# Software Engineering
## Unit 2- Part 2

**Topics**

1. **Software Design**
    a. **Design process**
    b. **Design Characteristics**
    c. **Design Concepts**
2. **Design Models**

**a. Architectural Design- Software approach, Data design, Architectural styles and patterns, architectural design.**

**b. Object Oriented Design- Objects and Classes, Design process and design evaluation.**

**3. User Interface Design- Interface Analysis, Interface Design steps and design evolution.**

**What is Design and Why is it important?**

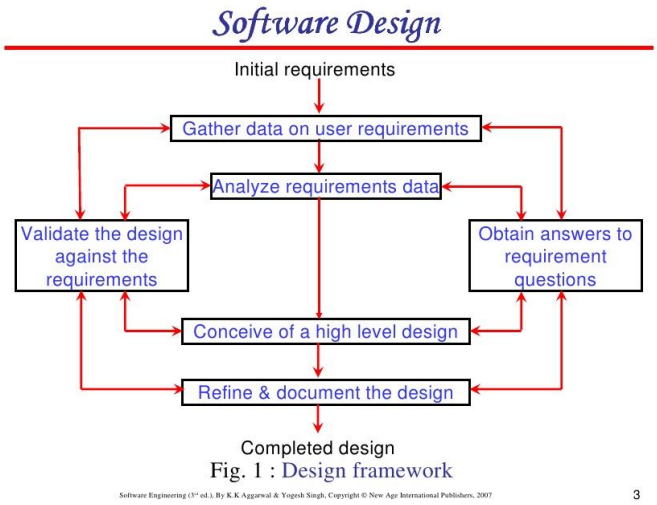
Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components. Computer software is considerably more complex than a house; hence, we need a blueprint which is the design document.

**What are the major steps of design?**

Design begins with the requirements model. We work to transform this model into four levels of design detail: 1. the data structure, 2. the system architecture, 3. the interface representation, and 4. the component level detail. During each design activity, we apply basic concepts and principles that lead to high quality. The software design work products are reviewed for clarity, correctness, completeness, and consistency with the requirements and with one another.

## THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the design is represented at a high level of abstraction a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.




**There are two major phases to any design process:**

1. **Diversification**: Diversification is the *acquisition* of a collection of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind.

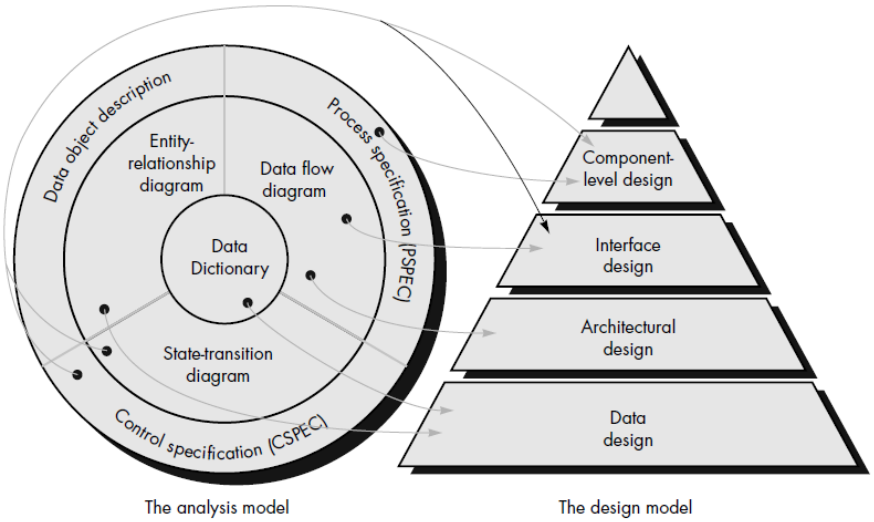
2. **Convergence**: During convergence, the designer chooses and combines appropriate elements from this collection to meet the design objectives, as stated in the requirements document and as agreed to by the customer.

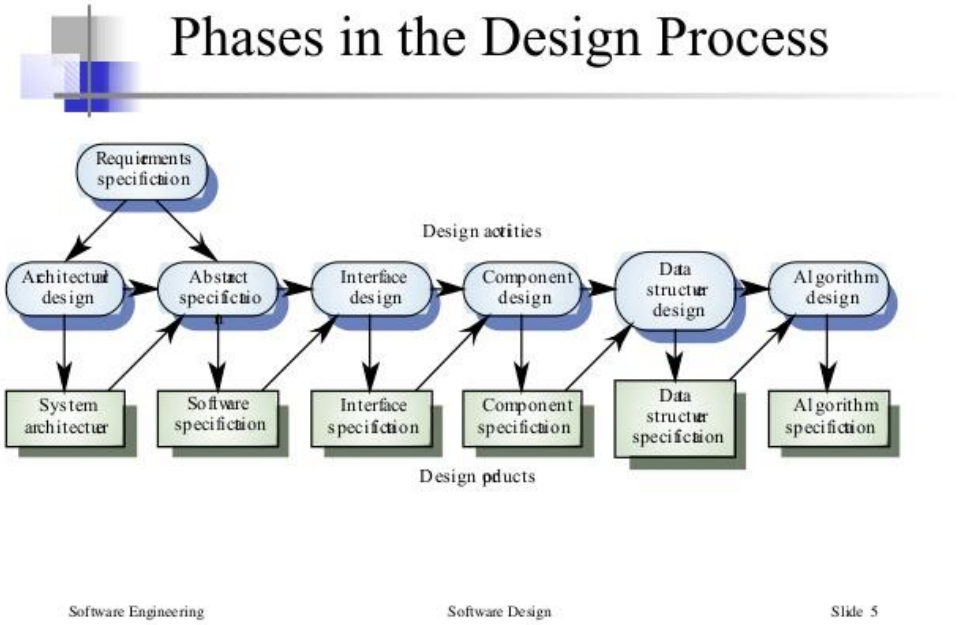
## SOFTWARE DESIGN AND SOFTWARE ENGINEERING

Using one of a number of design methods (like object oriented design), the design task produces a **data design, an architectural design, an interface design, and a component design**.
Where,

- The ***data design*** transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity.
- The ***architectural design*** defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system. The architectural design representation— the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.
- The ***interface design*** describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. data and control flow diagrams provide much of the information required for interface design.

- The ***component-level design*** transforms structural elements of the software architecture into a procedural description of software components.




**Characteristics that serve as a guide for the evaluation of a good design:**

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

- A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and subfunctions.
- A design should contain distinct representations of data, architecture, interfaces, and components (modules).
- A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.

**Design Principles:**

- Design process should consider alternative approaches.
- The design should be traceable to the analysis model.
- Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
- The structure of the software design should (whenever possible) mimic the structure of the problem domain.
- The design should exhibit uniformity and integration
- The design should be structured to accommodate change.

## DESIGN CONCEPTS

### Abstraction:

When we consider a modular solution to any problem, many *levels of abstraction* can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. As we move through different levels of abstraction, we work to create procedural and data abstractions. A *procedural abstraction* is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word *open* for a door. A *data abstraction* is a named collection of data that describes a data object In the context of the procedural abstraction *open*, we can define a data abstraction
called **door**.

### Refinement:

A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

### Modularity

A software is divided into separately named and addressable components, often called *modules,* that are integrated to satisfy problem requirements.

**Software Architecture**

In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components.

**Control Hierarchy**

*Control hierarchy,* also called *program structure,* represents the organization of program components (modules) and implies a hierarchy of control.

**Structural Partitioning**

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically.

**Data Structure**

*Data structure* is a representation of the logical relationship among individual elements of data.

**Software Procedure**

Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure

**Information Hiding**

Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module

## EFFECTIVE MODULAR DESIGN

A modular design reduces complexity facilitates change, and results in easier implementation by encouraging parallel development of different parts of a system.

A module has to be functionally independent which a key to good design is, and design is the key to software quality. Independence is measured using two qualitative criteria: **cohesion** and **coupling**.

- ***Cohesion*** is a measure of the relative functional strength of a module. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. High cohesion leads to the increased module reusability because the developers of the application will easily find the component they look for in the cohesive set of operations offered by the module. The module complexity also reduces, when there is a high cohesion in the programming.

  **Different types of cohesion:**

  **1. Coincidentally cohesive:** The subsystem in which the set of tasks are related with each other loosely then such subsystems are called coincidentally cohesive.

  **2. Logically cohesive**: A subsystem that performs the tasks that are logically related with each other is called logically cohesive.

**3. Temporal cohesive:** The subsystem in which the tasks need to be executed in some specific time span is called temporal cohesive.

**4. Procedural cohesive:** When processing elements of a subsystem are related with one another and must be executed in some specific order, such subsystems is called Procedural cohesive.

**5. Communication cohesion:** when the processing elements of a subsystem share the data then such subsystem is called communication cohesive.

**6. Sequential cohesion**: when the output of 1 subsystem is given as input for other subsystem is called Sequential cohesion.

- ***Coupling*** is a measure of interconnection among modules in a software structure.

Different types of coupling:

**1. Data coupling:** The data coupling is possible by parameter passing or data interaction.

**2. Control coupling:** The modules share related control data in control coupling.

**3. Common coupling:** In common coupling common data or global data is shared among the modules.

**4. Content coupling:** Content coupling occurs when one module makes use of data or control information maintained in another module.

**THE DESIGN MODEL**

The design principles and concepts discussed in this chapter establish a foundation for the creation of the design model that encompasses representations of data, architecture, interfaces, and components.

**ARCHITECTURAL DESIGN:**

Architectural design is the preliminary blueprint from which software is constructed. Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system. It is done by System Architect. The initial design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called architectural design. The output of this design process is a description of the software architecture.

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

**Why software architecture is important?**

- Representations of software architecture enables for communication between all parties (stakeholders) interested in the development of a computer- based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Large-scale reuse: the *architecture may be reusable* across a range of systems with similar requirements.

**Steps in Architectural design:**

1. Begins with **data design** and
2. Proceeds with the **architectural structure of the system** that is best suited for customers requirements.

**Data Design**

*Data design* (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

**Why data design is important?**

- At Program level: the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At Application level: the translation of a data model into a database is pivotal to achieving the business objectives of a system.
- At Business level: the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

**Data Modeling:**

The data objects defined during software requirements analysis are modeled using entity/relationship diagrams and the data dictionary. The data design activity translates these elements of the requirements model into data structures at the software component level and, when necessary, a database architecture at the application level.

**Data Modeling at Component Level:**

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components.

**ARCHITECTURAL STYLES**

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

(1) a set of ***components*** (e.g., a database, computational modules) that perform a function required by a system;

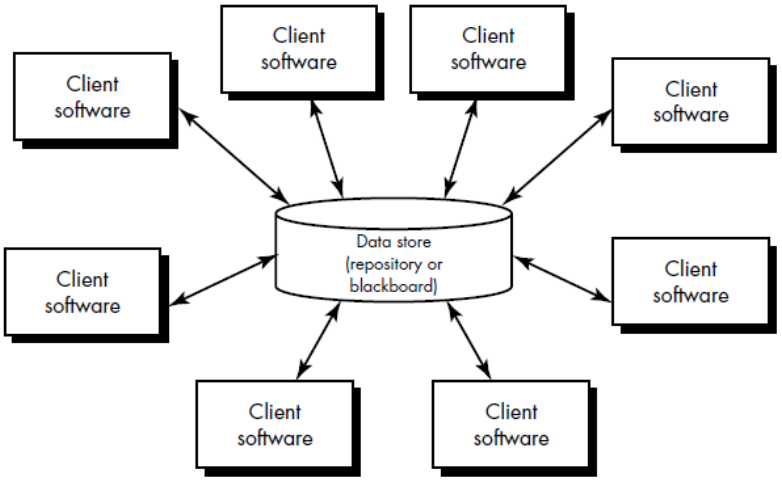
(2) a set of ***connectors*** that enable "communication, coordinations and cooperation" among components;

(3) ***constraints*** that define how components can be integrated to form the system; and

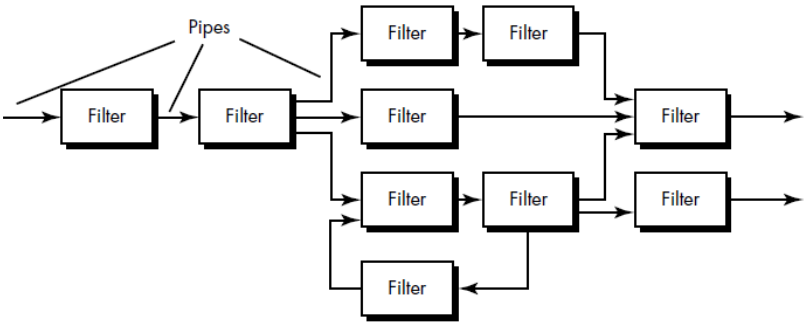
(4) ***semantic models*** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts

**Taxonomy/Types/Classification of Styles and Patterns**

**Data-centered architectures.** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. In this style existing components can be changed and new client components can be added.




**Data-flow architectures:** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A *pipe and filter pattern* (show in fig) has a set of components, called *filters*, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its neighboring filters.

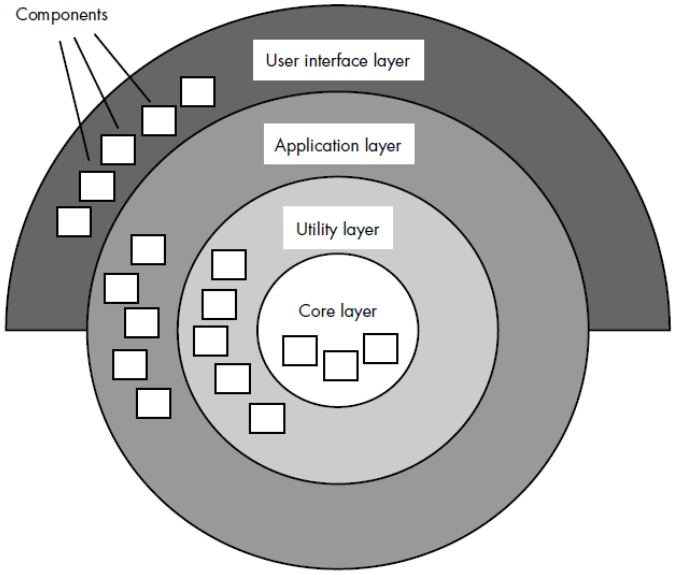



(a) Pipes and filters

**Call and return architectures: T**his architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. Ex: *Main program/subprogram architectures*. This classic program structure
decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.

**Object-oriented architectures.** The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

**Layered architectures.** The basic structure of a layered architecture is illustrated in fig. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



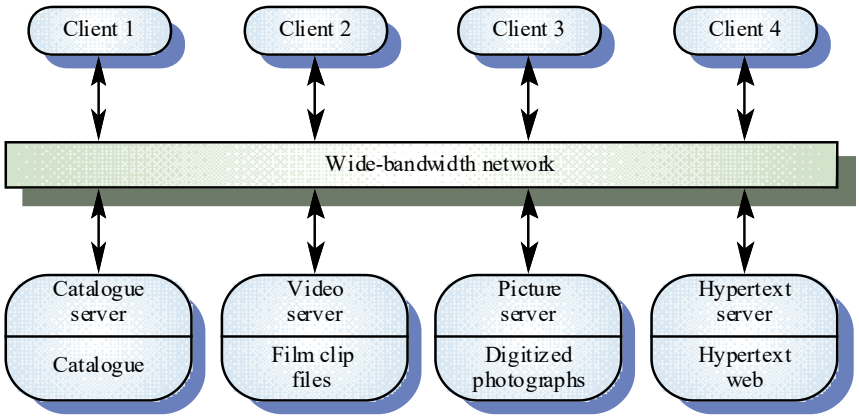
**The client-server mod**el:

The client-server architectural model is a system model where the system is organized as set of services and associated servers and clients that access and use the services. The major components of this model are:

A set of servers that offer services to other sub-systems.

A set of clients that call on the services offered by servers.

A network that allows the clients to access these services.




**Object Oriented Design**

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state.

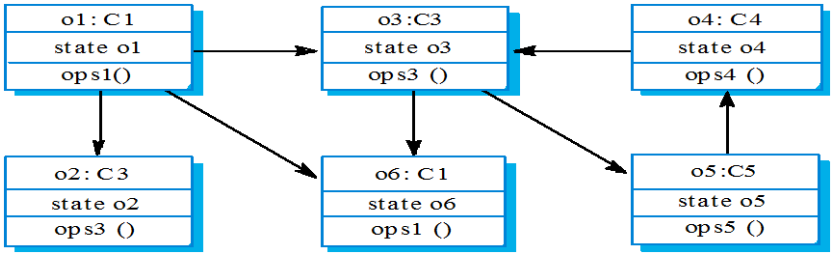



Fig: A system made up of interacting objects

- Object-oriented analysis, design and programming are related but distinct.
- OOA is concerned with developing an object model of the application domain.
- OOD is concerned with developing an object-oriented system model to implement requirements.
- OOP is concerned with realising an OOD using an OO programming language such as Java or C++.

**Characteristics of OOD**

- Objects are abstractions of real-world or system entities and manage themselves.
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services.

- Shared data areas are eliminated. Objects communicate by message passing (no global variables).
- Objects may be distributed and may execute sequentially or in parallel.

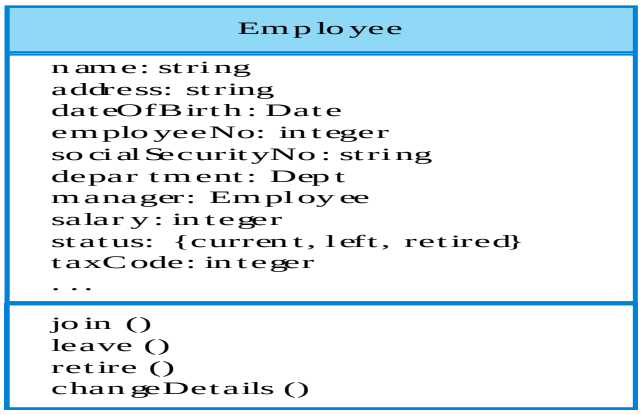
**Advantages of OOD**

- Easier maintenance. Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects.

**Objects and object classes**

- Objects are entities in a software system which represent instances of real-world and system entities.
- Object classes are templates for objects. They may be used to create objects.
- Object classes may inherit attributes and services from other object classes.

In the UML, an object class is represented as a named rectangle with two sections. The object attributes are listed in the top section. The operations that are associated with the object are set out in the bottom section




**Object communication**

- Conceptually, objects communicate by message passing.
- Messages
  - The name of the service requested by the calling object;
  - Copies of the information required to execute the service and the name of a holder for the result of the service.
- In practice, messages are often implemented by procedure calls
  Name = procedure name;
  Information = parameter list.

**Generalisation and inheritance**

- Classes may be arranged in a class hierarchy where one class (a super-class) is a generalisation of one or more other classes (sub-classes).

- A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own.
- Generalisation in the UML is implemented as inheritance in OO programming languages.

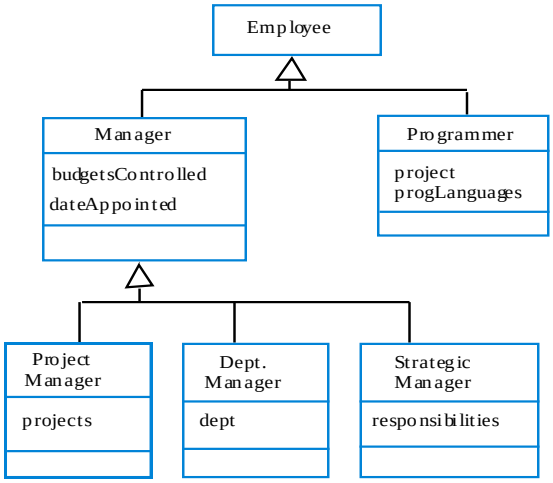



Fig: A generalisation hierarchy

**Advantages of inheritance:**

- It is an abstraction mechanism which may be used to classify entities.
- It is a reuse mechanism at both the design and the programming level.
- Minimize code duplication.

**Object-oriented design process:**

**Stages in design process:**

1. **Understand and define the context and the models of use of the system:**
The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. The system context is a static model that describes the other systems in that environment. The model of the system use is a dynamic model that describes how the system actually interacts with its environment.
2. **Design the system architecture:**
Once the interactions between the software system that is being designed and the systems, environment have been defined, you can use this information as a basis for designing the system architecture.
3. **Identify the principal objects in the system.**

**Design models:**

Design models show the objects and object classes and relationships between these entities. There are two types of design models that should normally be produced to describe an object-oriented design:

1. **Static models** describe the static structure of the system in terms of object classes and relationships.
2. **Dynamic models** describe the dynamic interactions between objects.

**Examples of design models**

- Sub-system models that show logical groupings of objects into coherent subsystems.
- Sequence models that show the sequence of object interactions.
- State machine models that show how individual objects change their state in response to events.
- Other models include use-case models, aggregation models, generalisation models, etc.

**Design evolution**

An important advantage of an object-oriented approach to design is that it simplifies the problem of making changes to the design. The reason for this is that object state representation does not influence the design. Changing the internal details of an object is unlikely to affect any other system objects.

# Software Engineering
## Unit 1

**What is software?**

- Computer programs and associated documentation such as requirements, design models and user manuals.
- Software products may be developed for a particular customer or may be developed for a general market.

**Types of Software products**

1. Generic Software – developed to be sold to a range of different customers
e.g. PC software such as Excel or Word.
2. Bespoke Software (custom) - developed for a single customer according to their specification.

New software can be created by developing new programs, configuring generic software systems or reusing existing software.

**What is software engineering?**

Software engineering is an engineering discipline that is concerned with all aspects of software production. So as to develop the software with the highest quality. Software engineers should adopt a systematic and organised approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.

**What is the difference between software engineering and computer science?**

Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.

**What is the difference between software engineering and system engineering?**

System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this process concerned with developing the software infrastructure, control, applications and databases in the system.

**NEED OF SOFTWARE ENGINEERING**

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

**Large software -** It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

**Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

**Cost-** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

**Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new

enhancements need to be done in the existing one. This is where software engineering plays a good role.

**Quality Management-** Better process of software development provides better and quality software product.

**What is a software process?**

A set of activities whose goal is the development or evolution of software.

Generic activities in all software processes are:

- Specification - what the system should do and its development constraints
- Development - production of the software system
- Validation - checking that the software is what the customer wants
- Evolution - changing the software in response to changing demands.

**What is a software process model?**

A simplified representation of a software process, presented from a specific perspective.

Generic process models

- Waterfall;
- Iterative development;
- Component-based software engineering.

**What are the costs of software engineering?**

Roughly 60% of costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.

Costs vary depending on the type of system being developed and the requirements of system attributes such as performance and system reliability.

**What are software engineering methods?**

Structured approaches to software development which include system models, notations, rules, design advice and process guidance.

Model descriptions -Descriptions of graphical models which should be produced;

Rules- Constraints applied to system models;

Recommendations- Advice on good design practice;

Process guidance-What activities to follow.

**What is CASE?**

Computer-Aided Software Engineering. CASE tools are software systems which are designed to support routine activities in the software process such as editing design diagrams, checking diagram consistency and keeping track of program tests which have been run.

**What are the attributes of good software?**

The software should deliver the required functionality and performance to the user and should be maintainable, dependable and acceptable.

- **Maintainability**-Software must evolve to meet changing needs;
- **Dependability**- Software must be trustworthy;
- **Efficiency-**Software should not make wasteful use of system resources;
- **Acceptability**- Software must accepted by the users for which it was designed. This means it must be understandable, usable and compatible with other systems.

**What are the key challenges facing software engineering?**

**Heterogeneity**- Developing techniques for building software that can cope with heterogeneous platforms and execution environments;

**Delivery**- Developing techniques that lead to faster delivery of software;

**Trust**- Developing techniques that demonstrate that software can be trusted by its users.

**Professional and ethical responsibility**

- Software engineering involves wider responsibilities than simply the application of technical skills.
- Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.
- Ethical behaviour is more than simply upholding the law.
- **Confidentiality** - Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
- **Competence** - Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.
- **Intellectual property rights**- Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
- **Computer misuse** -Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

**Software Process Generic Activities**

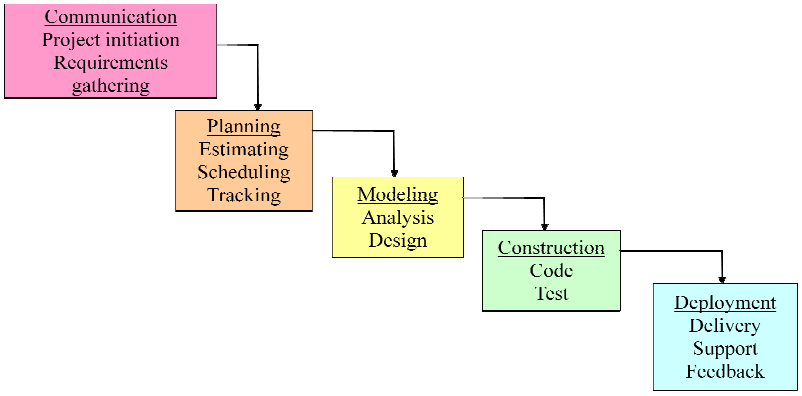
- **Communication**
  - Involves communication among the customer and other stake holders; encompasses requirements gathering
- **Planning**
  - Establishes a plan for software engineering work; addresses technical tasks, resources, work products, and work schedule
- **Modeling** (Analyze, Design)
  - Encompasses the creation of models to better understand the requirements and the design
- **Construction** (Code, Test)

  - Combines code generation and testing to uncover errors
- **Deployment**
  - Involves delivery of software to the customer for evaluation and feedback

**Umbrella Activities Involved in Software Development**

- Software requirements management
- Software project planning
- Software project tracking and oversight
- Software quality assurance
- Software configuration management
- Software subcontract management
- Formal technical reviews
- Risk management
- Measurement – process, project, product
- Reusability management (component reuse)
- Work product preparation and production

**Waterfall Model**




- Oldest software lifecycle model and best understood by upper management.
- Used when requirements are well understood and risk is low.
- Work flow is in a linear (i.e., sequential) fashion.
- Used often with well-defined adaptations or enhancements to current software.

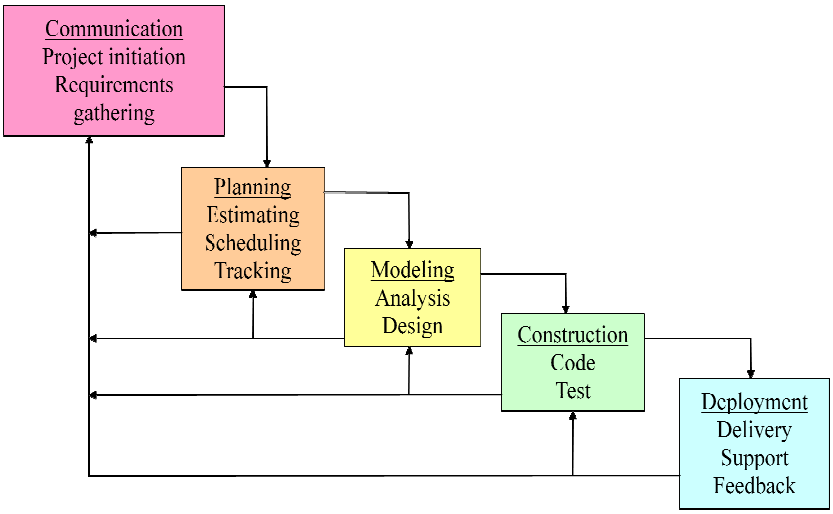
**Advantages:**

- Simple and easy to implement.
- Straightforward.
- Disciplined approach.

**Disadvantages:**

- Without completing one stage the next stage cannot be started.
- Cannot be used for changing requirements
- If any error committed in the previous stage then it cannot be rectified.
- Customer sees the product only at the end.

**Waterfall Model with Feedback (Iterative Waterfall Model)**

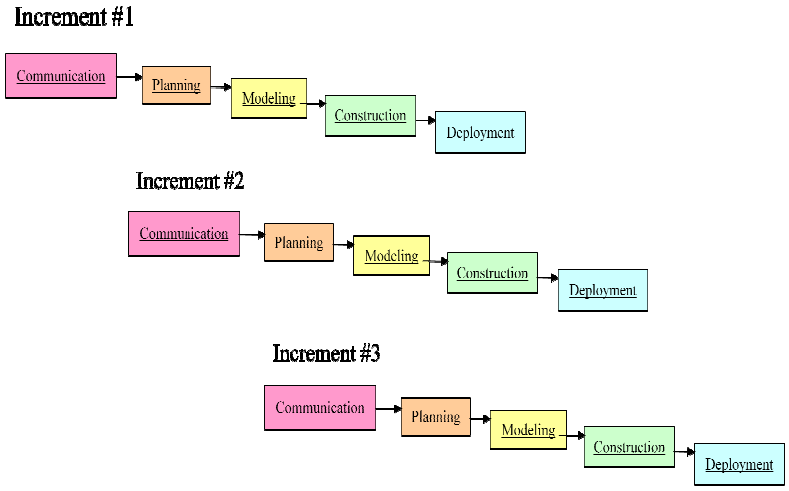



In this model a feedback after every stage can be collected to check if there are any shortcomings if so the same stage is repeated(iterated) to overcome the changes. This model overcomes the drawback of undoing the changes in the previous stage of classical waterfall model.

**Advantages:**

- Error can be rectified is any of the stage.
- Simple and easy to implement.
- Straightforward.
- Disciplined approach.

**Incremental Model**



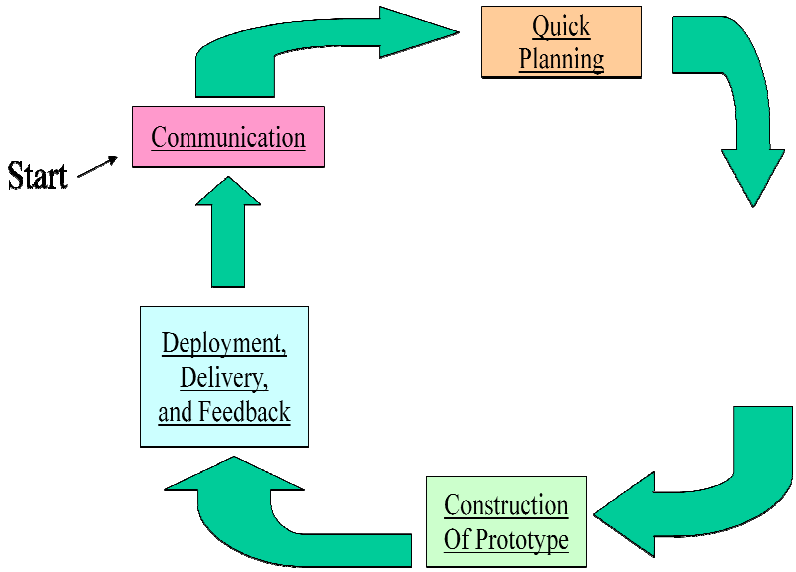

- Used when requirements are well understood
- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality
- Work flow is in a linear (i.e., sequential) fashion within an increment and is staggered between increments
- Iterative in nature; focuses on an operational product with each increment

- Provides a needed set of functionality sooner while delivering optional components later
- Useful also when staffing is too short for a full-scale development

**Advantages:**

- The customer is able to do some useful work after release
- Lower risk of overall project failure
- The highest priority system services tend to receive the most testing

**Prototyping Model**




The Software Prototyping refers to building software application prototypes which displays the functionality of the product under development, but may not actually hold the exact logic of the original software.

- Follows an linear and iterative approach
- Used when requirements are not well understood
- Serves as a mechanism for identifying software requirements
- Focuses on those aspects of the software that are visible to the customer/user
- Feedback is used to refine the prototype

**Advantages:**

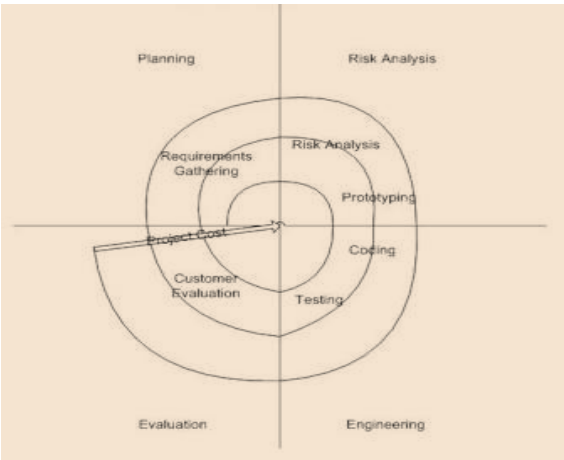
- Since a working model of the system is displayed, the users get a better understanding of the system being developed.
- It enables to understand customer requirements at an early stage of development.
- It helps get valuable feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.
- Missing functionality can be identified easily.
- Confusing or difficult functions can be identified.

**Disadvantages:**

- Users may get confused in the prototypes and actual systems.

- Developers often make implementation compromises to get the software running quickly (e.g., language choice, user interface, operating system choice, inefficient algorithms)
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- The effort invested in building prototypes may be too much if it is not monitored properly.

**Spiral Model**




- Invented by Dr. Barry Boehm in 1988 while working at TRW
- Follows an evolutionary approach
- Used when requirements are not well understood and risks are high
- Inner spirals focus on identifying software requirements and project risks; may also incorporate prototyping
- Outer spirals take on a classical waterfall approach after requirements have been defined, but permit iterative growth of the software
- Operates as a risk-driven model…a go/no-go decision occurs after each complete spiral in order to react to risk determinations
- Requires considerable expertise in risk assessment
- Serves as a realistic model for large-scale software development

The spiral model combines the idea of iterative development with the systematic, controlled aspects of the waterfall model. This Spiral model is a combination of iterative development process model and sequential linear development model i.e. the waterfall model with a very high emphasis on risk analysis. It allows incremental releases of the product or incremental refinement through each iteration around the spiral.

**Advantages of Spiral model:**

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the **software life cycle**.

**Disadvantages of Spiral model:**

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

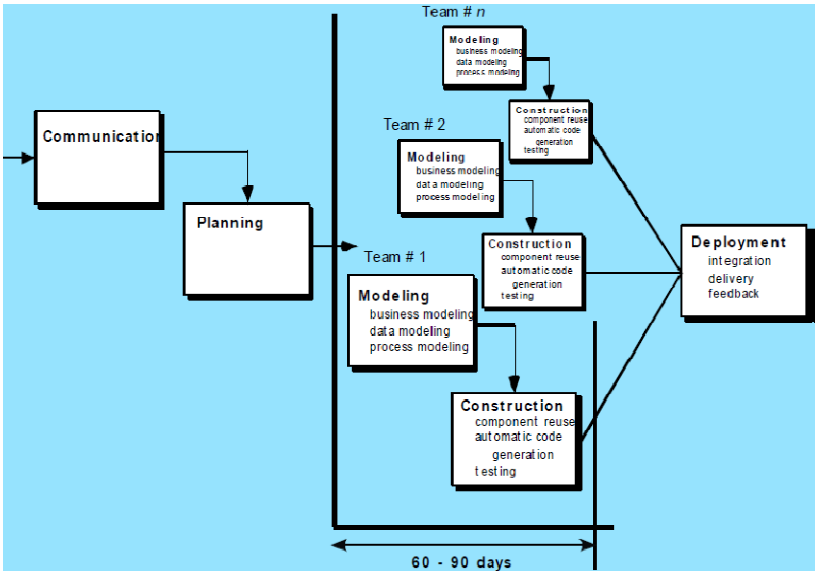
**When to use Spiral model:**

- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

**RAD Model (Rapid Application Development)**

Rapid Application Development is a linear sequential software development process model that emphasizes an extremely short development cycle. Rapid application achieved by using a component based construction approach. If requirements are well understood and project scope is constrained the RAD process enables a development team to create a —fully functional system

RAD projects follow iterative and incremental model and have small teams comprising of developers, domain experts, customer representatives and other IT resources working progressively on their component or prototype.




**Following are the various phases of the RAD Model –**

**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.

**Advantages**

- Reduced development time.
- Progress can be measured.
- Uses component-based construction and emphasizes reuse and code generation

**Disadvantages**

- Large human resource requirements (to create all of the teams).
- Requires highly skilled developers/designers.
- Management complexity is more.
- Requires strong commitment between developers and customers for “rapid-fire” activities.
- High performance requirements maybe can’t be met (requires tuning the components).

# Software Engineering
## Unit 1- Part 2 Requirement Engineering

**Topics:**

1. **Functional and Non Functional requirements**
2. **User Requirements**
3. **System Requirements**
4. **Software Requirement Specification Documents (SRS)**
5. **Feasibility Studies**
6. **Requirement Elicitation and Analysis**
7. **Requirement Validation**
8. **Requirement Management**

### Requirement Engineering

Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team carry out this job. They are called as ***system analysts***. The analyst starts *requirements gathering and analysis* activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to remove all ambiguities and inconsistencies from the initial customer perception of the problem

### What is Requirement and Requirement Engineering?

Requirements specify how the target system should behave. It specifies what to do, but not how to do. Requirements engineering refers to the process of understanding what a customer expects from the system to be developed, and to document them in a standard and easily readable and understandable format. This documentation will serve as reference for the subsequent design, implementation and verification of the system.

### Characteristics of Requirements

Requirements gathered for any new system to be developed should exhibit the following three properties:

**Unambiguity:** There should not be any confusion what a system to be developed should do. For example, consider you are developing a web application for your client. The client requires that enough number of people should be able to access the application simultaneously. What's the "enough number of people"? That could mean 10 to you, but, perhaps, 100 to the client. There's an ambiguity.

**Consistency:** The requirements gathered should be consistent. To illustrate this, consider the automation of a nuclear plant. Suppose one of the clients say that it the radiation level inside the plant exceeds R1, all reactors should be shut down. However, another person from the client side suggests that the threshold radiation level should be R2. Thus, there is an inconsistency between the two end users regarding what they consider as threshold level of radiation.

**Completeness:** All the requirements pertaining to a system need to be gathered. A particular requirement for a system should specify what the system should do and also what it should not. For example, consider a software to be developed for ATM. If a customer enters an amount greater than the maximum permissible withdrawal amount, the ATM should display an error message, and it should not dispense any cash.

**Categorization of Requirements**

Based on the target audience or subject matter, requirements can be classified into different types, as stated below:

- **User requirements:** They are written in natural language so that both customers can verify their requirements have been correctly identified

**Problems with natural language** while documenting the user requirements

1. Lack of clarity: Difficult to collect the requirements precisely without any confusions.
2. Requirement confusions: The FR, NFR, Design information many not be clearly documented.
3. Requirement amalgamation: Different requirements many be expressed together as a single requirement.

- **System requirements:** They are written involving technical terms and/or specifications, and are meant for the development or testing teams. These are expanded version of the user requirements. They add detail and explain how the used requirements should be provided by the system.

Requirements can be classified into two groups based on what they describe:

- **Functional requirements (FRs):** These describe the functionality of a system -- how a system should react to a particular set of inputs and what should be the corresponding output.
- **Non-functional requirements (NFRs):** They are not directly related what functionalities are expected from the system. However, NFRs could typically define how the system should behave under certain situations. For example, a NFR could say that the system should work with 128MB RAM. Under such condition, a NFR could be more critical than a FR.
- **Domain requirements (DR)**: Requirements that come from the application domain of the system and that reflect characteristics of that domain. Ex. DR for Libsys: Because of copyright restrictions, some documents must be deleted immediately on arrival.

**Functional Requirements**

**Identifying Functional Requirements**

Given a problem statement, the functional requirements could be identified by focusing on the following points:

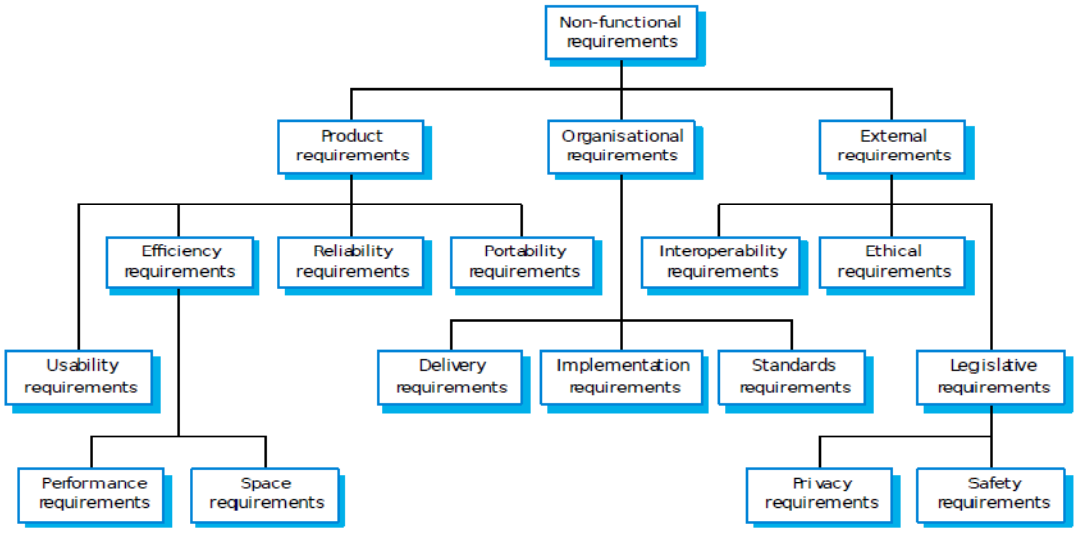
- Identify the high level functional requirements simply from the conceptual understanding of the problem. For example, a Library Management System, apart from anything else, should be able to issue and return books.
- Identify the cases where an end user gets some meaningful work done by using the system. For example, in a digital library a user might use the "Search Book" functionality to obtain information about the books of his interest.
- If we consider the system as a black box, there would be some inputs to it, and some output in return. This black box defines the functionalities of the system. For example, to search for a book, user gives title of the book as input and get the book details and location as the output.
- Any high level requirement identified could have different sub-requirements. For example, "Issue Book" module could behave differently for different class of users, or for a particular user who has issued the book thrice consecutively.

**Non-functional requirements could be further classified into different types like:**

- **Product requirements:** For example, a specification that the web application should use only plain HTML, and no frames
- **Performance requirements:** For example, the system should remain available 24x7

- **Organizational requirements:** The development process should comply to SEI CMM level 4

**Types of NFR:**




**Metrics for specifying the NFR:**

**Requirements measures**

| Property | Measure |
|---|---|
| Speed | Processed transaction/second |
| | User/Event response time |
| | Screen refresh time |
| Size | k bytes Number of RAM chips |
| Ease of use | Training time Number of help frames |
| Reliability | Mean time to failure |
| | Probability of unavailability |
| | Rate of failure occurance |
| | Availability |
| Robustness | Time to restart after failure |
| | Percentage of events causing failure |
| | Probability of data corruption on failure |
| Portability | Percentage of target-dependent statements |
| | Number of target systems |

**Notations for requirements specifications:**

- Structured natural language: This approach depends on defining standard forms or templates to express the requirements specification.
- Design description languages: This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system. This approach is not now widely used although it can be useful for interface specifications.
- Graphical notations: A graphical language, supplemented by text annotations is used to define the functional requirements for the system. Ex: sequence diagrams
- Mathematical: These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

**Example of FR and NFR for a library software:**

**FR:**

| | | |
|---|---|---|
| R1 | New user registration | High |
| R2 | User Login | High |
| R3 | Search book | High |
| R4 | Issue book | High |
| R5 | Return book | High |
| R6 | Reissue book | Low |

**NFR:**

- **Performance Requirements:**
  - This system should remain accessible 24x7
  - At least 50 users should be able to access the system altogether at any given time
- **Security Requirements:**
  - This system should be accessible only within the institute LAN

    - The database of LIS should not store any password in plain text -- a hashed value has to be stored
- **Software Quality Attributes**
- **Database Requirements**
- **Design Constraints:**
    - The LIS has to be developed as a web application, which should work with Firefox 5, Internet Explorer 8, Google Chrome 12, Opera 10
    - The system should be developed using HTML 5

Once all the functional and non-functional requirements have been identified, they are documented formally in SRS, which then serves as a legal agreement.

**The Software Requirements Specifications Document:**

Once all possible functional and non-functional requirements have been identified, which are complete, consistent, and non-ambiguous, the Software Requirements Specification (SRS) is to be prepared. The requirement document has a diverse set of users such as customers, managers, system engineers, test engineers and maintenance engineers.

The SRS is prepared by the service provider, and verified by its client. This document serves as a legal agreement between the client and the service provider. Once the concerned system has been developed and deployed, and a proposed feature was not found to be present in the system, the client can point this out from the SRS. Also, if after delivery, the client says a new feature is required, which was not mentioned in the SRS, the service provider can again point to the SRS. The scope of the current experiment, however, doesn't cover writing a SRS. IEEE format is most widely used format for preparing the SRS document.

**Structure of SRS Document:**

The IEEE standard suggests the following structure:

1. Table of contents
2. Preface
3. Introduction
4. Glossary
5. User requirements definition
6. System architecture
7. System requirements specification
8. System models
9. System evolution
10. Appendices
11. Index

A general structure of an SRS will be as follows:

1. **Introduction:**
   This provides an overview of the entire information described in SRS. This involves purpose and the scope of SRS, which states the functions to be performed by the system. In addition, it describes definitions, abbreviations, and the acronyms used. The references used in SRS provide a list of documents that is referenced in the document.
2. **Overall Description:** Describe the general factors that affect the product and its requirements. It should also be made clear that this section does not state specific requirements, it only makes those requirements easier to understand.

   It comprises the following sub-sections.

   1. **Product Perspective:** It determines whether the product is an independent product or an integral part of the larger product. It determines the interface with hardware, software, system, and communication. It also defines memory constraints and operations utilized by the user.
   2. **Product Functions:** Provide a summary of the functions that the software will perform.
   3. **User Characteristics:** Describe general characteristics of the users.
   4. **General Constraints:** It provides the genera1 description of the constraints such as regulatory policies, audit functions, reliability requirements, and so on.
   5. **Assumptions and Dependencies:** List each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption might be that a specific operating system will be available on the hardware designated for the software product. If, in fact, the operating system is not available, the SRS would then have to change accordingly.
3. **Specific Requirements:** Give the detailed requirements that are used to guide the project's software design, implementation, and testing. Each requirement in this section should be correct, unambiguous, verifiable, prioritized, complete, consistent, and uniquely identifiable. Attention should be paid to the carefully organize the requirements presented in this section

so that they may easily accessed and understood. Furthermore, this SRS is not the software design document, therefore one should avoid the tendency to over-constrain (and therefore design) the software project within this SRS.

1. **External Interface Requirements:** Include user, hardware, software, and communication interfaces.
2. **Functional Requirements:** Describes specific features of the software project. If desired, some requirements may be specified in the use-case format and listed in the Use Cases Section.
3. **Use Cases:** Describe all applicable use cases in the system.
4. **Classes and Objects:** Describe all classes by expressing its functions and attributes in the system.
5. **Non-Functional Requirements:** Requirements may exist for performance, reliability, availability, security, maintainability and portability. For example (95% of transaction shall be processed in less than a second, system downtime may not exceed 1 minute per day, > 30 day MTBF value, etc).
6. **Design Constraints:** Specify design constrains imposed by other standards, company policies, hardware limitation, etc. that will impact this software project.
7. **Other Requirements:** Catchall section for any additional requirements.

4. **Analysis Models:** List all analysis models used in developing specific requirements previously given in this SRS. Each model should include an introduction and a narrative description. Furthermore, each model should be traceable the SRS's requirements.
    1. **Sequence Diagrams:** It is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams typically are associated with use case realizations in the Logical View of the system under development.

    2. **Data Flow Diagrams:** It is a graphical representation of the "flow" of data through an information system, modeling its process aspects. Often they are a preliminary step used to create an overview of the system which can later be elaborated.

3. **State-Transition Diagrams:** Describe the system as finite number of states.

5. **Change Management Process:** Identify and describe the process that will be used to update the SRS, as needed, when project scope or requirements change. Who can submit changes and by what means, and how will these changes be approved.
6. **Appendices:** Provide additional (and hopefully helpful) information. If present, the SRS should explicitly state whether the information contained within an appendix is to be considered as a part of the SRS's overall set of requirements. Example Appendices could include (initial) conceptual documents for the software project, marketing materials, minutes of meetings with the customer(s), etc.

**Characteristics of SRS**

Software requirements specification should be accurate, complete, efficient, and of high quality, so that it does not affect the entire project plan. An SRS is said to be of high quality when the developer and user easily understand the prepared document. Other characteristics of SRS are discussed below.

1. **Correct:** SRS is correct when all user requirements are stated in the requirements document. The stated requirements should be according to the desired system. This implies that each requirement is examined to ensure that it (SRS) represents user requirements. Note that there is no specified tool or procedure to assure the correctness of SRS. Correctness ensures that all specified requirements are performed correctly.
2. **Unambiguous:** SRS is unambiguous when every stated requirement has only one interpretation. This implies that each requirement is uniquely interpreted. In case there is a term used with multiple meanings, the requirements document should specify the meanings in the SRS so that it is clear and easy to understand.
3. **Complete:** SRS is complete when the requirements clearly define what the software is required to do. This includes all the requirements related to performance, design and functionality.
4. **Ranked for importance/stability:** All requirements are not equally important, hence each requirement is identified to make differences among other requirements. For this, it is essential to clearly identify each requirement. Stability implies the probability of changes in the requirement in future.

5. **Modifiable:** The requirements of the user can change, hence requirements document should be created in such a manner that those changes can be modified easily, consistently maintaining the structure and style of the SRS.
6. **Traceable:** SRS is traceable when the source of each requirement is clear and facilitates the reference of each requirement in future. For this, forward tracing and backward tracing are used. Forward tracing implies that each requirement should be traceable to design and code elements. Backward tracing implies defining each requirement explicitly referencing its source.
7. **Verifiable:** SRS is verifiable when the specified requirements can be verified with a cost-effective process to check whether the final software meets those requirements. The requirements are verified with the help of reviews. Note that unambiguity is essential for verifiability.
8. **Consistent:** SRS is consistent when the subsets of individual requirements defined do not conflict with each other. For example, there can be a case when different requirements can use different terms to refer to the same object. There can be logical or temporal conflicts between the specified requirements and some requirements whose logical or temporal characteristics are not satisfied. For instance, a requirement states that an event 'a' is to occur before another event 'b'. But then another set of requirements states (directly or indirectly by transitivity) that event 'b' should occur before event 'a'.

**Requirement Engineering Process:**

Goal of requirement engg process is to create and maintain a system requirements documents. Processes used to discover, analyse and validate system requirements

The flowing are the four sub process in RE process:

- Feasibility Studies- concerned with assessing whether the system is useful to the business.
- Requirements elicitation and analysis- discovering requirements.
- Requirements Specification- converting the requirements into a standard form.
- Requirements Validation- Checking the requirements with the customers wants.

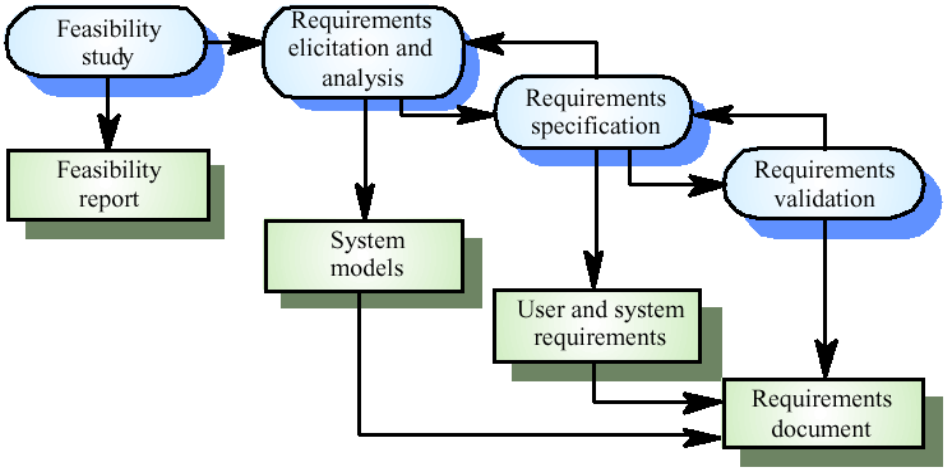
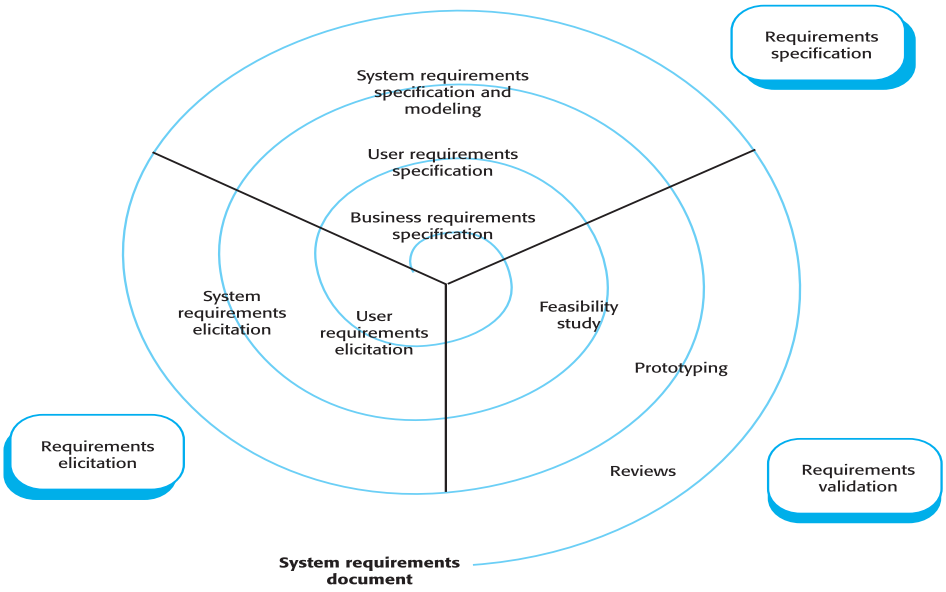



Fig: the requirement engg process

**Spiral model for Requirement Engineering Process:**

The Spiral model accommodates approaches to development in which the requirements are developed to different levels of detail. The no. of iterations around the spiral can vary so the spiral can be exited when the user requirements are elicitated.




**Feasibility studies**

A feasibility study decides whether or not the proposed system is worthwhile or doable. The results of feasibility study should be a report whether or not it is worth carrying on with the requirement engineering and system development process.

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.

**Feasibility study implementation**

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?

**Elicitation and analysis**

- Second stage of requirement engineering.
- In this activity, software engineers work with customers, end users to find about the application domain, services the system should do, performance of system, HW constraints etc.
- Sometimes called requirements elicitation or requirements discovery.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

Ex: Stake holders for a bank ATM are:

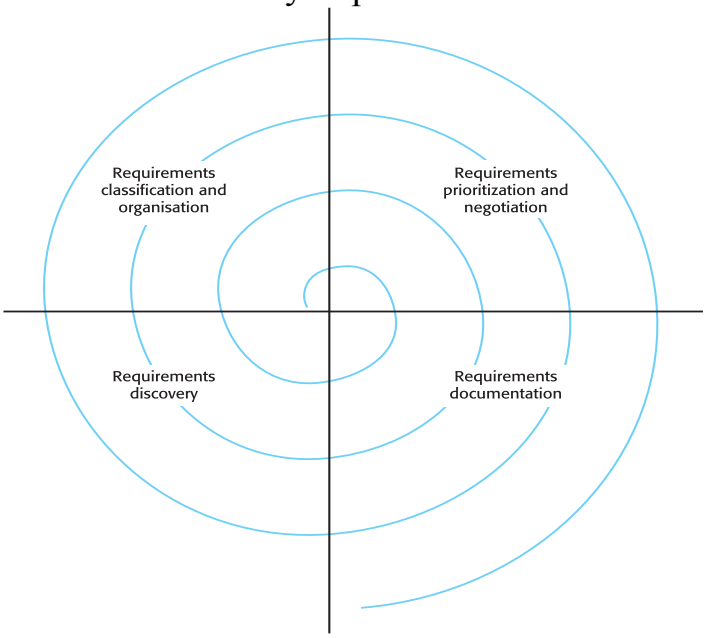
- Bank customers
- Representatives of other banks
- Bank managers
- Counter staff
- Database administrators
- Security managers
- Marketing department
- Hardware and software maintenance engineers
- Banking regulators

**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.

The requirements elicitation and analysis process:




**Process activities are:**

- Requirements discovery
  - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- Requirements classification and organisation
  - Groups related requirements and organises them into coherent clusters.
- Prioritisation and negotiation
  - Prioritising 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 the user and system requirements from this information.
- Sources of information include documentation, system stakeholders and the specifications of similar systems (templates). These requirement sources

cane be represented as system viewpoints, where each viewpoint represent subset of system.

**Viewpoints**

- Viewpoints are a way of structuring the requirements to represent the perspectives of different stakeholders. Stakeholders may be classified under different viewpoints.
- This multi-perspective analysis is important as there is no single correct way to analyse system requirements.
- Viewpoints can be used as a way to classify stakeholders and other sources requirements.

**Types of viewpoint:**

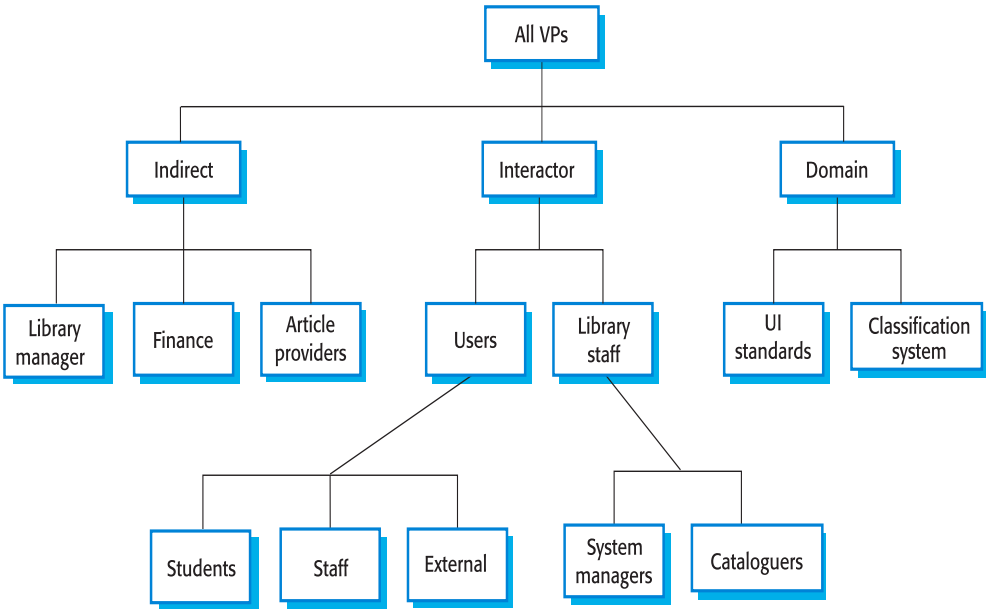
- Interactor viewpoints
    - People or other systems that interact directly with the system. In an ATM, the customers and the account database are interactor VPs.
- Indirect viewpoints
    - Stakeholders who do not use the system themselves but who influence the requirements. In an ATM, management and 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.

**Viewpoint identification**

Identify viewpoints using

- Providers and receivers of system services;
- Systems that interact directly with the system being specified;
- Regulations and standards;
- Sources of business and non-functional requirements.
- Engineers who have to develop and maintain the system;
- Marketing and other business viewpoints.

Example:
LIBSYS viewpoint




**Interviewing**

- In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.
- There are two types of interview
    - 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 are explored with stakeholders.

  In practice, a mix of closed and open-ended interviewing are conducted with stakeholders.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- 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 think that it isn't worth articulating.

Characteristics of Effective interview

- Interviewers should be open-minded, willing to listen to stakeholders and should not have pre-conceived ideas about the requirements.
- They should prompt the interviewee with a question or a proposal and should not simply expect them to respond to a question such as 'what do you want'.

## Scenarios

- People find easy to relate real-life examples than abstract descriptions. Scenarios are real-life examples of how a system can be used. Software engineers can use this information gained from discussion to formulate the actual system requirements.
- They should 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;
    - A description of the state when the scenario finishes.

**Ex: LIBSYS scenario**
**Initial assumption**: The user has logged on to the LIBSYS system and has located the journal containing the copy of the article.

**Normal**: The user selects the article to be copied. He or she is then prompted by the system to either provide subscriber information for the journal or to indicate how they will pay for the article. Alternative payment methods are by credit card or by quoting an organisational account number.

The user is then asked to fill in a copyright form that maintains details of the transaction and they then submit this to the LIBSYS system.

The copyright form is checked and, if OK, the PDF version of the article is downloaded to the LIBSYS working area on the user's computer and the user is informed that it is available. The user is asked to select a printer and a copy of the article is printed. If the article has been flagged as 'print-only' it is deleted from the user's system once the user has confirmed that printing is complete.

**What can go wrong**: The user may fail to fill in the copyright form correctly. In this case, the form should be re-presented to the user for correction. If the resubmitted form is still incorrect then the user's request for the article is rejected.

The payment may be rejected by the system. The user's request for the article is rejected.

The article download may fail. Retry until successful or the user terminates the session.

It may not be possible to print the article. If the article is not flagged as 'print-only' then it is held in the LIBSYS workspace. Otherwise, the article is deleted and the user's account credited with the cost of the article.

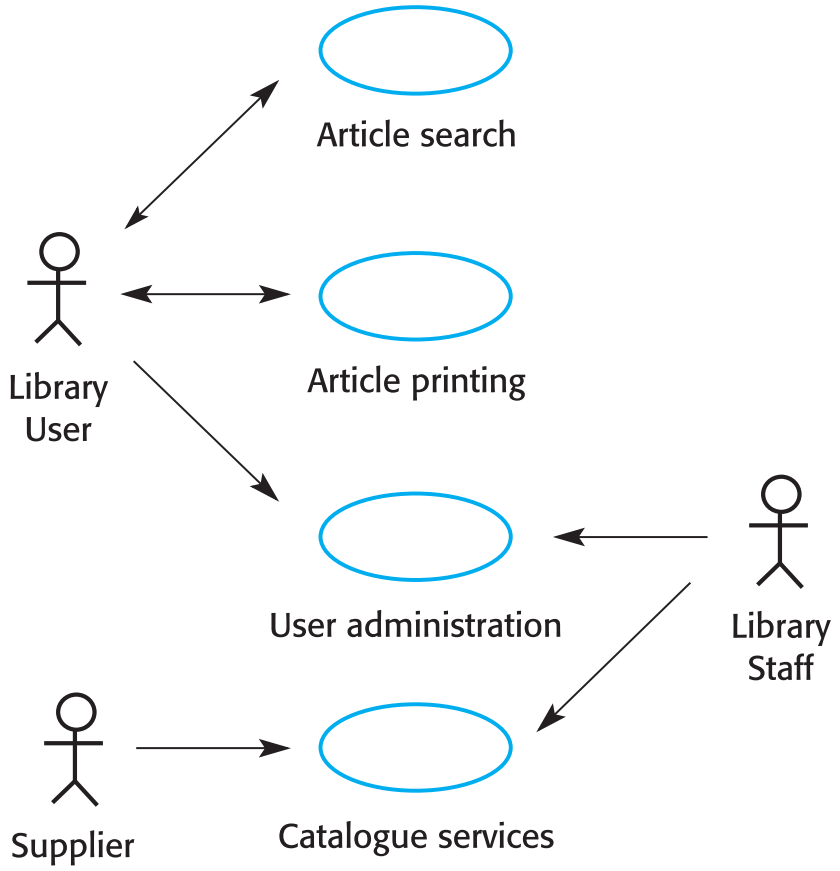
**Other activities**: Simultaneous downloads of other articles.

**System state on completion**: User is logged on. The downloaded article has been deleted from LIBSYS workspace if it has been flagged as print-only.

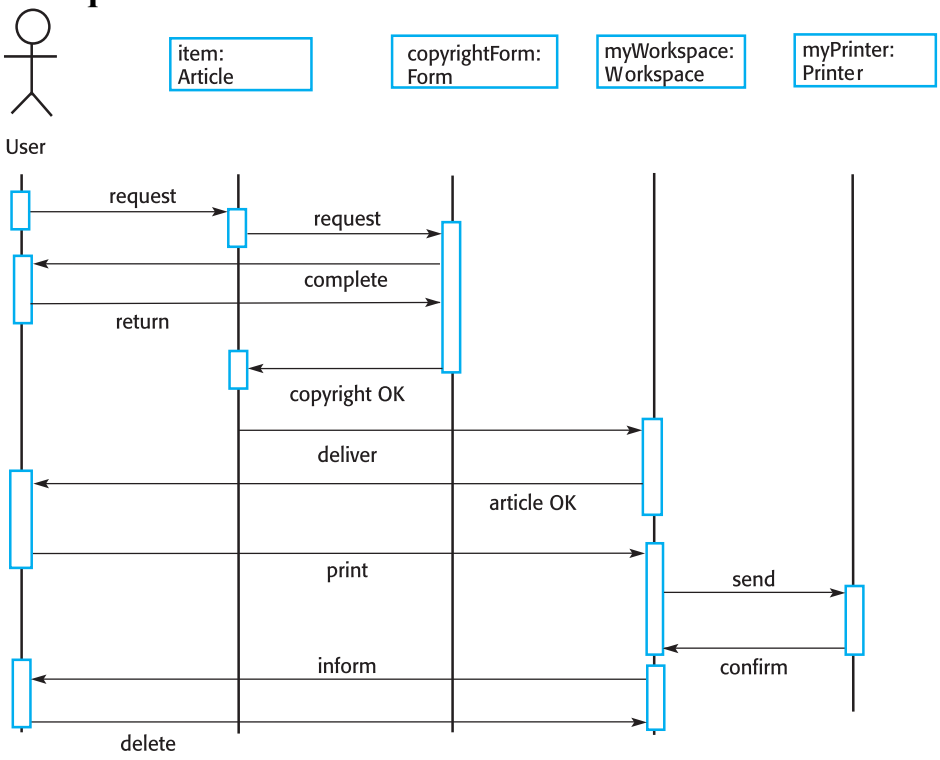
**Use cases**

- 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.
- Actors are represented by sticks and each interaction is represented by ellipse.
- **Sequence diagrams** may be used to add detail to use-cases by showing the sequence of event processing in the system.

Example: LIBSYS use cases



**Print article sequence**




**Ethnography:**

- Software systems are used in a social and organisational context. This can influence or even dominate the system requirements.
- Social and organisational factors of importance may be observed.
- Satisfying these social and organizational requirements is often critical for the success of the system.

Ethnography is an observational technique that can be used to understand the social and organizational requirements. Analyst must work in the environment where the system will be used to document the requirements.

Ethnography is useful for:

- Requirements that are derived from the way that people actually work rather than the way in which process definitions suggest that they ought to work.
- Requirements that are derived from cooperation and awareness of other people's activities.

**Requirements validation**

- Concerned with demonstrating that the requirements define the system that the 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 fixing an implementation error.

During the requirement validation process, checks should be carried out on the requirements in the requirements documents. 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 given available budget and technology
- Verifiability. Can the requirements be checked?

**Requirements validation techniques**

- Requirements reviews
    - Systematic manual analysis of the requirements.
- Prototyping
    - Using an executable model of the system to check requirements. Covered in Chapter 17.
- Test-case generation
    - Developing tests for requirements to check testability.

**Requirements reviews**

- 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.

**Review checks**

- 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?

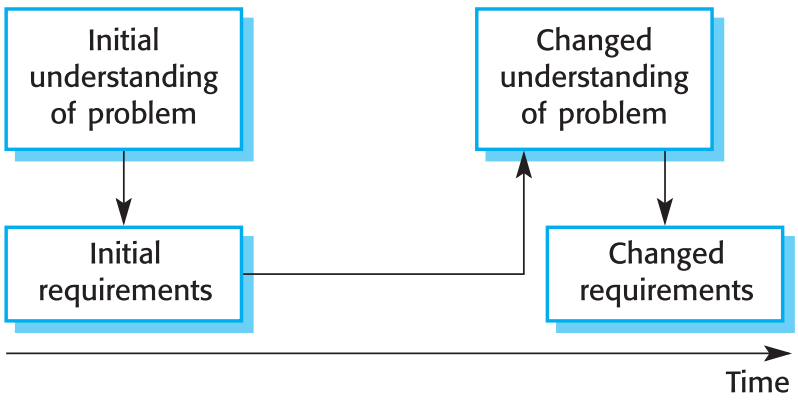
**Requirements management**

The process of managing the changes in system, software and organization environment is called requirement management.

**Requirements are inevitably incomplete and inconsistent**

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

**Requirements evolution**




**From an evolution perspective, requirements fall into two categories:**

- Enduring requirements. Stable requirements derived from the core activity of the customer organisation. E.g. a hospital will always have doctors, nurses, etc. May be derived from domain models
- Volatile requirements. Requirements which change during development or when the system is in use. In a hospital, requirements derived from health-care policy.

**Requirements management planning**

- During the requirements engineering process, you have to plan:
    - Requirements identification
        - How requirements are individually identified;
    - A change management process

        - The process followed when analysing a requirements change;
    - Traceability policies
        - The amount of information about requirements relationships that is maintained;
    - CASE tool support
        - The tool support required to help manage requirements change;

**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 change management**

- Should apply to all proposed changes to the requirements.

Principal stages to change management process:

- 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 to reflect change.

# Software Engineering
## Unit 2- Part1

**Topics**

1. **System Analysis Models**
2. **Context Models**
3. **Data Models**
4. **Flow Oriented Modeling- DFD**
5. **Behavioural Models- Use Cases**

User requirements should be written in natural language because they have to be understood by people who are not technical experts. However, more detailed system requirements may be expressed in a more technical way. One widely used technique is to document the system specification as a set of system models. A system model is an abstraction of the system being studied rather than an alternative representation of that system. These models are graphical representations that describe business processes, the problem to be solved and the system that is to be developed. Because of the graphical representations used, models are often more understandable than detailed natural language descriptions of the system requirements. They are also an important bridge between the analysis and design processes.

- Analysis modeling uses a combination of text and diagrammatic forms to depict requirements for data, function, and behavior in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness and consistency.

**Different Models are used for different perspective: For ex:**

1. External Perspective: where the context or environment of the system is modeled.
2. Behavioural Perspective: where the behaviour of the system is modeled.
3. A structural perspective: where the architecture of the system or the structure of the data processed by the system is modeled

**THE ELEMENTS OF THE ANALYSIS MODEL**

The analysis model must achieve three primary objectives:
(1) to describe what the customer requires,
(2) to establish a basis for the creation of a software design, and
(3) to define a set of requirements that can be validated once the software is built.

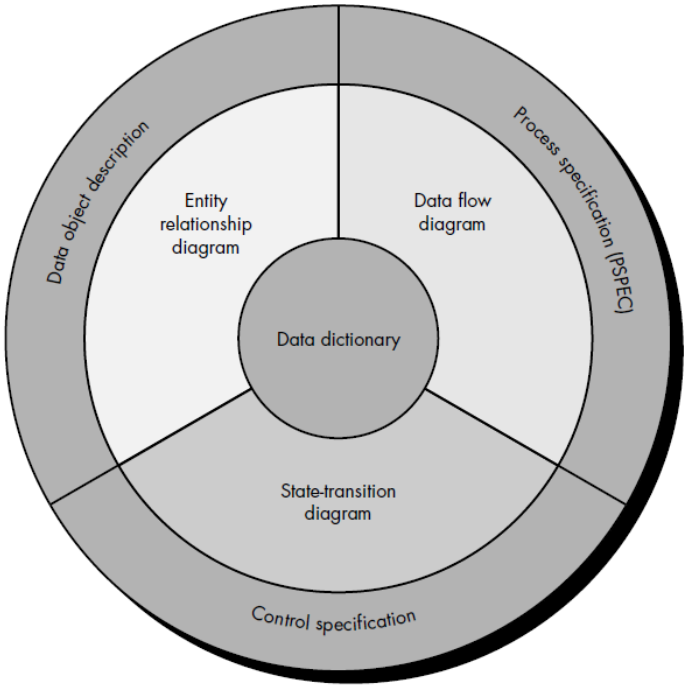



**Fig:** The structure of the analysis model

***data dictionary***—a repository that contains descriptions of all data objects consumed or produced by the software.
***entity relation diagram*** (ERD) depicts relationships between data objects
***data flow diagram*** (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and subfunctions) that transform the data flow
***state transition diagram*** (STD) indicates how the system behaves as a consequence of external events
Additional information about the control aspects of the software is contained in the ***control specification*** (CSPEC).
A **process specification** is a method used to document, analyze and explain the decision-making logic and formulas used to create output data from **process** input data.

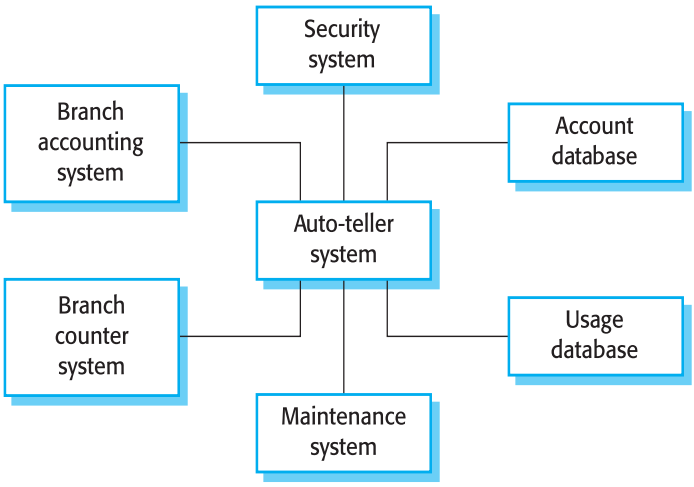
**Types of system models that you might create during the analysis process are:**

- **Data processing model** showing how the data is processed at different stages.
- **Composition model** showing how entities are composed of other entities.
- **Architectural model** showing principal sub-systems.
- **Classification model** showing how entities have common characteristics.
- **Stimulus/response model** showing the system's reaction to events.

**Context models**

Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.

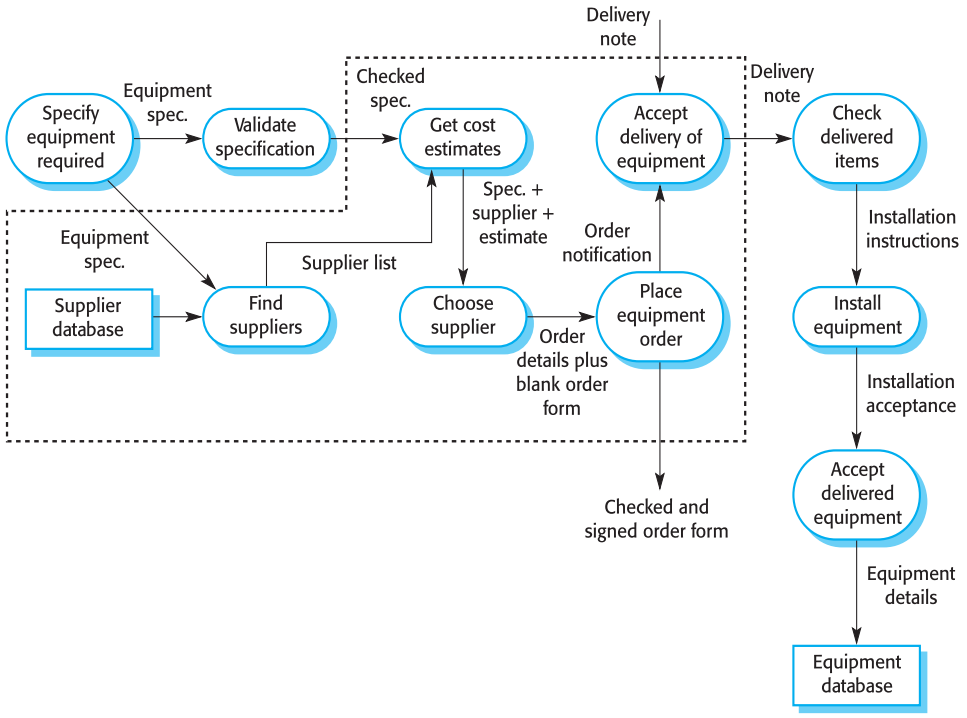
The context of an ATM system




**Architectural models show the system and its relationship with other systems.** However, they do not show the relationships between the other systems in the environment and the system that is being specified. Therefore, simple architectural models are normally supplemented by other models, such as process models, that show the process activities supported by the system.

**Process models**

Process models show the overall process and the processes that are supported by the system. Following Figure illustrates a process model for the process of procuring equipment in an organisation. This involves specifying the equipment required, finding and choosing suppliers, ordering the equipment, taking delivery of the equipment and testing it after delivery. When specifying computer support for this process, you have to decide which of these activities will actually be supported. The other activities are outside the boundary of the system. In Figure 8.2, the dotted line encloses the activities that are within the system boundary.




**Behavioural models**

- Behavioural models are used to describe the overall behaviour of a system.
- Two types of behavioural model are:
  - **Data processing models** that show how data is processed as it moves through the system;
  - **State machine models** that show the systems response to events.
- These models show different perspectives so both of them are required to describe the system's behaviour.

**Data Flow Diagram**

Data flow diagram is a graphical representation of data flow in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data.

**Types of DFD**

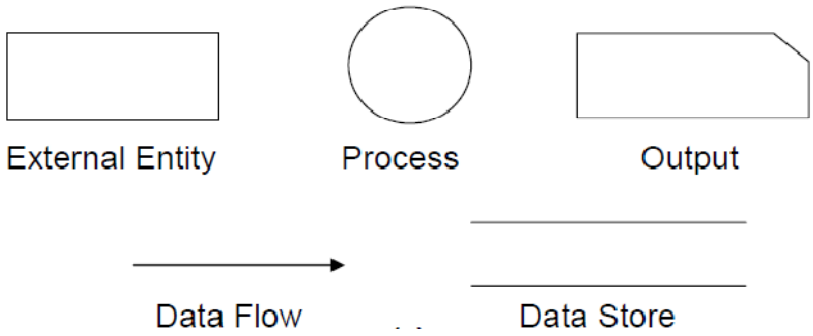
Data Flow Diagrams are either Logical or Physical.

1.**Logical DFD** - This type of DFD concentrates on the system process and flow of data in the system. For example in a Banking software system, how data is moved between different entities.

2. **Physical DFD** - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

**DFD Components**

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




**Entities** - Entities are source and destination of information data. Entities are represented by rectangles with their respective names.

**Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.

**Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.

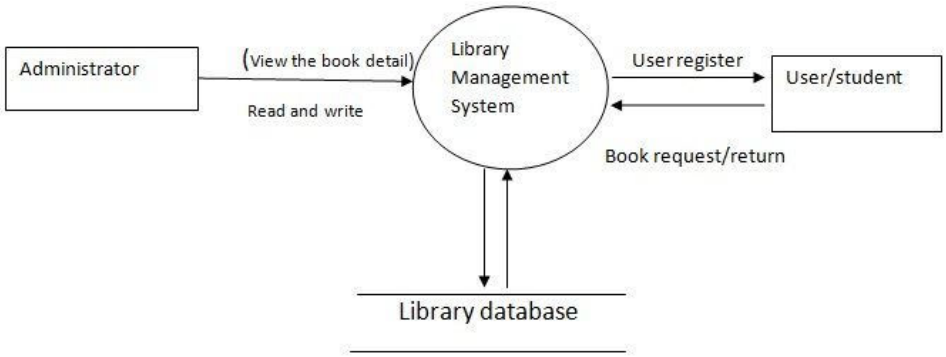
**Data Flow** - Movement of data is shown by pointed arrows..

**Importance of DFDs in a good software design**

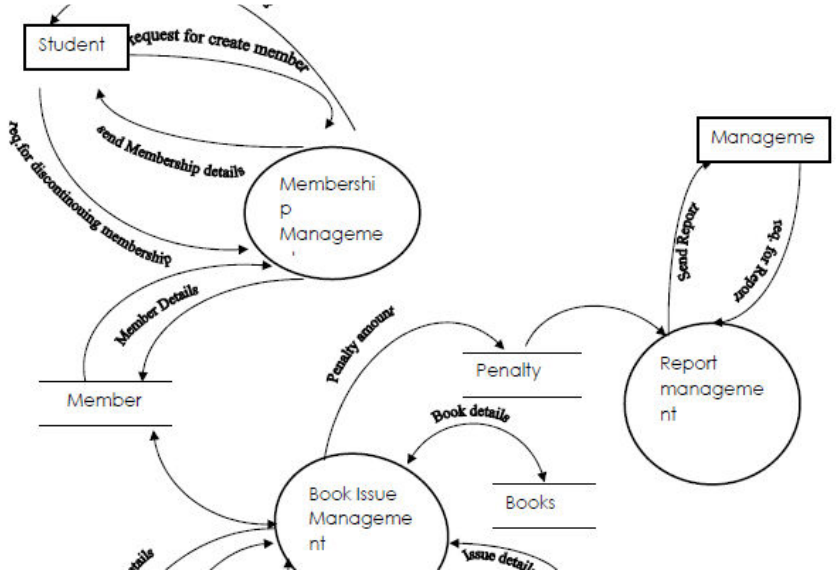
- it is simple to understand and use
- a DFD model hierarchically represents various sub-functions

**Data Flow Diagram Levels**

**Context Diagram.** A context diagram is a top level (also known as "Level 0") data flow diagram. It only contains one process node ("Process 0") that generalizes the function of the entire system in relationship to external entities. The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble. This bubble is labeled according to the main function of the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are also represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows.




The first level DFD shows the main processes within the system. Each of these processes can be broken into further processes




**State machine models**

- These model the behaviour of the system in response to external and internal events.
- The state machine model shows system states and events that cause transitions from one state to another. It does not show the flow of data within the system.
- This type of model is often used for modelling 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.

- A state machine model of a system assumes that, at any time, the system is in one of a number of possible states.
- Statecharts are an integral part of the UML and are used to represent state machine models.

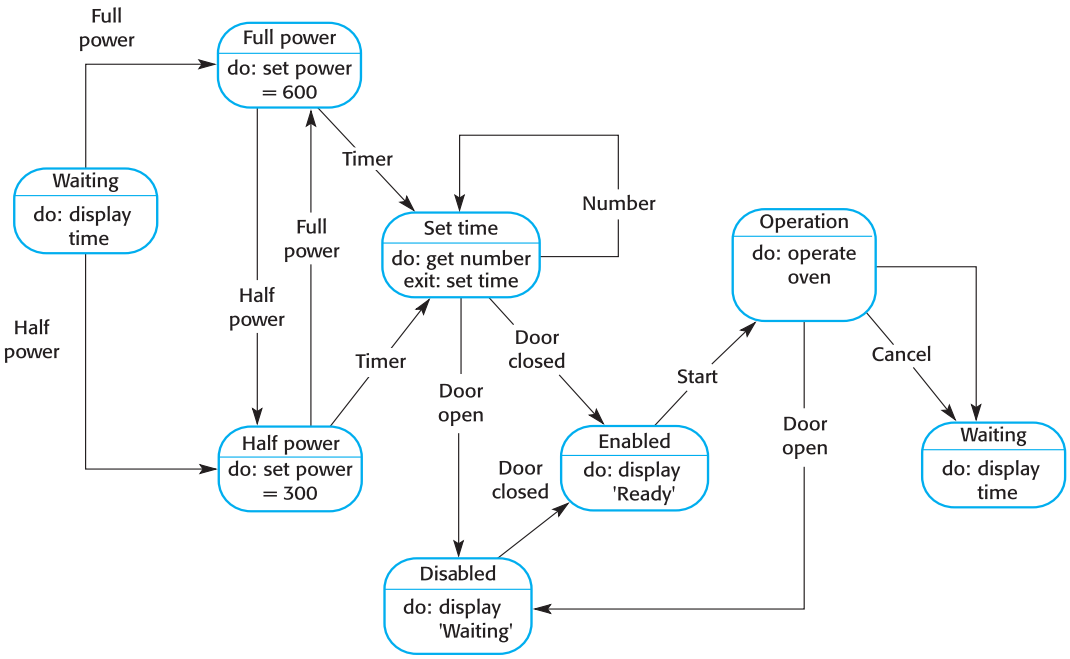



Fig: State machine model of a simple microwaveoven

**Data models**

- Also called as **semantic data model**. Because it defines the logical form of data processed by the system.
- Used to describe the logical structure of data processed by the system
- The most widely used data modelling technique is **Entity-Relation**-Attribute modeling (ERA modelling), which shows the data entities, their associated attributes and the relations between these entities. ER model is used in database design.

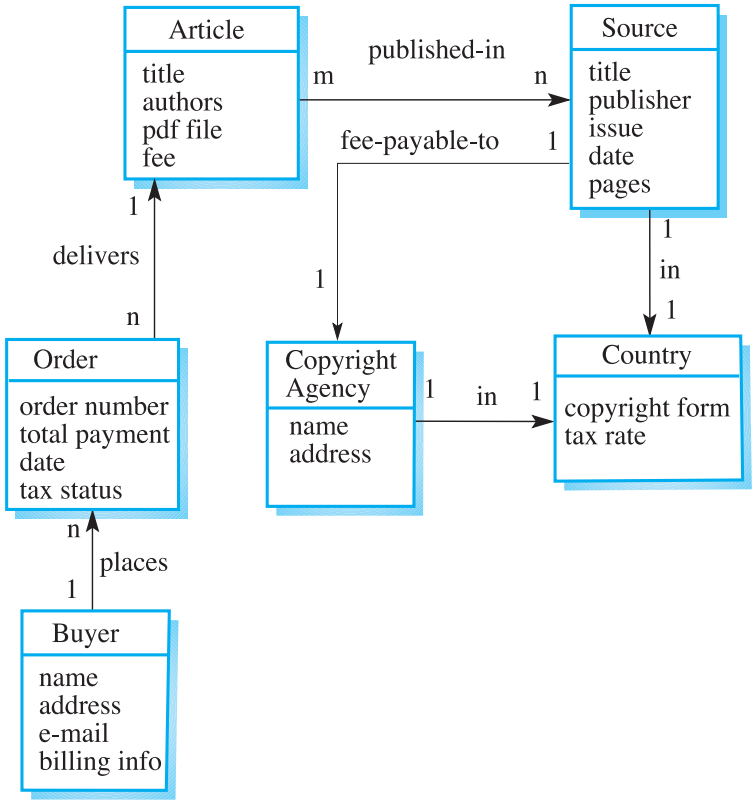



Fig. Semantic data model (ER Model) for the Library Management system (LIBSYS)

**Data dictionaries**

Data dictionaries are lists of all of the names used in the system models. All system names, whether they are names of entities, relations, attributes or services. should be entered in the dictionary.

**Advantages**

- *It is a mechanism for name management.* Many people may have to invent names for entities and relationships when developing a large system model. These names should be used consistently and should not clash. The data dictionary software can check for name uniqueness where necessary and warn requirements analysts of name duplications.
- *It serves as a store of organisational information.* As the system is developed, information that can link analysis, design, implementation and evolution is added to the data dictionary, so that all information about an entity is in one place.

**Example of data dictionary Entries for LIBSys:**

| Name | Description | Type | Date |
|---|---|---|---|
| Article | Details of the published article that may be ordered by people using LIBSYS. | Entity | 30.12.2002 |
| authors | The names of the authors of the article who may be due a share of the fee. | Attribute | 30.12.2002 |
| Buyer | The person or organisation that orders a copy of the article. | Entity | 30.12.2002 |
| fee-payable-to | A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee. | Relation | 29.12.2002 |
| Address (Buyer) | The address of the buyer. This is used to any paper billing information that is required. | Attribute | 31.12.2002 |