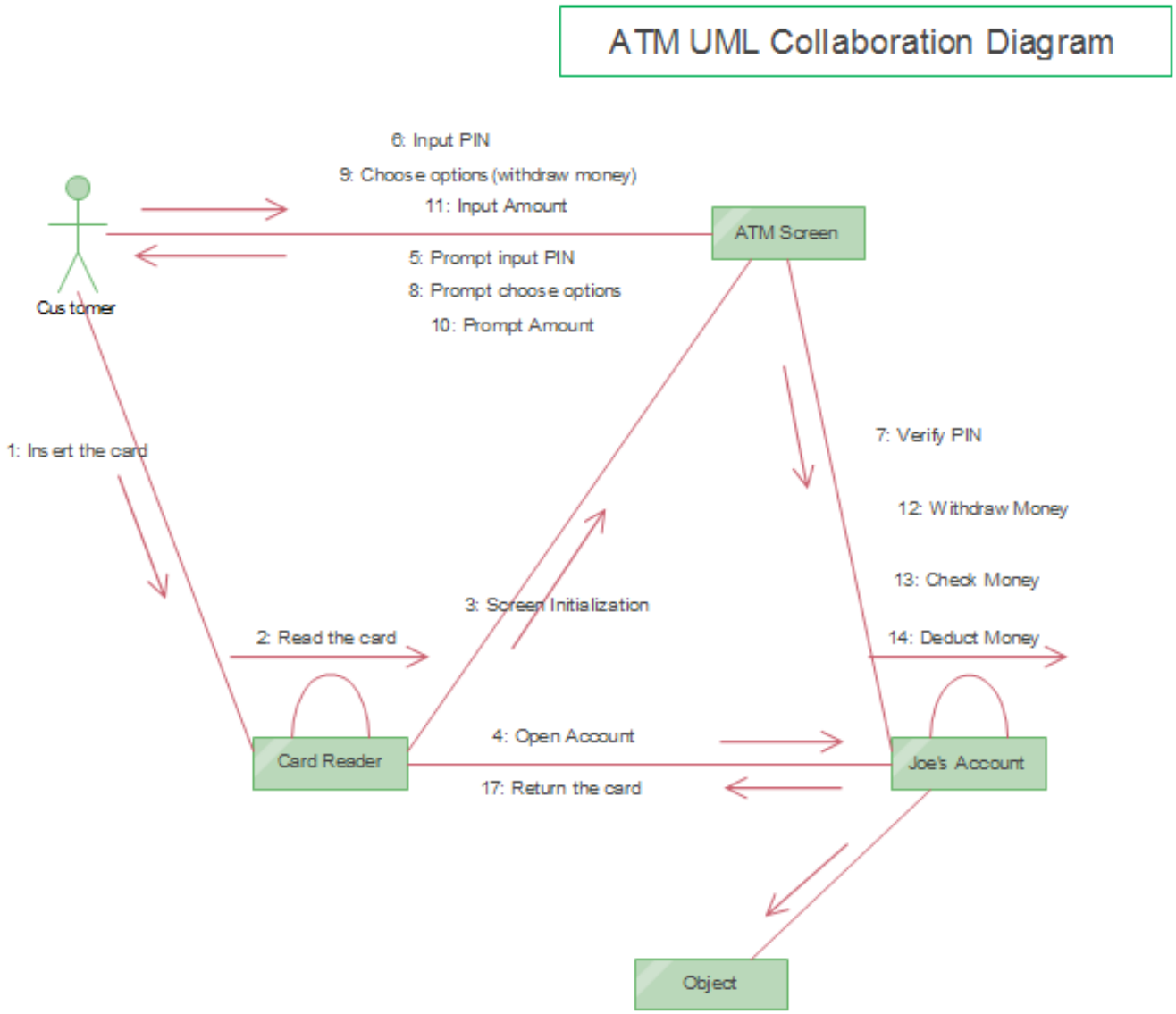
Graphical representation:

Example:



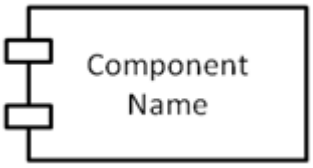
Collaboration: A collaboration is the collection of interactions among objects to achieve a goal. Graphically collaboration is represented as a dashed ellipse. A Collaboration can be a collection of classes or other elements.

Example:

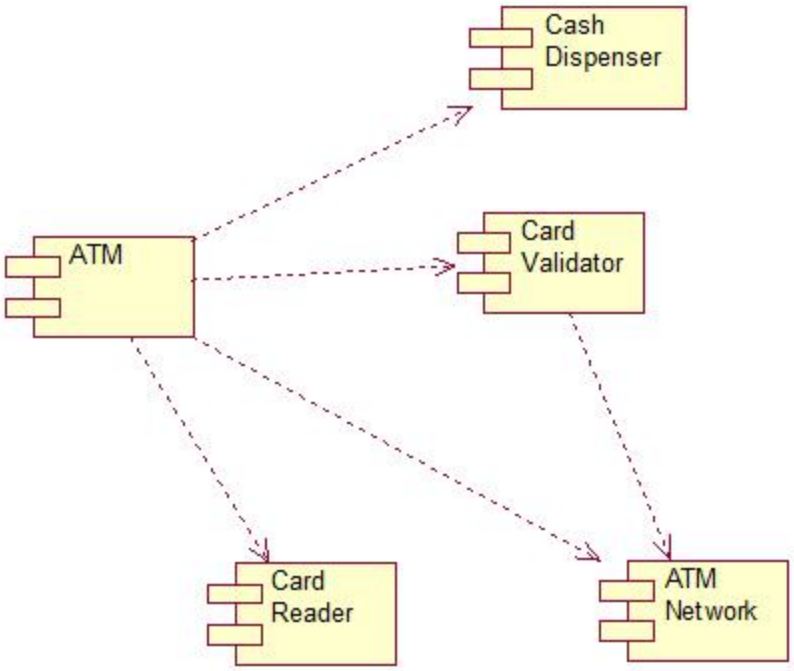


Component: A component is a physical and replaceable part of a system. Graphically component is represented as a tabbed rectangle. Examples of components are executable files, dll files, database tables, files and documents.

Graphical representation:

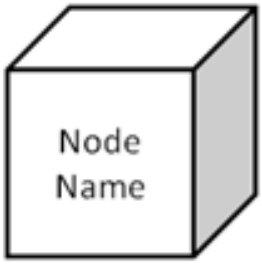


Example:

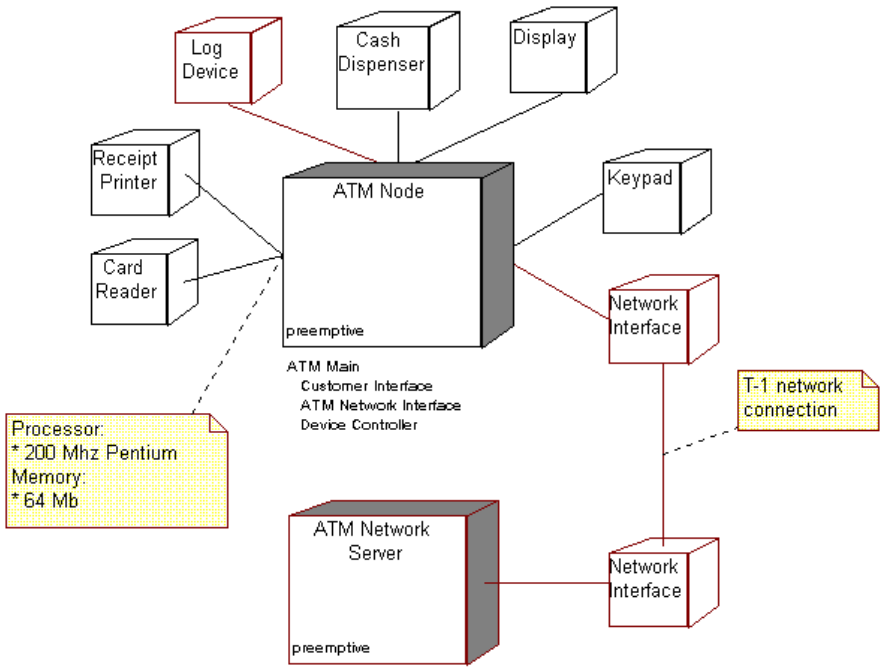


Node: A node is a physical element that exists at run time and represents a computational resource. Graphically node is represented as a cube. Examples of nodes are PCs, laptops, smart phones or any embedded system.

Graphical representation:



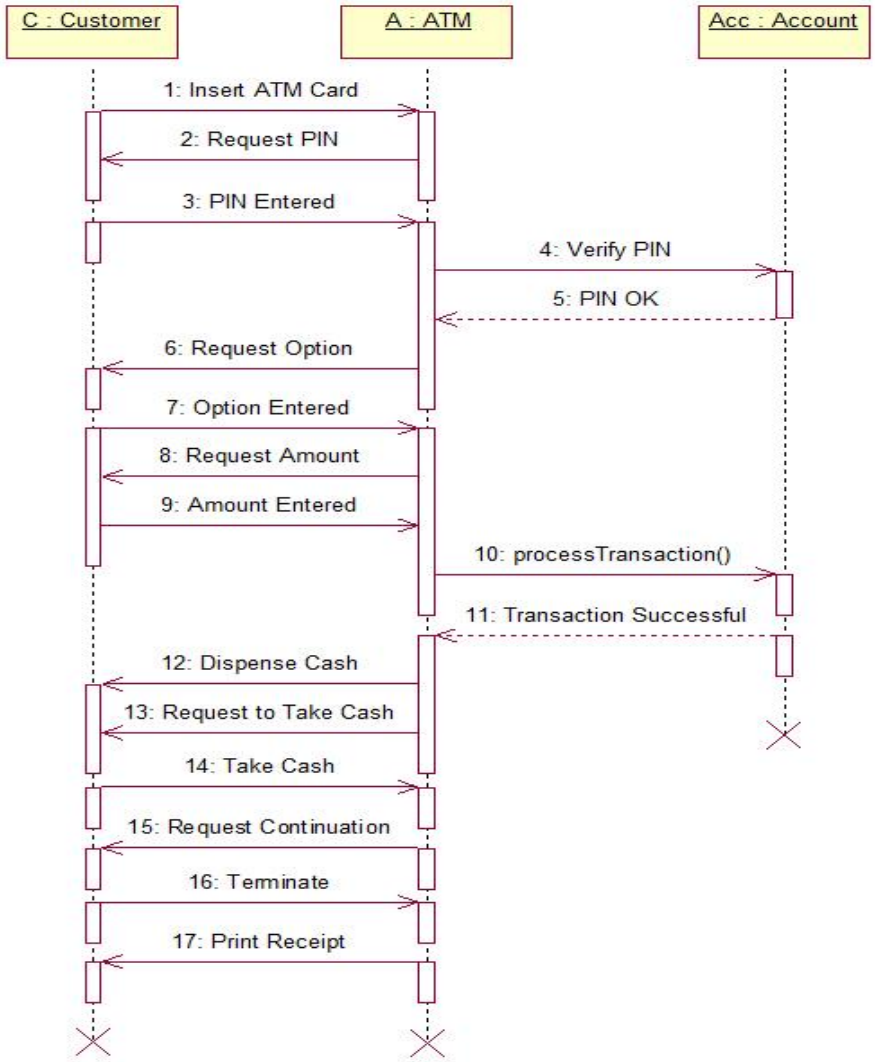
Example:



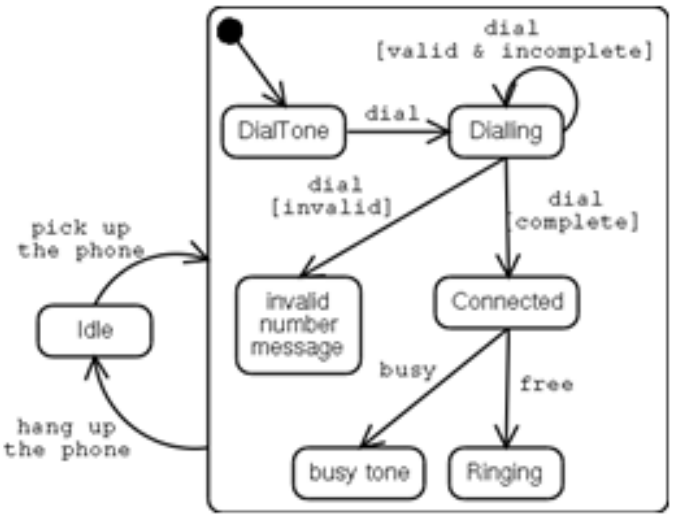
b.Behavioral Things

It represents the dynamic aspects of a software system. Behavior of a software system can be modeled as interactions or as a sequence of state changes.

Interaction or sequence: A behavior made up of a set of messages exchanged among a set of objects to perform a particular task. A message is represented as a solid arrow. Below is an example of interaction representing a phone conversation:


C : Customer
A : ATM
Acc : Account
1: Insert ATM Card
2: Request PIN
3: PIN Entered
4: Verify PIN
5: PIN OK
6: Request Option
7: Option Entered
8: Request Amount
9: Amount Entered
10: processTransaction()
11: Transaction Successful
12: Dispense Cash
13: Request to Take Cash
14: Take Cash
15: Request Continuation
16: Terminate
17: Print Receipt

State Machine: A behavior that specifies the sequences of states an object or interaction goes through during its' lifetime in response to events. A state is represented as a rectangle with rounded corners. Below is an example of state machine representing the states of a phone system:

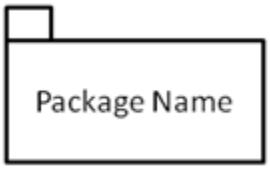


c.Grouping Things

Elements which are used for organizing related things and relationships in models.

Package: A general purpose mechanism for organizing elements into groups. Graphically package is represented as a tabbed folder. When the diagrams become large and cluttered, related are grouped into a package so that the diagram can become less complex and easy to understand.

Graphical representation:

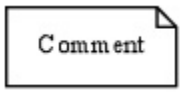


Example:

d. Annotational Things

Note: A symbol to display comments. Graphically note is represented as a rectangle with a dog ear at the top right corner.

Graphical representation:

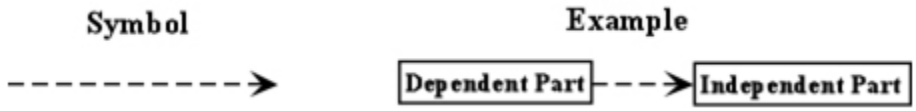


2. Relationships

The things in a diagram are connected through relationships. So, a relationship is a connection between two or more things.

Dependency: A semantic relationship, in which a change in one thing (the independent thing) may cause changes in the other thing (the dependent thing). This relationship is also known as “using” relationship. Graphically represented as dashed line with stick arrow head.

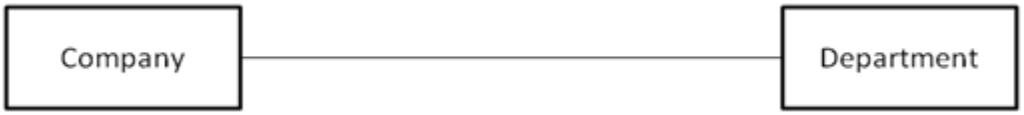
Graphical representation:



Example:

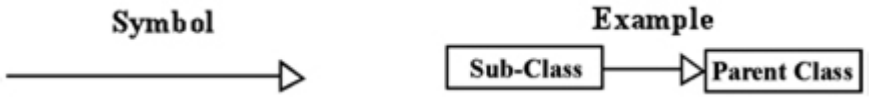
Association: A structural relationship describing connections between two or more things. Graphically represented as a solid line with optional stick arrow representing navigation.

Example:

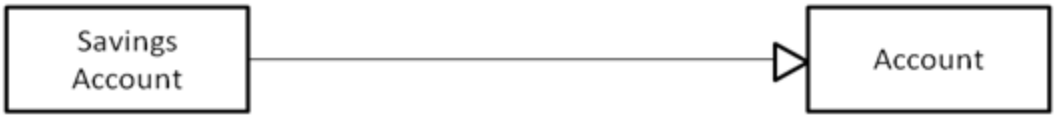


Generalization: Is a generalization-specialization relationship. Simply put this describes the relationship of a parent class (generalization) to its subclasses (specializations). Also known as “is-a” relationship.

Graphical representation:

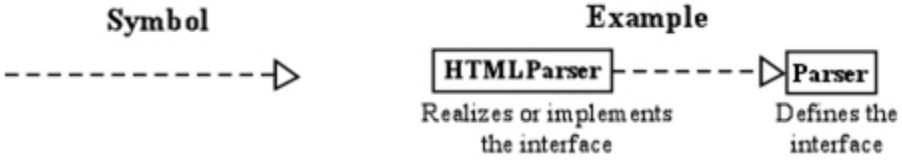


Example:



Realization: Defines a semantic relationship in which one class specifies something that another class will perform. Example: The relationship between an interface and the class that realizes or executes that interface.

Graphical representation and Example:



3. Diagrams

A diagram is a collection of elements often represented as a graph consisting of vertices and edges joining these vertices. These vertices in UML are things and the edges are relationships. UML includes nine diagrams:

1) Class diagram 2) Object diagram 3) Use case diagram 4) Component diagram 5) Deployment diagram 6) Sequence diagram 7) Collaboration diagram 8) Statechart diagram and 9) Activity diagram.

II. Rules (conceptual model of UML)

The rules of UML specify how the UMLs building blocks come together to develop diagrams. The rules enable the users to create well-formed models. A well-formed model is self-consistent and also consistent with the other models.

UML has rules for:

Names – What elements can be called as things, relationships and diagrams

Scope – The context that gives a specific meaning to a name

Visibility – How these names are seen and can be used by the other names

Integrity – How things properly relate to one another

Execution – What it means to run or simulate a model

III. Common Mechanisms in UML (conceptual model of uml)

Why UML is easy to learn and use? It's because of the four common mechanisms that apply throughout the UML. They are:

Specifications

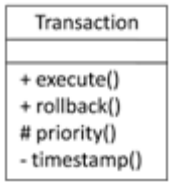
Adornments

Common divisions

Extensibility mechanisms

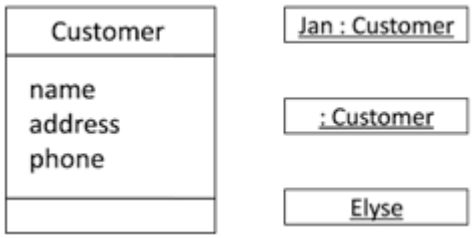
Specifications: Behind every graphical notation in UML there is a precise specification of the details that element represents. For example, a class icon is a rectangle and it specifies the name, attributes and operations of the class.

Adornments: The mechanism in UML which allows the users to specify extra information with the basic notation of an element is the adornments.



In the above example, the access specifiers: + (public), # (protected) and – (private) represent the visibility of the attributes which is extra information over the basic attribute representation.

Common Divisions: In UML there is clear division between semantically related elements like: separation between a class and an object and the separation between an interface and its implementation.



Extensibility Mechanisms

UMLs extensibility mechanisms allow the user to extend (new additions) the language in a controlled way. The extensibility mechanisms in UML are:

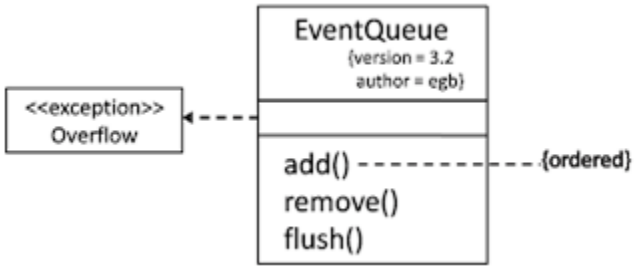
Stereotypes – Extends the vocabulary of UML. Allows users to declare new building blocks (icons) or extend the basic notations of the existing building blocks by stereotyping them using guillemets.

Tagged Values – Extends the properties of an UML building block. Allows us to specify extra information in the elements specification. Represented as text written inside braces and placed under the element name. The general syntax of a property is:

{ property name = value }

Constraints – Extends the semantics of a UMLs building block such as specifying new rules or modifying existing rules. Represented as text enclosed in braces and placed adjacent or beside the element name.

Example:



In the above example, we are specifying the exception “Overflow” using the class symbol and stereo typing it with “exception”. Also under the class name, “EventQueue” we are specifying additional properties like “version” and “author” using tagged values.

Finally, we are constraining the operation named “add” that before adding a new event to the EventQueue object, all the events must be “ordered” in some manner. This is specified using constraints in UML.

IV Software Testing Strategies:

- The process of investigating and checking a program to find whether there is an error or not, and does it fulfill the requirements or not, is called testing.
- The main objective of software testing is to design the tests in such a way that it systematically finds different types of errors without taking much time and effort so that less time is required for the development of the software.
- Testing is a set of activities that can be planned in advance and conducted systematically.
- Software Testing is a type of investigation to find out if there is any default or error present in the software so that the errors can be reduced or removed to increase the quality of the software and to check whether it fulfills the specifies requirements or not.
- **Software Testing is important,** because if there are any bugs or errors in the software, it can be identified early and can be solved before delivery of the software product.
- Properly tested software product ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction.

A strategic approach to software testing:

- A strategy (plan of action) for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then accepted, and how much effort, time, and resources will be required.
- Any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

Characteristics of good testing (or) The characteristic that process the testing templates:

- The developer should conduct the successful technical reviews to perform the testing successful.
- Testing starts with the component level and work from outside toward the integration of the whole computer based system.
- Different testing techniques are suitable at different point in time.
- Testing is organized by the developer of the software and by an independent test group.
- Debugging and testing are different activities, but debugging should be accommodated in any strategy of testing.
- The overall strategy for testing software includes:

Verification and Validation:

- Verification refers to the set of tasks that ensure (guarantee) that software correctly implements a specific function.
- Validation refers to a different set of tasks that ensure (guarantee) that the software that has been built is traceable to customer requirements.

According to Boehm scientist:

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

Difference between Verification and Validation

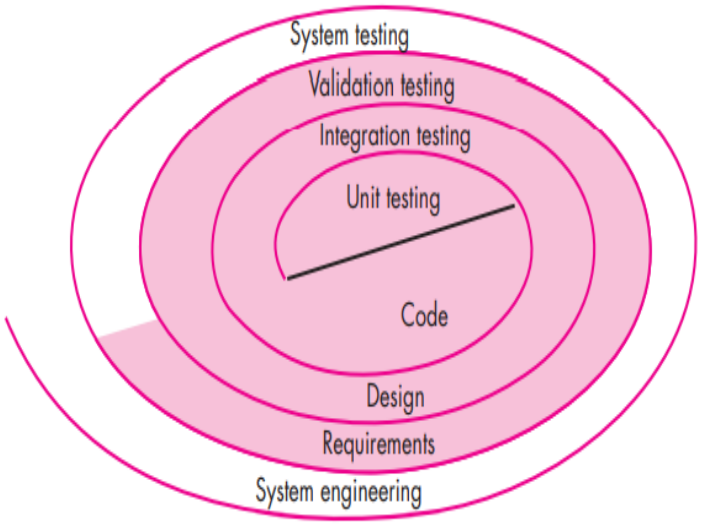
Verification	Validation
Verification is the process to find whether the software meets the specified requirements for particular phase.	The validation process is checked whether the software meets requirements and expectation of the customer.
It estimates an intermediate product.	It estimates the final product.
The objectives of verification are to check whether software is constructed according to requirement and design specification.	The objectives of the validation are to check whether the specifications are correct and satisfy the business need.
It describes whether the outputs are as per the inputs or not.	It explains whether they are accepted by the user or not.
Verification is done before the validation.	It is done after the verification.

Plans, requirement, specification, code are evaluated during the verifications.	Actual product or software is tested under validation.
It manually checks the files and document.	It is computer software or developed program based checking of files and document.

Software Testing Strategy:

A strategy of software testing is shown in the context of spiral.

Following figure shows the testing strategy:



Test strategies for conventional software:

Following are the four strategies for conventional software:

1) Unit testing
2) Integration testing
3) Validation testing
4) System testing

1) Unit testing

- Unit testing starts at the centre and each unit is implemented in source code.
- Unit testing focus on the smallest unit of software design, i.e module or software component.
- Test strategy conducted on each module interface to access the flow of input and output.
- The local data structure is accessible to verify integrity during execution.
- Boundary conditions are tested.
- In which all error handling paths are tested.
- An Independent path is tested.

Following figure shows the unit testing:

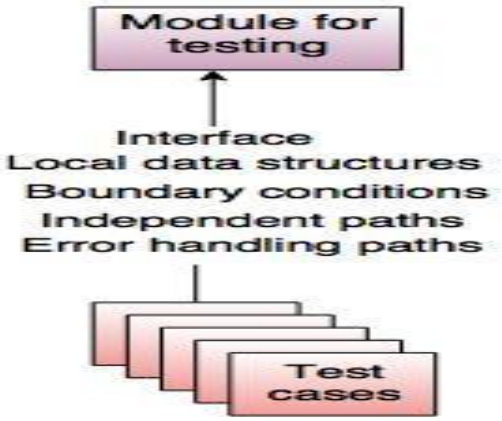


Fig. - Unit test

The unit test environment is as shown in following figure:

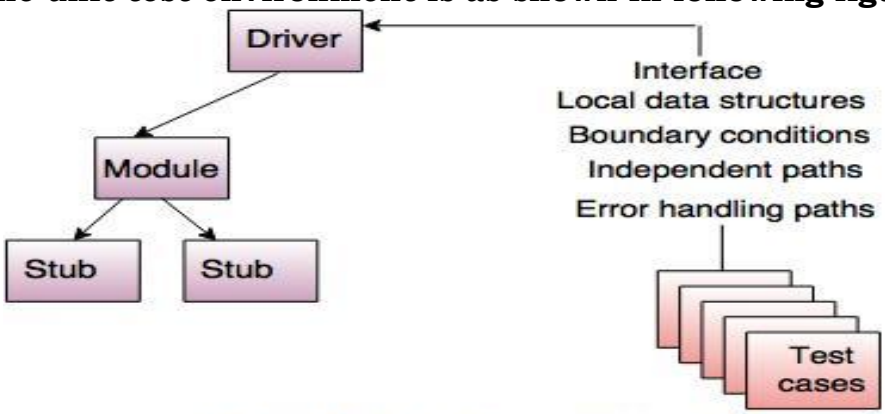


Fig. - Unit test environment

Difference between stub and driver

Stub	Driver
Stub is considered as subprogram.	It is a simple main program.
Stub does not accept test case data.	Driver accepts test case data.
It replaces the modules of the program into subprograms and is tested by the next driver.	Pass the data to the tested components and print the returned result.

2) Integration testing (sometimes called sandwich testing)

- An integration testing focuses on the construction and design of the software.
- Integration testing is used for the construction of software architecture.
- Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.
- As integration testing is conducted, the tester should identify critical modules.

There are two approaches of incremental testing are:

i) Non incremental integration testing

ii) Incremental integration testing

i) Non incremental integration testing

- Combines all the components in advanced.

ii) Incremental integration testing

- The programs are built and tested in small increments.
- The errors are easier to correct and isolate.
- Interfaces are fully tested and applied for a systematic test approach to it.

Following are the incremental integration strategies:

a. Top-down integration

b. Bottom-up integration

a. Top-down integration

- It is an incremental approach for building the software architecture.
- It starts with the main control module or program.
- Modules are merged by moving downward through the control hierarchy.

Following figure shows the top down integration.

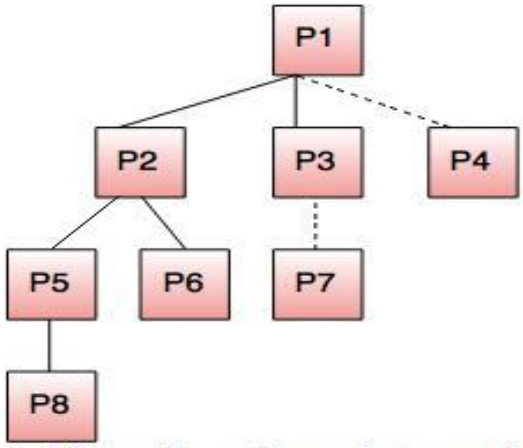


Fig. - Top-down integration

b. Bottom-up integration

In bottom up integration testing the components are combined from the lowest level in the program structure.

The bottom-up integration is implemented in following steps:

- The low level components are merged into clusters which perform a specific software sub function.
- A control program for testing (driver) coordinate test case input and output.
- After these steps are tested in cluster.
- The driver is removed and clusters are merged by moving upward on the program structure.

Following figure shows the bottom up integration:

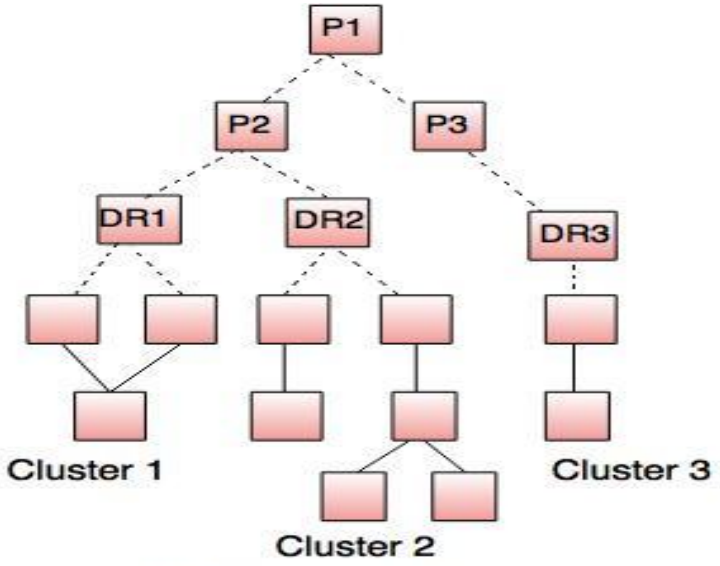


Fig. - Bottom-up integration

c) Regression testing

- Regression testing is used to check for defects propagated (circulated) to other modules by changes made to existing program. Thus regression testing is used to reduce the side effects of the changes.
- In regression testing the software architecture changes every time when a new module is added as part of integration testing.
- Each time a new module is added as part of integration testing, the software changes.
- New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly.
- In the context of an integration test strategy, regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- When any modification or changes are done to the application or even when any small change is done to the code then it can bring unexpected issues. Along with the new changes it becomes very important to test whether the existing functionality is intact or not.
- This can be achieved by doing the **regression testing.**
- The purpose of the regression testing is to find the bugs which may get introduced accidentally because of the new changes or modification.
- During confirmation testing the defect got fixed and that part of the application started working as intended. But there might be a possibility that the fix may have introduced or uncovered a different defect elsewhere in the software.

- The way to detect these **unexpected side-effects** of fixes is to do regression testing.
- This also ensures that the bugs found earlier are NOT creatable.
- Usually the regression testing is done by automation tools because in order to fix the defect the same test is carried out again and again and it will be very tedious and time consuming to do it manually.
- During regression testing the test cases are prioritized depending upon the changes done to the feature or module in the application. The feature or module where the changes or modification is done that entire feature is taken into priority for testing.
- This testing becomes very important when there are continuous modifications or enhancements done in the application or product.
- These changes or enhancements should NOT introduce new issues in the existing tested code.
- This helps in maintaining the quality of the product along with the new changes in the application.

Example:

Let's assume that there is an application which maintains the details of all the students in school. This application has four buttons Add, Save, Delete and Refresh. All the buttons functionalities are working as expected. Recently a new button Update is added in the application. This Update button functionality is tested and confirmed that it's working as expected. But at the same time it becomes very important to know that the introduction of this new button should not impact the other existing buttons functionality. Along with the Update button all the other buttons functionality are tested in order to find any new issues in the existing code. This process is known as regression testing.

When to use Regression testing it:

1. Any new feature is added
2. Any enhancement is done
3. Any bug is fixed
4. Any performance related issue is fixed

Advantages of Regression testing:

- It helps us to make sure that any changes like bug fixes or any enhancements to the module or application have not impacted the existing tested code.
- It ensures that the bugs found earlier are NOT creatable.
- Regression testing can be done by using the automation tools
- It helps in improving the quality of the product.

Disadvantages of Regression testing:

- If regression testing is done without using automated tools then it can be very tedious and time consuming because here we execute the same set of test cases again and again.
- Regression test is required even when a very small change is done in the code because this small modification can bring unexpected issues in the existing functionality.

d) Smoke testing

- The smoke testing is a kind of integration testing technique used for time critical projects wherein the project needs to be assessed (judged) on frequent basis.
- The developed software components are translated into code and merge to complete the product.
- Software components already translated into code are integrated into a "build". The "build" can be data files, libraries, reusable modules, or program components.
- A series of tests are designed to expose errors from build so that the "build" performs its functioning correctly.
- The "build" is integrated with the other builds and the entire product is some tested daily.
- Smoke testing provides a number of benefits when it is applied on complex, time critical software projects.
- Integration risk is minimized.
- The quality of the end product is improved.
- Error diagnosis and correction are simplified.
- Progress is easier to assess.

Difference between Regression and smoke testing

Regression testing	Smoke testing
Regression testing is used to check defects generated to other modules by making the changes in existing programs.	At the time of developing a software product smoke testing is used.
In regression tested components are tested again to verify the errors.	It permits the software development team to test projects on a regular basis.
Regression testing needs extra manpower because the cost of the project increases.	Smoke testing does not need an extra manpower because it does not affect the cost of project.
Testers conduct the regression testing.	Developer conducts smoke testing just before releasing the product.

3).Validation testing:

It checks all the requirements like functional, behavioral and performance requirement are validate against the construction software.
In validation testing the main focus is to identify errors in

- ✓ System i/o
- ✓ System functions and information data
- ✓ System interfaces with external parts
- ✓ User interfaces
- ✓ System behavior and performance

Acceptance Testing:

The Acceptance Testing is a kind of testing conducted to ensure that the s/w works correctly in the user work environment. It can be conducted over a period of weeks or months.

Types of Acceptance testing are:

Alpha and Beta testing:

Alpha testing	Beta testing
Alpha testing is executed at developers end by the customer. Tested or performed at developer's site	Beta testing is executed at end-user sites in the absence of a developer. Tested or performed at customer's site
It handles the software project and applications.	It usually handles software product.
It is not open to market and the public.	It is always open to the market and the public.
Alpha testing does not have any different name.	Beta testing is also known as the field testing.
Alpha testing is not able to test the errors because the developer does not known the type of user.	In beta testing, the developer corrects the errors as users report the problems.
In alpha testing, developer modifies the codes before release the software without user feedback.	In beta testing, developer modifies the code after getting the feedback from user.

4) System testing

- System testing is known as the testing behavior of the system or software, according to the software requirement specification.
- It is a series of various tests.
- It allows testing, verifying and validating the business requirement and application architecture.

- The primary motive of the tests is entirely to test the computer-based system.

Following are the system tests for software-based system

1. Recovery testing

- It is intended to check the system's ability to recover from failures.
- To check the recovery of the software, force the software to fail in various ways.
- Reinitialization, check pointing mechanism, data recovery and restart are evaluated correctness.

2. Security testing

- It verifies that system protection mechanism prevent improper access or data alteration.
- It also verifies that protection mechanisms build into the system prevent intrusion such as unauthorized internal or external access or determined the damage.
- It checks the system protection mechanism and secure improper access.

3. Stress testing

- Determines breakpoint of a system to establish maximum service level.
- System executes in a way which demands resources in abnormal quantity, frequency or volume.
- A variation of stress testing is known as sensitivity testing.

4. Performance testing

- Performance testing is designed to test run-time performance of the system in the context of an integrated system.
- It always combines with the stress testing and needs both hardware and software requirements.

5. Deployment testing

- It is also known as configuration testing.
- The software works in each environment in which it is to be operated.

Types of Unit testing:

A).White Box Testing:

- White-box testing is the detailed investigation of internal logic and structure of the code.
- White-box testing, sometimes called glass-box testing or open-box testing.
- It is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.
- The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.
- Using White-box testing methods, you 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.

Advantages	Disadvantages
As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively	Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased.
It helps in optimizing the code.	Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested.
Extra lines of code can be removed which can bring in hidden defects.	It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.
Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.	

1.Basis Path Testing:

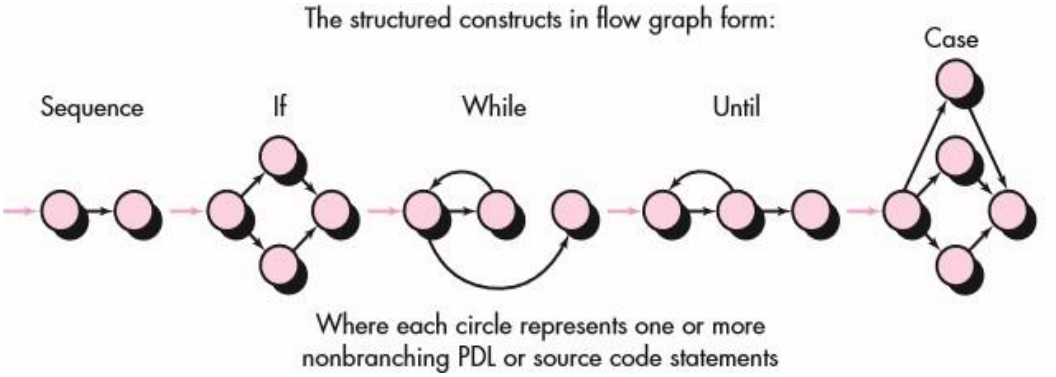
- Basis path testing is a white-box testing technique.
- The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

I. Flow Graph Notation
II. Independent Program Paths
III. Deriving Test Cases
IV. Graph Matrices

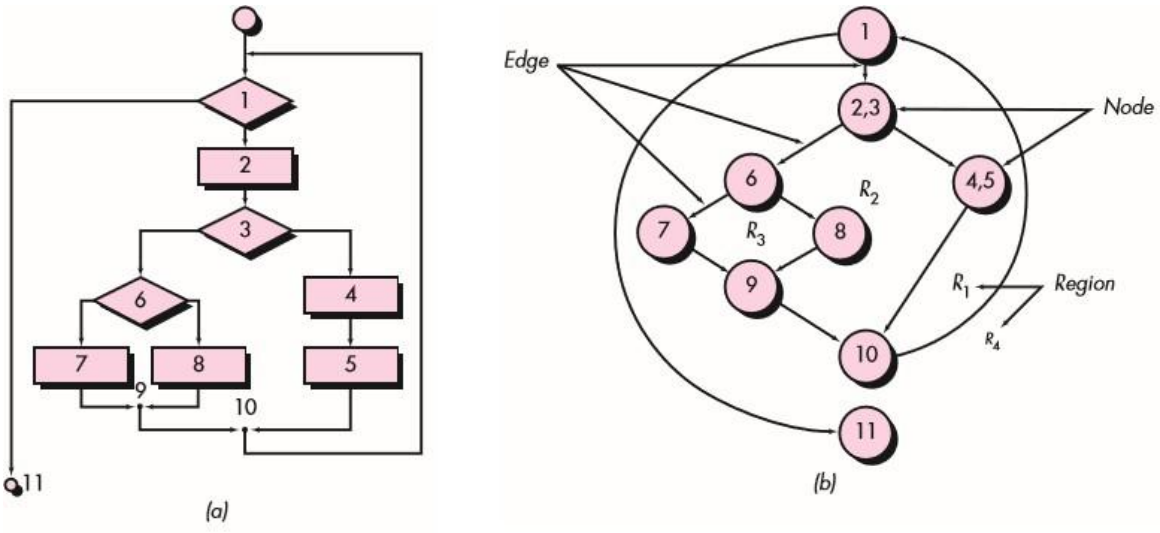
I.Flow Graph Notation:

- A simple notation for the representation of control flow, called a flow graph (or program graph).
- The flow graph depicts logical control flow using the notation in the following figure.
- Arrows called edges represent flow of control
- Circles called nodes represent one or more actions.
- Areas bounded by edges and nodes called regions.
- A predicate node is a node containing a condition.
- Any procedural design can be translated into a flow graph.
- Note that compound Boolean expressions at tests generate at least two

predicate node and additional arcs.



- To illustrate the use of a flow graph, consider the procedural design representation in Figure.
- Here, Figure (a) flow chart is used to depict program control structure.
- Figure(b) maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).
- Figure(b), each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called regions. When counting regions.



II.Independent Program Paths:

- 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 software metric that provides a quantitative measure of the logical complexity of a program.
- When used in the context of the basis path testing method, the value computed for Cyclomatic complexity defines the number of independent paths in the basis set of a program and provides with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.
- Cyclomatic complexity has a foundation in graph theory and provides with extremely useful software metric.
- Complexity is computed in one of three ways:

1. The number of regions of the flow graph is called 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 and N is the number of flow graph nodes.
3. Cyclomatic complexity V(G) for a flow graph G is also defined as

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

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

III.Deriving Test Cases:

The following steps can be applied to derive the basis set:

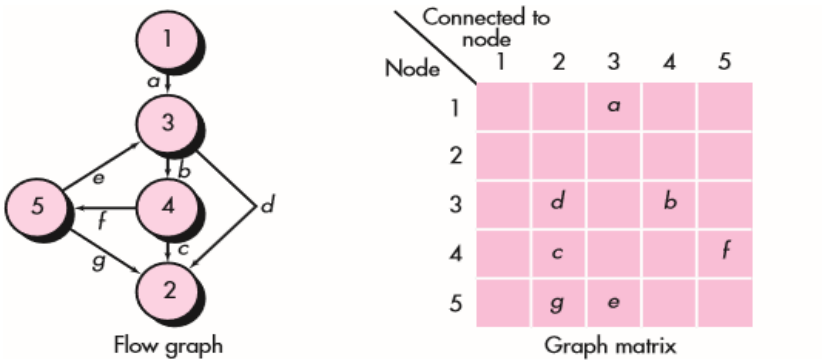
1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the Cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

IV.Graph Matrices:

- A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing.
- A graph matrix is a square matrix whose size (i.e., number of rows and columns) 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.
- A simple example of a flow graph and its corresponding graph matrix is shown in Figure.
- Referring to the figure, each node on the flow graph is identified by

numbers, while each edge is identified by letters.

- A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b.
- The graph matrix is nothing more than a tabular representation of a flow graph.
- By adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.
- The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).



2. Control Structure Testing:

- Although basis path testing is simple and highly effective, it is not sufficient in itself.
- Other variations on control structure testing necessary. These broaden testing coverage and improve the quality of white-box testing.

I. Condition testing:

- Condition testing is a test-case design method that exercises the logical conditions contained in a program module.
- A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator.
- A relational expression takes the form
  E1<relational-operator> E2
- Where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following:
- A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses.
- The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

II. Data Flow Testing

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

- To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables.
- For a statement with S as its statement number,
- DEF(S)= {X | statement S contains a definition of X}
- USE(S) = {X | statement S contains a use of X}
- If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be live at statement S‘ if there exists a path from statement S to statement S‘ that contains no other definition of X.

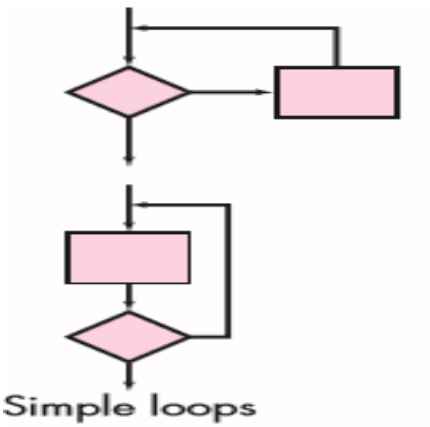
III.Loop Testing

- Loops are the foundation for the vast majority of all algorithms implemented in software.
- Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs.
- Four different classes of loops can be defined:

a.Simple loops:

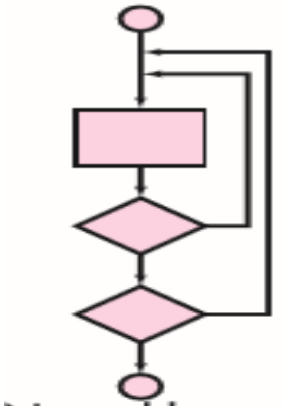
The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where m < n.
5. n - 1, n, n + 1 passes through the loop.



b.Nested loops:

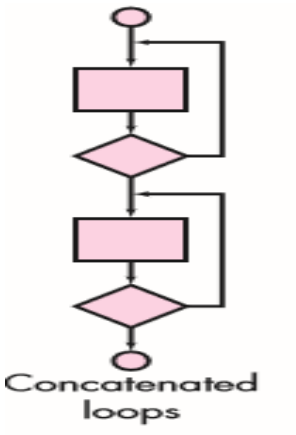
If we were to extend the test approach for simple loops to nested loops, the numberof possible tests would grow geometrically as the level of nesting increases.



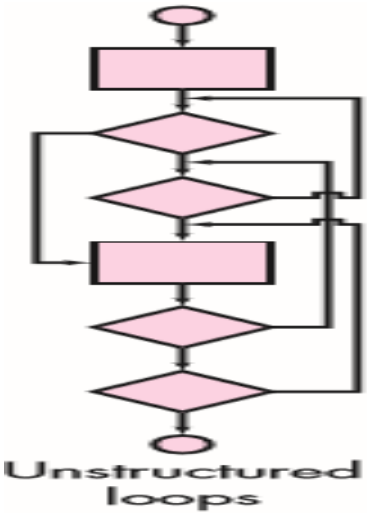
1. Beizer suggests an approach that will help to reduce the number of tests:
2. Start at the innermost loop. Set all other loops to minimum values.
3. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
4. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimumvalues and other nested loops to -typical|| values.
5. Continue until all loops have been tested.

c.Concatenated loops:

In the concatenated loops, if two loops are independent of each other thenthey are tested using simple loops or else test them as nested loops. However if the loop counterfor one loop is used as the initial value for the others, then it will not be considered as an independent loops.



d.Unstructured loops: Whenever possible, this class of loops should be redesigned to reflect theuse of the structured programming constructs.



B).Black Box Testing:

- Black-box testing, also called behavioral testing.
- It focuses on the functional requirements of the software.
- A Black-box testing technique enables to derive sets of input conditions that will fullyexercise all functional requirements for a program.
- Black-box testing attempts to find errors in the following categories:
  - ✓ incorrect or missing functions
  - ✓ interface errors
  - ✓ errors in data structures or external database access
  - ✓ behavior or performance errors
  - ✓ initialization and termination errors.

Black – Box Testing Techniques are:

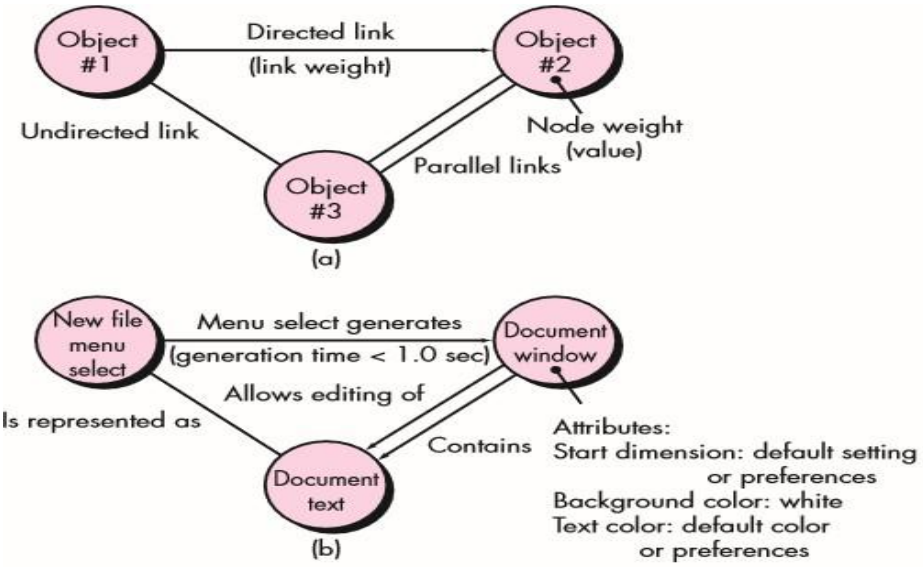
1. **Graph-Based Testing Methods**
2. **Equivalence Partitioning**
3. **Boundary Value Analysis**
4. **Orthogonal Array Testing**

1. Graph-Based Testing Methods:

- The first step in black-box testing is to understand the objects5 that are modeled in software and the relationships that connect these objects.
- Next step is to define a series of tests that verify -all objects have the expected relationship to one another.
- To accomplish these steps, create a **graph**—a collection of nodes that represent objects, **links** that represent the relationships

between objects, **node weights** that describe the properties of a node (e.g., a specific data value or state behavior), and **link weights** that describe some characteristic of a link.

- The symbolic representation of a graph is shown in below Figure.
- **Nodes** are represented as circles connected by links that take a number of different forms.
- **A directed link** (represented by an arrow) indicates that a relationship moves in only one direction.
- **A bidirectional link**, also called a symmetric link, implies that the relationship applies in both directions.
- **Parallel links** are used when a number of different relationships are established between graph nodes.



2. Equivalence Partitioning:

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.
- Equivalence classes may be defined according to the following guidelines:
  - If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
  - If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
  - If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.

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

3. Boundary Value Analysis:

- A greater number of errors occur at the boundaries of the input domain rather than in the -center of input domain.
- For this reason that boundary value analysis (BVA) has been developed as a testing technique
- Boundary value analysis leads to a selection of test cases that exercise bounding values.
- BVA leads to the selection of test cases at the -edges‖ of the class. Rather than focusing solely on input conditions.
- BVA derives test cases from the output domain also.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary. Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

4. Orthogonal Array Testing:

- Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- The orthogonal array testing method is particularly useful in finding **region faults—**an error category associated with faulty logic within a software component.
- **For example**, when a train ticket has to be verified, the factors such

as - the number of passengers, ticket number, seat numbers and the train numbers has to be tested, which becomes difficult when a tester verifies input one by one. Hence, it will be more efficient when he combines more inputs together and does testing. Here, use the Orthogonal Array testing method.

- When orthogonal array testing occurs, an **L9 orthogonal array** of test cases is created.
- The L9 orthogonal array has a -balancing property.
- That is, test cases (represented by dark dots in the figure) are -dispersed uniformly throughout the test domain,‖ as illustrated in the right-hand cube in Figure.
- To illustrate the use of the L9 orthogonal array, consider the send

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

function for a fax application.

- Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values:
- P1 = 1, send it now : P1 = 2, send it one hour later : P1 = 3, send it after midnight
- P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions.
- If a -one input item at a time testing strategy were chosen, the following sequence of tests (P1,P2,P3,P4) would be specified: (1,1,1,1),(2,1,1,1),(3,1,1,1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3).
- The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure.

Difference between white and black box testing:

White-Box Testing	Black-box Testing
White-box testing known as glass-box or clear box testing.	Black-box testing also called as behavioral testing.
White box testing requires the tester to know and understand how the software works, they are able to "see inside" the program.	**Black box testing** requires the tester to understand what the program is supposed to do, but not how it works, they are unable to "see inside" the program.
It starts early in the testing process.	It is applied in the final stages of testing.
In white box testing the procedural details, all the logical paths all the internal data structures are closely examined.	Black box testing examines fundamental aspect of the system. A black box refers to a system whose behavior has to be observed entirely by inputs and outputs.
In this testing knowledge of implementation is needed.	In this testing knowledge of implementation is not needed.
White box testing is mainly done by the developer.	This testing is done by the testers.
In this testing, the tester must be technically sound.	In black box testing, testers may or may not be technically sound.
Various white box testing methods are: Basic Path Testing and Control Structure Testing.	Various black box testing are: Graph-Based testing method, Equivalence partitioning, Boundary Value Analysis, Orthogonal Array Testing.
This type of testing is suitable for small projects.	This type of testing is suitable for large projects.
White box testing lead to test the program thoroughly	During black box testing the program cannot be tested 100 percent

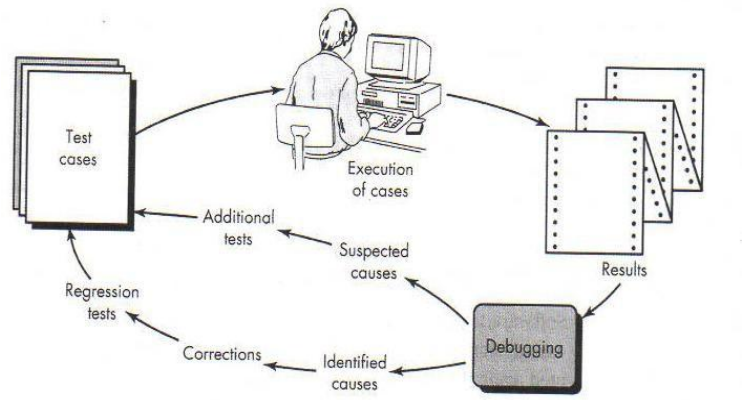
Art of Debugging:

Debugging process

- Debugging process is not a testing process, but it is the result of testing.
- This process starts with the test cases.
- The debugging process gives two results, i.e., the cause is found and corrected second is the cause is not found.
- Debugging identifies the correct cause of error.
- Debugging process beings with the execution of a test case.

Debugging Strategies:

- Objective of debugging is to find and correct the cause of a software error or defect.
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies



Debugging Strategies:

Following are the debugging strategies:

1. Brute force

- A brute force approach is **an approach that finds all the possible solutions to find a satisfactory solution to a given problem**. The brute force algorithm tries out all the possibilities till a satisfactory solution is not found.
- Most common and least efficient method for isolating the cause of an s/w error. This is applied when all else fails.
- In this method, run-time traces are invoked and program is loaded with output statements.
- It tries to find the cause from the lot of information leads to waste of time and effort.
- This is the least efficient method of debugging. In this method "let computer find the error" approach is used.

2. Backtracking

- Backtracking is **a technique based on algorithm to solve problem**. It uses recursive calling to find the solution by building a solution step by step increasing values with time.
- It removes the solution that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.
- It is a common debugging approach, Useful for small programs.
- **Backtracking** is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any

point in time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

- The source code is traced manually till the cause is found.

3. Cause elimination

- Cause elimination establishes the concept of binary partitioning to reduce the number of locations where errors can exist.
- Thus testing is an essential activity carried out during software development process for improving quality of the product.
- It is a common debugging approach, Useful for small programs.

4.Automated Debugging:

- This supplements the above approaches with debugging tools that provide semi-automated support like debugging compilers, dynamic debugging aids, test case generators, mapping tools etc.

Software Quality:

- Software Quality conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
- There are three main reasons for why software quality gets failed.
  1. Software requirements must be well understood before the software development process begins.
  2. If the software confirms the explicit (clear) requirements but not satisfying the implicit requirements then surely quality of software being developed is poor.
  3. The set of development criteria has to be decided in order to specify the standards of the product. If such a criteria is not been fixed then definitely the software product will lack the quality.

Factors that affect software quality can be categorized in two broad groups:

a) Factors that can be directly measured (e.g. defects uncovered during testing)

b) Factors that can be measured only indirectly (e.g. usability or maintainability)

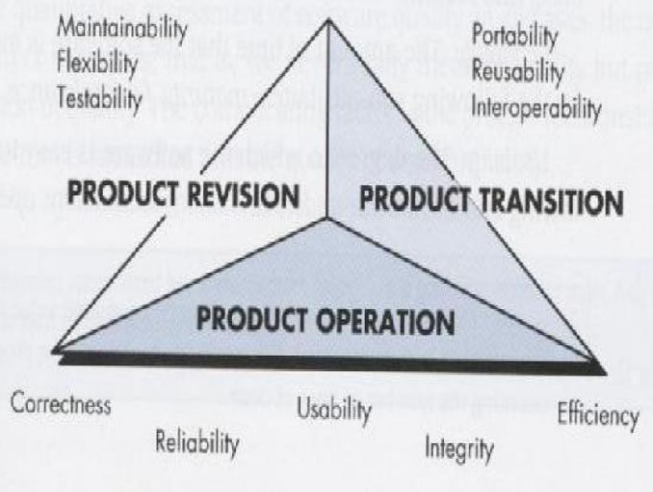
- McCall, Richards, and Walters propose a useful categorization of factors that affect software quality.

McCall's quality factors:

1. Product operation
   - Correctness
   - Reliability
   - Efficiency
   - Integrity
   - Usability

2. Product Revision
   - Maintainability
   - Flexibility
   - Testability

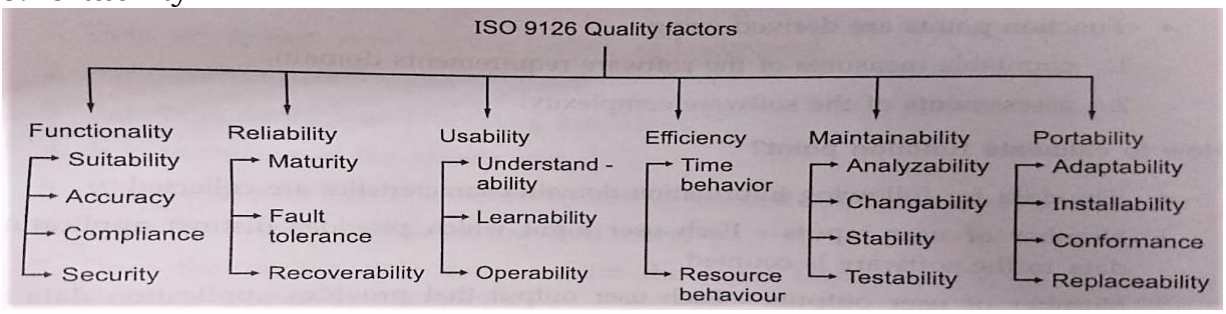
3. Product Transition
   - Portability
   - Reusability
   - Interoperability



1. ***Correctness***. The ability to fulfill the specification and customers requirements.
2. ***Reliability***. The extent to which a program can be expected to perform its intended function with required precision.
3. ***Efficiency***. The amount of computing resources and time required by a program to perform its function.
4. ***Integrity***. This is the controlling ability by which unauthorized access to the system can be prevented.
5. ***Usability***. The ability to prepare the valid input and interpret the correct output of a program.
6. ***Maintainability***. *The* effort required to locate and fix an error in a program.
7. ***Flexibility***. Effort required to modify an operational program.
8. ***Testability***. Effort required to test a program to ensure that it performs its intended function.
9. ***Portability***. Effort required to transfer the program from one hardware and/or software system environment to another.
10. ***Reusability***. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.
11. ***Interoperability***. The ability of the system to work with other system.

ISO 9126 Quality Factors

1.Functionality
2.Reliability
3.Usability
4.Efficiency
5.Maintainability
6.Portability



FURPS: functionality, usability, reliability, performance, and supportability. The FURPS quality factors draw liberally from earlier work, defining the following attributes for each of the five major factors:

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ***Usability*** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ***Performance*** is measured by processing speed, response time, resource consumption, throughput, and efficiency.
- ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

Types of Software Metrics:

Software testing metrics are divided into three categories:

1. **Product Metrics:** A product's size, design, performance, quality, and complexity are defined by product metrics. Developers can improve the quality of their software development by utilizing these features.
2. **Process Metrics:** A project's characteristics and execution are defined by process metrics. These features are critical to the SDLC process's improvement and maintenance (Software Development Life Cycle).
3. **Project Metrics:** Project Metrics are used to assess a project's overall quality. It is used to estimate a project's resources and deliverables, as well as to determine costs, productivity, and flaws.

Product Metrics:
1).Metrics for the Analysis model:

- It is useful in estimating the size and quality of the project.
- In order to determine the metrics in analysis model "size" of the software is used as a measure.

a).Function point (FP) model metric:

- It is based on functionality of the delivered application.
- It is proposed by Albrecht in 1979 for IBM
- Function points (FP) are derived using countable measures of the software requirements domain and assessments of the software complexity.
- It measures the functionality of the delivered application.

FP computed from the following parameters:

• Number of user inputs –Each user input which provides distinct application data to the s/w is counted.

• Number of user outputs –Each user output that provides application data to the user is counted. Ex: screens, reports and Error messages.

• Number of user inquiries –An on-line input that results in the generation of some immediate software response in the form of an o/p.

• Number of files –Each logical master file, i.e., a logical grouping of data that may be part of a database or a separate file.

• Number of external interfaces- All machine-readable interfaces that are used to transmit information to another system is counted.
• The organization need to develop criteria which determine whether a particular entry is simple, average or complex.

Types of FP Attributes

Measurements Parameters	Examples
1.Number of External Inputs(EI)	Input screen and tables
2. Number of External Output (EO)	Output screens and reports
3. Number of external inquiries (EQ)	Prompts and interrupts.
4. Number of internal files (ILF)	Databases and directories
5. Number of external interfaces (EIF)	Shared databases and functions.

All these parameters are then individually assessed for complexity.

Measurement parameter	Count		Weighting Factor: Simple	Average	Complex		
Number of user inputs	3	×	3	4	6	=	9
Number of user outputs	2	×	4	5	7	=	8
Number of user inquiries	2	×	3	4	6	=	6
Number of files	1	×	7	10	15	=	7
Number of external interfaces	4	×	5	7	10	=	20
Count total							50

FP = count total * (0.65 + (0.01 * sum (F_i)))

- ∑**(f_i)** is the sum of all 14 questionnaires(given below and each question value ranges from 0-5).

General System Characteristics (GSCs)	Degree of Influence (DI) 0 - 5
1. Data Communications	_____
2. Distributed Data Processing	_____
3. Performance	_____
4. Heavily Used Configuration	_____
5. Transaction Rate	_____
6. Online Data Entry	_____
7. End-User Efficiency	_____
8. Online Update	_____
9. Complex Processing	_____
10. Reusability	_____
11. Installation Ease	_____
12. Operational Ease	_____
13. Multiple Sites	_____
14. Facilitate Change	_____

- ∑**(f_i)** (degree of influence) ranges from 0 to 70(14*5, if max rating is 5 for each question), i.e., 0 <= ∑(f_i) <=70

Once the FP is calculated then we can compute various measures such as

1. Productivity=FP/Person-month
2. Quality=Number of faults/FP
3. Cost=$/FP
4. Documentation=pages of documentation/FP

Advantages:

- This method is independent of programming languages.
- It is based on the data which can be obtained in early stage of project.

Disadvantages:

- The FP has no significant meaning. It is just a numerical value.

Example: Compute the function point, productivity, documentation, cost per function for the following data:

1. Number of user inputs = 24
2. Number of user outputs = 46
3. Number of inquiries = 8
4. Number of files = 4
5. Number of external interfaces = 2
6. Effort = 36.9 p-m
7. Technical documents = 265 pages
8. User documents = 122 pages
9. Cost = $7744/ month

Various processing complexity factors are: 4, 1, 0, 3, 3, 5, 4, 4, 3, 3, 2, 2, 4, 5.

Solution:

Measurement Parameter	**Count**		**Weighing factor**
1. Number of external inputs (EI)	24	*	4 = 96
2. Number of external outputs (EO)	46	*	4 = 184
3. Number of external inquiries (EQ)	8	*	6 = 48
4. Number of internal files (ILF)	4	*	10 = 40
5. Number of external interfaces (EIF) Count-total →	2	*	5 = 10 378

So sum of all f_i (i ← 1 to 14) = 4 + 1 + 0 + 3 + 5 + 4 + 4 + 3 + 3 + 2 + 2 + 4 + 5 = 43

FP = Count-total * [0.65 + 0.01 *∑(f_i)]
= 378 * [0.65 + 0.01 * 43]
= 378 * [0.65 + 0.43]
= 378 * 1.08 = 408

$$\text{Productivity} = \frac{\text{FP}}{\text{Effort}} = \frac{408}{36.9} = 11.1$$

Total pages of documentation = technical document + user document
= 265 + 122 = 387pages

Documentation = Pages of documentation/FP
= 387/408 = 0.94

$$\text{Cost per function} = \frac{\text{cost}}{\text{productivity}} = \frac{7744}{11.1} = \$700$$

2). Metrics for Design Model:

This model is developed by considering three aspects such as

- Architectural design metrics
- Metrics for Object Oriented Design (MOOD)

a).Architectural design Metrics:

- While determining the architectural design, primarily the characteristics of program architecture are considered.
- It does not focus on inner working of the system.

i).Metrics by Card and Glass

- Two scientists Card and Glass have suggested three complexity measures as

Structural complexity:

- It depends upon the fan-out for modules.
- It can be defined as $S(k)=f^2_{out}(k)$
- Where f_{out} represents fan-out for module k [fan out means number of modules that are subordinating module k]

Data complexity:

- It is the complexity within the interface of internal module.
- For some module k it can be defined as

$$D(k)=tot\_var(k) /[fout(k)+1]$$

Where tot_var is total number of input and output variables going to and coming out of the module.

System complexity:

- System complexity is the combination of structural and data complexity.
- It can be denoted as $Sy(k) = S(k)+D(k)$

When structural, data and system complexity get increased the overall architectural complexity also gets increased.

ii).Metrics by Fenton:

- These are simple morphology metrics that are used to compare different architectures with the help of size, depth, width and edge to node ratio.
- The metrics can be given as below-

$$Size = n + e$$

Where n is the number of nodes and e is number of edges.

- Depth is the longest path from root to leaf node.
- Width is the maximum number of nodes at particular level
- Edge to node ration r= e/n

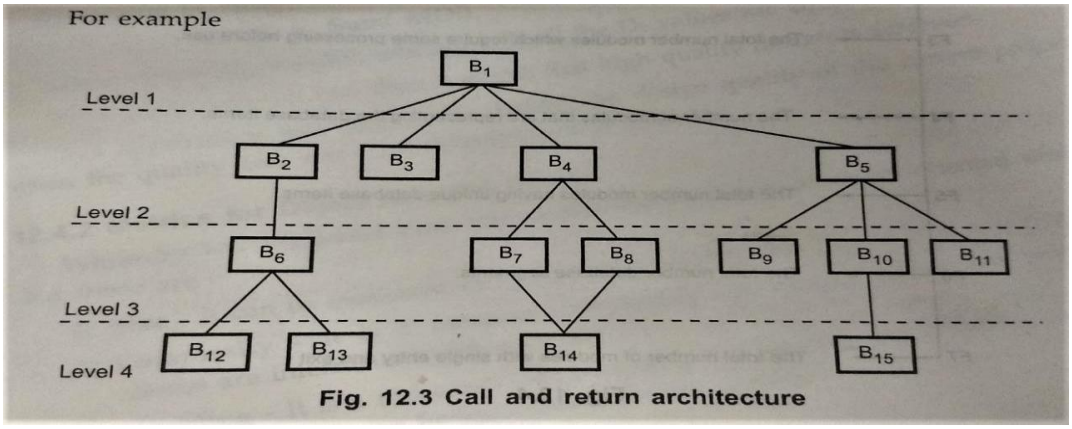


Fig. 12.3 Call and return architecture

Size=n+e=15+15 (here 15 nodes and 15 edges)
=30
Depth=longest path from the root
=3
Width =maximum number of nodes at particular level
=6 (at level 2 there are 6 nodes)
Similarly edge to node ration can be=e/n
=15/15 =1

b). Metrics for Object Oriented Design (MOOD):

i). Whitmire has suggested nine measurable characteristics of object oriented design and that are-

1) Size - It can be used to measure the size of the product.
2) Complexity -It is a measure representing the characteristics that how the classes are interrelated with each other.
3) Coupling - It is a measure stating the collaborations between classes or number of messages that can be passed between the objects.
   a.Population: means a number of classes and operations
   b.Volume: means total number of classes or operations that are collected dynamically
   c.Length: means total number of interconnected design elements
   d.Functionality: is a measure of output delivered to the customer

4) Completeness - It is the measure representing all the requirements of the design component
5) Cohesion - It is the degree by which the set of properties that are working together to solve particular property.
6) Sufficiency - It is the measure representing the necessary requirements of the design component
7) Primitiveness - The degree by which the operations are simple. In other words, the measure by which number of operations are independent upon other.

8) Similarity- The degrees to which two or more classes are similar with respect to their functionalities and behavior.
9) Volatility -Due to changes in requirements or some other reasons modifications in the design of application may occur. Volatility is a measure that represents the probability of changes that will occur.

These product metrics for object oriented design is applicable to design as well as analysis model.

ii)CK Metrics Suite:

- The metric for object oriented design is mainly based on the fundamental unit called 'classes'.
- Chidamber and Kemerer have proposed object oriented software metrics which are popularly known as CK metrics suite.
- These are six class based design metrics for 00 (object oriented) systems.
- Weighted Methods per class (WMC), Depth of Inheritance Tree (DIT), Number of children (NOC), coupling between object classes (CBO), Response for a class (RFC), Lack of cohesion in Methods (LSOM).

iii) MOOD (Metric for Object Oriented Design) Metrics Suite:

- MOOD metrics suite is proposed by Harrison Counsell and Nithi for object oriented design.
- It includes two metrics
  1) Method Inheritance Factor (MIF) and
  2) Coupling Factor (CF)

1).Method Inheritance Factor (MIF):

The MIF can be computed as

$$MIF = \Sigma M_i(C_i)) / \Sigma M_a(C_i))$$

- Where $M_a(C_i) = M_d(C_i) + M_i(C_i)$
- And i vary from 1 to n and n denotes the total number of classes in the architecture.
- C_i is a class within the architecture
- $M_a(C_i)$ is the total number of Methods in C_i
- $M_i(C_i)$ is number of Methods inherited in C_i
- $M_d(C_i)$ is number of Methods declared in class C_i
- MIF represents the impact of inheritance on Object Oriented system

2. Coupling Factor (CF):

- In software systems coupling represents the connection between elements of object oriented design.
- The coupling factor can be denoted as follows-

$$CF = \Sigma_i \Sigma_j IsClient\ (C_i, C_j)/(T^2c - T_c)$$

where,

- i and j varies from 1 to total number of classes in the architecture T_c
- IsClient is a Boolean function, it is = 1 if there exists a relationship between client and server classes
- IsClient = 0 if there is no relationship between client class and server classes.
- As CF increases the complexity of object oriented design gets increased.

iv).Lorenz and Kidd 00 Metrics:

Lorenz and Kidd have proposed the conceptual division of class based metric into four distinct categories

1) Class 2) Inheritance 3) Class internals 4) Externals

1. Class metrics- the total number of attributes and methods are counted for each class.
2. Inheritance metrics -deals with the number of operations that can be reused in the class hierarchy.
3. The class internals metric -deals with cohesion and code oriented issues.
4. The external metric -is based on number of couplings between the classes.

3).METRIC FOR SOURCE CODE:

- Halstead has proposed some software science metrics based on commonsense, information theory and psychology.
- It is used to measure the size of the program, after the code is generated or estimated once design is completed.

Let n1 = the number of distinct operators that appear in a program
n2 = the number of distinct operands that appear in a program

- Overall program length N can be computed as
  $N = n_1 \log_2 n_1 + n_2 \log_2 n_2$
- The program volume can be defined as $V = N \log_2(n_1 + n_2)$

Let N = Total count for all the operands in the program.

The program volume ration L can be defined as:
$L = 2/n_1 * n_2/N$

4).METRIC FOR TESTING:

- Test metrics help to determine what types of enhancements are required in order to create a defect-free, high-quality software product.
- Halstead's metrics for estimating the testing efforts are as given below.
- The Halstead Effort (e) can be defined as

$$e = V/PL$$

where

Program volume *V*, and program level PL and it can be computed as

$PL = 1/[(n1/2) \times (N / n_2)]$

n1 = the number of distinct operators that appear in a program

n2 = the number of distinct operands that appear in a program

N = the total number of operator occurrences.

The percentage of overall testing effort=testing effort of specific module/testing efforts of all the modules.

5).METRICS FOR SOFTWARE MAINTENANCE:

- Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is necessity for almost any kind of product.
- However, most products need maintenance due to the wear and tear caused by use.
- On the other hand, software products do not need maintenance on this count, but need maintenance to correct errors, enhance features, port to new platforms, etc.

Boehm [1981] proposed a formula for estimating maintenance costs using Software Maturity Index, SMI. It is defined as:

$$SMI = [Mt-(Fc + Fa + Fd)/ Mt]$$

Mt = the number of modules in the current release (version)

Fc = the number of modules in the current release that have been changed

Fa = the number of modules that are added in the current version

Fd = the number of modules deleted in the current version

When SMI reaches to the value 1.0 the product becomes more and more stabilized. This SMI metrics is used for planning the software maintenance activities.

V- Unit
Metrics for Process and Projects

Software measurements:

- Software measurement is a measure of software characteristics which are measurable or countable. Software measurements are valuable for many reasons, including measuring software performance, planning work items, measuring productivity, and many other uses.
- Software measurements can be categorized as Direct measures and Indirect measures.
- Direct measures of the software process include cost and effort applied. Direct measures of the product, include lines of code (LOC) produced, the cost and effort required to build software, execution speed, memory size, and defects reported over some set period of time.
- Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, and maintainability.

Classification of Software Measurements:

These are Product, Process and Project metrics:

1. Product Metrics: These are the measures of various characteristics of the software product. A product's size, design, performance, quality, and complexity are defined by product metrics. Developers can improve the quality of their software development by utilizing these features.

2. Process Metrics: These are the measures of various characteristics of the software development life cycle (SDLC)process. A project's characteristics and execution are defined by process metrics. These are used to measure the characteristics of methods, techniques, and tools that are used for developing software.

3. Project metrics: Project metrics are the metrics used by the project manager to check the project's progress.

- Data from the past projects are used to collect various metrics, like time and cost; these estimates are used as a base of new software.
- The project manager will check its progress from time-to-time and will compare the effort, cost, and time with the original effort, cost and time.
- These metrics are used to decrease the development costs, time efforts and risks.
- With the project metrics the project quality can also be improved.
- As quality improves, the number of errors and time, as well as cost required, is also being reduced.

- Product metrics are further classified into size oriented metrics.

a). Size Oriented Metrics:

- Size of the product is measured in LOC (line of code).
- It is one of the earliest and simpler metrics for calculating the size of the computer program. It is generally used in calculating and comparing the productivity of programmers.
- These metrics are used to find the quality and productivity of the product by using the size of the product as a metric.

Following are the points regarding LOC measures:

1. It is an older method that was developed when FORTRAN and COBOL programming were very popular.
2. Productivity is defined as Kilo lines of code / effort i.e., KLOC / EFFORT, where effort is measured in how many moths a person has worked.
3. Size-oriented metrics depend on the programming language used.
4. As productivity depends on KLOC, so assembly and C language code will have more productivity.
5. LOC measure requires a level of detail which may not be practically achievable.
6. The more expressive is the programming language, the lower is the productivity.
7. LOC method of measurement does not apply to projects that deal with visual (GUI-based) programming. User Interfaces (GUIs) use forms basically. LOC metric is not applicable here.
8. These metrics are not universally accepted.
9. While counting lines of code don't count blank lines and don't count comments.

Based on the LOC/KLOC count of software, the following size oriented measures are computed:

- Quality = No of defects/ KLOC
- Cost = $/KLOC
- Documentation = Pages of documentation/KLOC
- Errors = No.of Errors / person-month
- Productivity = LOC / person-month

Here KLOC means Thousands of Lines of Code.

Advantages of LOC:

1. Simple to measure

Disadvantage of LOC:

1. It is defined on the code. For example, it cannot measure the size of the specification.

2. It characterizes only one specific view of size, namely length, it takes no account of functionality or complexity
3. Bad software design may cause an excessive line of code
4. It is language dependent
5. Users cannot easily understand it

b).Function oriented metric:

- It is based on functionality of the delivered application.
- It is proposed by Albrecht in 1979 for IBM
- Function points (FP) are derived using countable measures of the software requirements to evaluate the software complexity.

Once the FP is calculated then we can compute various measures such as

- Productivity=FP/Person-month
- Quality=Number of faults/FP
- Cost=$/FP
- Documentation=pages of documentation/FP

Metrics for Software Quality:

- The overriding goal of software engineering is to produce a high-quality system, application, or product within a time frame that satisfies a market need.
- The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors.

a).Measuring Quality:

- There are many measures of software quality such as correctness, maintainability, integrity, and usability.
- These provide useful indicators for the project team

Correctness:

- Correctness is the degree to which the software performs its required function.
- The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
- Defects are those problems reported by a user of the program after the program has been released for general use.
- For quality assessment purposes, defects are counted over a standard period of time, typically one year.

Maintainability:

- Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment

changes, or enhanced if the customer desires a change in requirements.

- *Mean -time-to-change (MTTC), the time it takes to analyze the change* request, design an appropriate modification, implement the change, test it, and distribute the change to all users.

Integrity:

- Software integrity has become increasingly important in the age of cyber terrorists and hackers.
- Attacks can be made on all three components of software: programs, data, and documentation.
- To measure integrity, two additional attributes must be defined: threat and security.
- *Threat* is the probability (which can be estimated or derived from observed evidence) that an attack of a specific type will occur within a given time.
- *Security* is the probability (which can be estimated or derived from experiential evidence) that the attack of a specific type will be repelled.
- The integrity of a system can then be defined as

  integrity = summation [(1 – threat) × (1 – security)]

- where threat and security are summed over each type of attack.

Usability:

- If a program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable
- Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics: (1) the physical and or intellectual skill required to learn the system, (2) the time required to become moderately efficient in the use of the system, (3) the net increase in productivity measured when the system is used by someone who is moderately efficient, and (4) a subjective assessment.

b). Defect Removal Efficiency:

- Defect removal efficiency provides benefits at both the project and process level
- It is a measure of the filtering ability of quality assurances activities as they are applied throughout all process framework activities
- It indicates the percentage of software errors found before software release
- It is defined as DRE = E / (E + D)
- E is the number of errors found before delivery of the software to the end user
- D is the number of defects found after delivery
- As D increases, DRE decreases (i.e., becomes a smaller and smaller fraction)
- The ideal value of DRE is 1, which means no defects are found after delivery
- DRE encourages a software team to institute techniques for finding as many errors as possible before delivery.

RISK:

- A Risk is a Hazard. It is any real or potential condition that may be a damage to or loss of a system, equipment or property; or damage to the environment.
- A risk can lead to one or several consequences.
- A Risk is
  - The expectation of a loss or damage (consequence)
  - The combined severity and probability of a loss
  - The long term rate of loss
- A potential problem (leading to a loss) that may - or may not occur in the future.
- Treating a risk means understanding it better, avoiding or reducing it (risk mitigation), or preparing for the risk to materialize.
- Risk Management is a set of practices and support tools to identify, analyze, and treat risks explicitly.
- Risk management tries to reduce the probability of a risk to occur and the impact (loss) caused by risks.

REACTIVE VERSUS PROACTIVE RISK STRATEGIES:

- The majority of software teams rely exclusively on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems.

- The software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire-fighting mode.
- A considerably more intelligent strategy for risk management is to be proactive.
- A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then,
- The software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

Risk always involves two characteristics:

- *Uncertainty*—the risk may or may not happen; that is, there are no 100% probable risks.
- *Loss*—if the risk becomes a reality, unwanted consequences or losses will occur.
- When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk

Software Risks:

Different categories of software risks are follows:

- *Project risks* threaten the project plan. That is, if project risks become real, it is likely that project schedule will fall and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.
- *Technical risks* threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impos- sible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors.
- *Business risks* threaten the viability of the software to be built.

Business risks often jeopardize the project or the product.

- Top five business risks are:
- Building a excellent product or system that no one really wants (market risk),
- Building a product that no longer fits into the overall business strategy for the company (strategic risk),
- Building a product that the sales force doesn't understand how to sell (sales risk)
- Losing the support of senior management due to a change in focus or a change in people (management risk), and
- Losing budgetary or personnel commitment (budget risks).

Another general categorization of risks has been proposed by Charette.

- *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being devel- oped, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).
- *Predictable risks* are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).
- *Unpredictable risks* can occur, but they are extremely difficult to identify in advance.

RISK IDENTIFICATION:

- Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.).
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.
- There are two distinct types of risks:
  Generic risks and Product-specific risks.
- Generic risks are a potential threat to every software project.
- Product-specific risks can be identified only by those with a clear

understanding of the technology, the people, and the environment that is specific to the software that is to be built.

- To identify product-specific risks, the project plan and the software statement of scope are examined, and an answer to the following question is developed: -What special characteristics of this product may threaten our project plan?
- One method for identifying risks is to create a risk item checklist.
- The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:
- Product size risks -associated with the overall size of the software to be built or modified.
- Business impact risks - associated with constraints imposed by management or the marketplace.
- Stakeholder characteristics risks -associated with the sophistication of the stakeholders and the developer‘s ability to communicate with stakeholders in a timely manner.
- Process definition risks -associated with the degree to which the software process has been defined and is followed by the development organization.
- Development environment risks -associated with the availability and quality of the tools to be used to build the product.
- Technology to be built risks -associated with the complexity of the system to be built and the newness of the technology that is packaged by the system.
- Staff size and experience risks -associated with the overall technical and project experience ofthe software engineers who will do the work.

i).Assessing Overall Project Risk:

The following questions have been derived from risk data obtained by surveying experienced softwareproject managers in different parts of the world.

1. Have top software and customer managers formally committed to support the project?
2. Are end users enthusiastically committed to the project and the system/ product to be built?
3. Are requirements fully understood by the software engineering team and its customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end users have realistic expectations?
6. Is the project scope stable?
7. Does the software engineering team have the right mix of skills?

8. Are project requirements stable?
9. Doestheprojectteamhaveexperiencewiththetechnologytobeimplemented?
10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on therequirements for the system/product to be built the project manager identify the risk drivers that affect software risk components— performance,cost, support, and schedule.

ii).The risk components are defined in the following manner:

- Performance risk—the degree of uncertainty that the product will meet its requirements and be fitfor its intended use.
- Cost risk—the degree of uncertainty that the project budget will be maintained.
- Support risk—the degree of uncertainty that the resultant software will be easy to correct, adapt,and enhance.
- Schedule risk—the degree of uncertainty that the project schedule will be maintained and that theproduct will be delivered on time.
- The impact of each risk driver on the risk component is divided into one of four impactcategories—negligible, marginal, critical, or catastrophic.

RISK PROJECTION:

- Risk projection, also called risk estimation, attempts to rate each risk in two ways.
  1. The likelihood or probability or chance that the risk is real and
  2. The consequences (penalty) of the problems associated with the risk, should it occur

Managers and technical staff to perform four risk projection steps:

1. Establish a scale that reflects the chance of a risk.
2. Describe the consequences of the risk.
3. Estimate the impact of the risk on the project and the product.
4. Evaluate the overall accuracy of the risk projection so that there will be no misunderstandings.

The intent of these steps is to consider risks in a manner that leads to prioritization. No software team has the resources to address every possible risk with the same degree of rigor. By prioritizing risks, you can allocate resources where they will have the most impact.

i).Developing a Risk Table

- A risk table provides you with a simple technique for risk projection.

- A sample risk table is illustrated in the given Figure.
- List all the risks in the first column of the table.
- Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk).
- The probability of occurrence of each risk is entered in the next column of the table.
- The probability value for each risk can be estimated by team members individually.
- Next, the impact of each risk is assessed. Each risk component is assessed, and an impact category is determined.
- The categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.
- Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact.
- High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom.

Sample Risk table prior to sorting

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
Σ				
Σ				
Σ				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

ii).Assessing Risk Impact

- Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing.
- The nature of the risk indicates the problems that are likely if it occurs.
- For example, a poorly defined external interface to customer hardware (a technical risk) will prevent early design and testing

and will likely lead to system integration problems late in a project.

- The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many stakeholders are harmed?).
- The timing of a risk considers when and for how long the impact will be felt. In most cases, you want the -bad news‖ to occur as soon as possible, but in some cases, the longer the delay, the better.
- The overall risk exposure RE is determined using the following relationship

$$RE = P * C$$

Where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

RISK REFINEMENT:

- It may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.
- One way to do this is to represent the risk in *condition-transition-consequence* (CTC) format. That is, the risk is stated in the following form:

<condition> then there is concern that (possibly) <consequence>.

- Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.
- This general condition can be refined in the following manner:
- Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.
- Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

- Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

Risk Mitigation, Monitoring, and Management (RMMM):

- To assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues:
  - Risk mitigation
  - Risk monitoring
  - Risk management and contingency planning
- If a software team adopts a proactive (practical) approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation.*
- To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are:
  - Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, and competitive job market).
  - Moderate those causes that are under the control before the project starts.
  - Organize project teams so that information about each development activity is widely dispersed.
  - Define documentation standards
  - Conduct peer reviews of all work
  - Assign a backup staff member for every critical technologist.

Risk monitoring:

- As the project proceeds, risk monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. The following factors can be monitored:
  - General attitude of team members based on project pressures.
  - The degree to which the team has worked together.
  - Interpersonal relationships among team members.
  - Potential problems with compensation and benefits.
  - The availability of jobs within the company and outside it.

Risk management and contingency planning*:*

- *Risk management and contingency planning* assumes that mitigation efforts have failed and that the risk has become a reality.
- Continuing the example, the project is well underway and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team.
- In addition, the project manager may temporarily refocus resources (and read just the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to "get up to speed."
- Those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode."
- This might include video-based knowledge capture, the development of "commentary documents," and/or meeting with other team members who will remain on the project.
- It is important to note that RMMM steps invite additional project cost.
- For example, spending the time to "backup" every critical technologist costs money.
- Part of risk management, therefore, is to evaluate when the benefits increased by the RMMM steps are outweighed by the costs associated with implementing them.
- The work performed during earlier risk analysis steps will help the planner to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure).
- For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don't fall into the critical 20 percent (the risks with highest project priority).

Software Quality:

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

The modern view of a quality associated with a software product several quality methods such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc. categories of users can easily invoke the functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Software Quality Management System

A quality management system is the principal methods used by organizations to provide that the products they develop have the desired quality.

A quality system subsists of the following:

Managerial Structure and Individual Responsibilities: A quality system is the responsibility of the organization as a whole. However, every organization has a sever quality department to perform various quality system activities. The quality system of an arrangement should have the support of the top management. Without help for the quality system at a high level in a company, some members of staff will take the quality system seriously.

Quality System Activities: The quality system activities encompass the following:

Auditing of projects

Review of the quality system

Development of standards, methods, and guidelines, etc.

Production of documents for the top management summarizing the effectiveness of the quality system in the organization.

Software Quality Assurance (SQA) is simply a way to assure quality in the software. It is the set of activities which ensure processes, procedures as well as standards are suitable for the project and implemented correctly. Software Quality Assurance is a process which works parallel to development of software. It focuses on improving the process of

development of software so that problems can be prevented before they become a major issue. Software Quality Assurance is a kind of Umbrella activity that is applied throughout the software process.

Software Quality Assurance has:

1. A quality management approach
2. Formal technical reviews
3. Multi testing strategy
4. Effective software engineering technology
5. Measurement and reporting mechanism

Major Software Quality Assurance Activities:

1. **SQA Management Plan:**
   Make a plan for how you will carry out the sqa through out the project. Think about which set of software engineering activities are the best for project. check level of sqa team skills.

2. **Set The Check Points:**
   SQA team should set checkpoints. Evaluate the performance of the project on the basis of collected data on different check points.

3. **Multi testing Strategy:**
   Do not depend on a single testing approach. When you have a lot of testing approaches available use them.

4. **Measure Change Impact:**
   The changes for making the correction of an error sometimes re introduces more errors keep the measure of impact of change on project. Reset the new change to change check the compatibility of this fix with whole project.

5. **Manage Good Relations:**
   In the working environment managing good relations with other teams involved in the project development is mandatory. Bad relation of sqa team with programmers team will impact directly and badly on project. Don’t play politics.

Benefits of Software Quality Assurance (SQA):

1. SQA produces high quality software.
2. High quality application saves time and cost.
3. SQA is beneficial for better reliability.
4. SQA is beneficial in the condition of no maintenance for a long time.
5. High quality commercial software increase market share of company.
6. Improving the process of creating software.
7. Improves the quality of the software.

Disadvantage of SQA:

There are a number of disadvantages of quality assurance. Some of them

include adding more resources, employing more workers to help maintain quality and so much more.

Evolution of Quality Management System

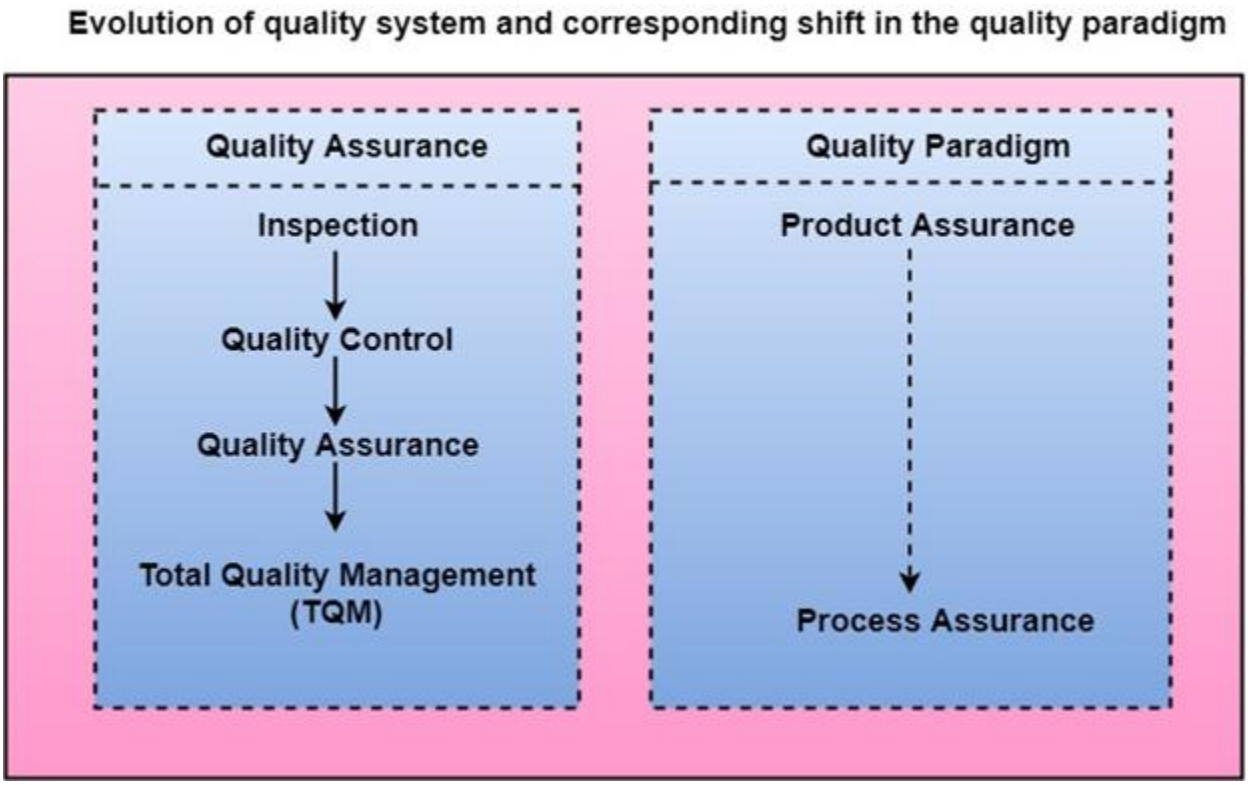
Quality systems have increasingly evolved over the last five decades. Before World War II, the usual function to produce quality products was to inspect the finished products to remove defective devices. Since that time, quality systems of organizations have undergone through four steps of evolution, as shown in the fig. The first product inspection task gave method to quality control (QC).

Quality control target not only on detecting the defective devices and removes them but also on determining the causes behind the defects. Thus, quality control aims at correcting the reasons for bugs and not just rejecting the products. The next breakthrough in quality methods was the development of quality assurance methods.

The primary premise of modern quality assurance is that if an organization's processes are proper and are followed rigorously, then the products are obligated to be of good quality. The new quality functions include guidance for recognizing, defining, analyzing, and improving the production process.

Total quality management (TQM) advocates that the procedure followed by an organization must be continuously improved through process measurements. TQM goes stages further than quality assurance and aims at frequently process improvement. TQM goes beyond documenting steps to optimizing them through a redesign. A term linked to TQM is Business Process Reengineering (BPR).

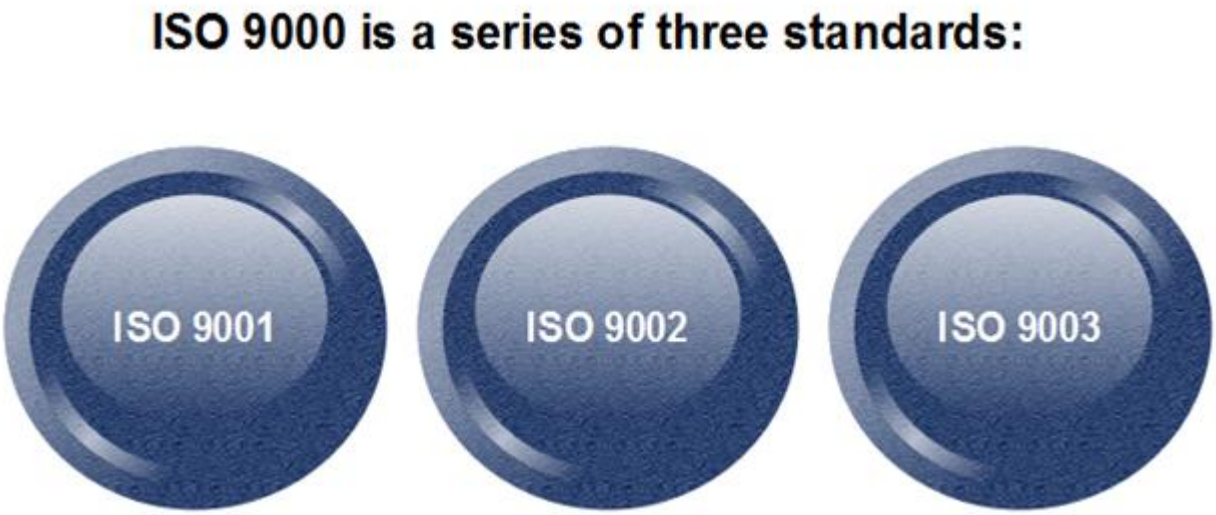
BPR aims at reengineering the method business is carried out in an organization. From the above conversation, it can be stated that over the years, the quality paradigm has changed from product assurance to process assurance, as shown in fig.



ISO 9000 Certification

ISO (International Standards Organization) is a group or consortium of 63 countries established to plan and fosters standardization. ISO declared its 9000 series of standards in 1987. It serves as a reference for the contract between independent parties. The ISO 9000 standard determines the guidelines for maintaining a quality system. The ISO standard mainly addresses operational methods and organizational methods such as responsibilities, reporting, etc. ISO 9000 defines a set of guidelines for the production process and is not directly concerned about the product itself.

Types of ISO 9000 Quality Standards



The ISO 9000 series of standards is based on the assumption that if a proper stage is followed for production, then good quality products are

bound to follow automatically. The types of industries to which the various ISO standards apply are as follows.

1. **ISO 9001:** This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.
2. **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing those products. Therefore, ISO 9002 does not apply to software development organizations.
3. **ISO 9003:** This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

How to get ISO 9000 Certification?

An organization determines to obtain ISO 9000 certification applies to ISO registrar office for registration. The process consists of the following stages:

1. **Application:** Once an organization decided to go for ISO certification, it applies to the registrar for registration.
2. **Pre-Assessment:** During this stage, the registrar makes a rough assessment of the organization.
3. **Document review and Adequacy of Audit:** During this stage, the registrar reviews the document submitted by the organization and suggest an improvement.
4. **Compliance Audit:** During this stage, the registrar checks whether the organization has compiled the suggestion made by it during the review or not.

5. **Registration:** The Registrar awards the ISO certification after the successful completion of all the phases.
6. **Continued Inspection:** The registrar continued to monitor the organization time by time.