# Unit – III

## [Software requirement analysis and specification & Software Architecture]

## 1. Value of good SRS :

The origin of most software systems is in the needs of some clients. The software system itself is created by some developers. Finally, the completed system will be used by the end users. Thus, there are three major parties interested in a new system: the client, the developer, and the users. Somehow the requirements for the system that will satisfy the needs of the clients and the concerns of the users have to be communicated to the developer. The problem is that the client usually does not understand software or the software development process, and the developer often does not understand the client's problem and application area. This causes a communication gap between the parties involved in the development project.

**A basic purpose of the SRS is to bridge this communication gap so they have a shared vision of the software being built. Hence, one of the main advantages of a good SRS is:**

1. An SRS establishes the basis for agreement between the client and the supplier on what the software product will do.

This basis for agreement is frequently formalized into a legal contract between the client (or the customer) and the developer (the supplier). So, through SRS, the client clearly describes what it expects from the supplier, and the developer clearly understands what capabilities to build in the software. A related, but important, advantage is:

2. An SRS provides a reference for validation of the final product.

That is, the SRS helps the client determine if the software meets the requirements. Without a proper SRS, there is no way a client can determine if the software being delivered is what was ordered, and there is no way the developer can convince the client that all the requirements have been fulfilled.

3. A high-quality SRS is a prerequisite to high-quality software.

4. A high-quality SRS reduces the development cost.

# 2. Requirement Process:

The requirement process is the sequence of activities that need to be performed in the requirements phase and that culminate in producing a high-quality document containing the SRS.

The requirements process typically consists of three basic tasks:

1. problem or requirement analysis
2. Requirements specification
3. Requirements validation.

## 1. Problem or Requirement analysis:

- Problem analysis often starts with a high-level "problem statement".

The basic purpose of this activity is to obtain a thorough understanding of what the software needs to provide.

- Frequently, during analysis, the analyst will have a series of meetings with the clients and end users. In the early meetings, the clients and end users will explain to the analyst about their work, their environment, and their needs as they perceive them.

- In the final few meetings, the analyst essentially explains to the client what he understands the system should do and uses the meetings as a means of verifying if what he proposes the system should do is indeed consistent with the objectives of the clients.

## 2. Requirements specification :

- The understanding obtained by problem analysis forms the basis of requirements specification, in which the focus is on clearly specifying the requirements in a document.

- Issues such as representation, specification languages, and tools are addressed during this activity. As analysis produces large amounts of information and knowledge with possible redundancies, properly organizing and describing the requirements is an important goal of this activity.

## 3. Requirements validation :

- Requirements validation focuses on ensuring that what have been specified in the SRS are indeed all the requirements of the software and making sure that the SRS is of good quality. The requirements process terminates with the production of the validated SRS.

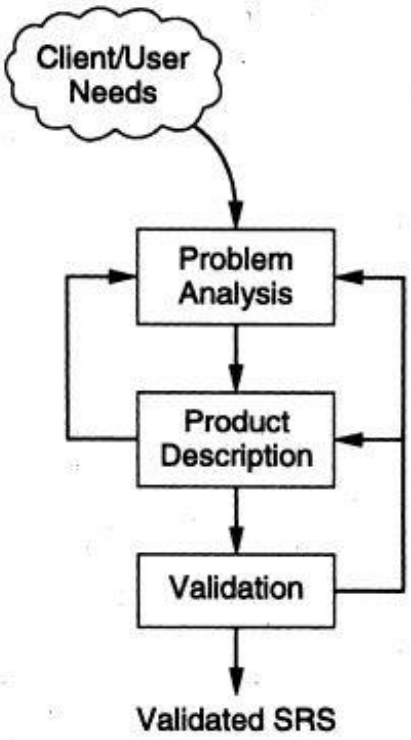



Figure 1: The requirement process

- The overall requirement process is shown in Figure 1. As shown in the figure, from the specification activity we may go back to the analysis activity.

- This happens as frequently some parts of the problem are analyzed and then specified before other parts are analyzed and specified.

- Furthermore, the process of specification frequently shows shortcomings in the knowledge of the problem, thereby necessitating further analysis.

- Once the specification is done, it goes through the validation activity. This activity may reveal problems in the specification itself, which requires going back to the specification step, or may reveal shortcomings in the understanding of the problem, which requires going back to the analysis activity.

## 3.Requirement Specification :

# What is Software Requirement Specification - [SRS]?

A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

Some of the desirable characteristics of an SRS are [53]:

1. Correct
2. Complete
3. Unambiguous
4. Verifiable
5. Consistent
6. Ranked for importance and/or stability

1. An SRS is **correct** if every requirement included in the SRS represents something required in the final system.

2. It is **complete** if everything the software is supposed to do and the responses of the software to all classes of input data are specified in the SRS.

3. It is **unambiguous** if and only if every requirement stated has one and only one interpretation. Requirements are often written in natural language, which is inherently ambiguous. If the requirements are specified in a natural language, the SRS writer has to be especially careful to ensure that there are no ambiguities.

4. An SRS is **verifiable** if and only if every stated requirement is verifiable. A requirement is verifiable if there exists some cost-effective process that can check whether the final software meets that requirement.

5. It is **consistent** if there is no requirement that conflicts with another. Terminology can cause inconsistencies; for example, different requirements may use different terms to refer to the same object. There may be logical or temporal conflict between requirements that causes inconsistencies. This occurs if the SRS contains two or together by any software system. For example, suppose a requirement states that an event *e* is to occur before another event *f*. But then another set of requirements states (directly or indirectly by transitivity) that event *f* should occur before event e. Inconsistencies in an SRS can reflect some major problems.

6.

An SRS is **ranked** for importance and/or stability if for each requirement the importance and the stability of the requirement are indicated. Stability of a requirement reflects the chances of it changing in the future. It can be reflected in terms of the expected change volume. This understanding of value each requirement provides is essential for iterative development—selection of requirements for an iteration is based on this evaluation.

Of all these characteristics, completeness is perhaps the most important and also the most difficult property to establish. One of the most common defects in requirements specification is incompleteness. Missing requirements necessitate additions and modifications to the requirements later in the development cycle, which are often expensive to incorporate. Incompleteness is also a major source of disagreement between the client and the supplier.

.

## 4. COMPONENTS OF AN SRS

Here we describe some of system properties that an SRS should specify.
The basic issues, an SRS must address are:

1. Functional requirements
2. Performance requirements
3. Design constraints
4. External interface requirements

Conceptually, any SRS should have these components. Now we will discuss them one by one.

**1. Functional Requirements**

Functional requirements specify what output should be produced from the given inputs. So they basically describe the connectivity between the input and output of the system. For each functional requirement:

1. A detailed description of all the data inputs and their sources, the units of measure, and the range of valid inputs be specified:

2. All the operations to be performed on the input data obtain the output should be specified, and

3. Care must be taken not to specify any algorithms that are not parts of the system but that may be needed to implement the system.

4. It must clearly state what the system should do if system behaves abnormally when any invalid input is given or due to some error during computation. Specifically, it should specify the behaviour of the system for invalid inputs and invalid outputs.

**2. Performance Requirements (Speed Requirements)**

This part of an SRS specifies the performance constraints on the software system. All the requirements related to the performance characteristics of the system must be clearly specified. Performance requirements are typically expressed as processed transaction s per second or response time from the system for a user event or screen refresh time or a combination of these. It is a good idea to pin down performance requirements for the most used or critical transactions, user events and screens.

**2. Design Constraints**

The client environment may restrict the designer to include some design constraints that must be followed. The various design constraints are standard compliance, resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

**Standard Compliance:** It specifies the requirements for the standard the system must follow. The standards may include the report format and according procedures.

**Hardware Limitations**: The software needs some existing or predetermined hardware to operate, thus imposing restrictions on the design. Hardware limitations can includes the types of machines to be used operating system availability memory space etc.

**Fault Tolerance**: Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex and expensive, so they should be minimized.

**Security:** Currently security requirements have become essential and major for all types of systems. Security requirements place restriction s on the use of certain commands control access to database, provide different kinds of access, requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system.

**4. External Interface Requirements**

For each external interface requirements:

1. All the possible interactions of the software with people hardware and other software should be clearly specified,

2. The characteristics of each user interface of the software product should be specified and

3. The SRS should specify the logical characteristics of each interface between the software product and the hardware components for hardware interfacing.

## 5.Structure of a Requirements Document

The IEEE standards recognize the fact that different projects may require their requirements to be organized differently, that is, there is no one method that is suitable for all projects. It provides different ways of structuring the SRS. The first two sections of the SRS are the same in all of them. The general structure of an SRS is

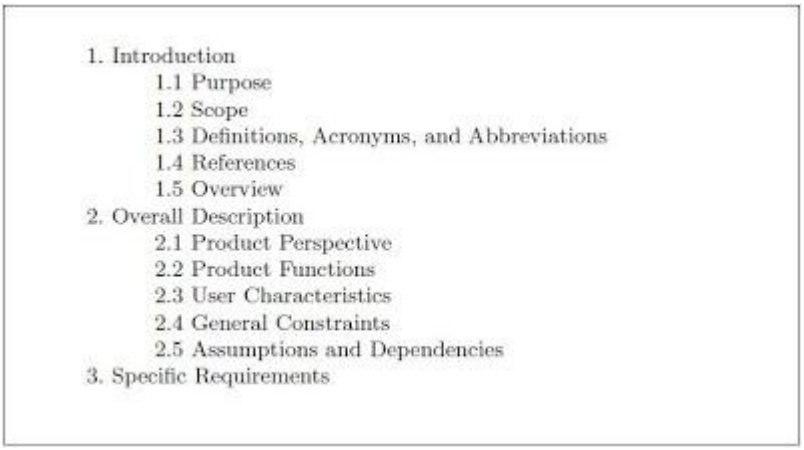



Figure : General structure of an SRS

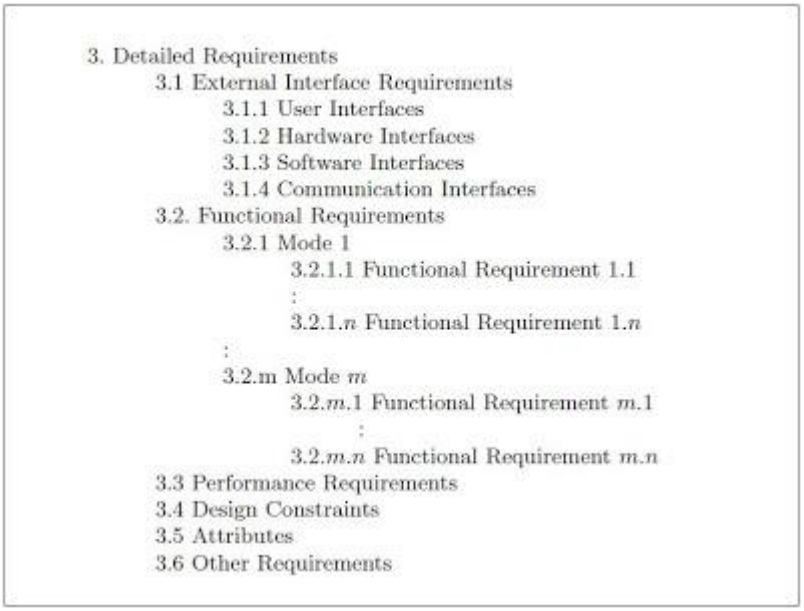



Figure : One organization for specific requirements

- The introduction section contains the purpose, scope, overview, etc., of the requirements document. The key aspect here is to clarify the motivation and business objectives that are driving this project, and the scope of the project.

- The next section gives an overall perspective of the system—how it fits into the larger system, and an overview of all the requirements of this system.

- Product perspective is essentially the relationship of the product to other products; defining if the product is independent or is a part of a larger product, and what the principal interfaces of the product are. A general abstract description of the functions to be performed by the product is given.

- The detailed requirements section describes the details of the requirements that a developer needs to know for designing and developing the system. This is typically the largest and most important part of the document. For this section, different organizations have been suggested in the standard.

- One method to organize the specific requirements is to first specify the external interfaces, followed by functional requirements, performance requirements, design constraints, and system attributes. This structure is shown in Figure.

- In the functional requirements section, the functional capabilities of the system are described.

- The performance section should specify both static and dynamic performance requirements.

## 6. Functional Specifications With Use Cases :

- Functional requirements often form the core of a requirements document. The traditional approach for specifying functionality is to specify each function that the system should provide.

- Use cases specify the functionality of a system by specifying the behavior of the system, captured as interactions of the users with the system.

- Use cases can be used to describe the business processes of the larger business or organization that deploys the software, or it could just describe the behavior of the software system. We will focus on describing the behavior of software systems that are to be built.

Though use cases are primarily for specifying behavior, they can also be used effectively for analysis.

**Use -Case Terms**

Actor :In UML, someone or something outside the system that interacts with the system.

Primary actor : The main actor for whom a use case is initiated and whose goal satisfaction is the main objective of the use case.

Scenario : a set of actions that are performed to achieve a goal under some specified conditions.

Main Success scenario: Describes the interaction if nothing fails and all steps in the scenario succeed.

Extension scenario: Describes the system behaviour if some of the steps in the main scenario do not complete successfully.

# 7. NOTE : PRACTICE USE CASE , DFD'S , ERD DIAGRAMS FROM ASSAIGNMENT .

# 8. Developing Use Cases :

UCs can be evolved in a stepwise refinement manner with each step adding more details. This approach allows UCs to be presented at different levels of abstraction. Though any number of levels of abstraction is possible, four natural levels emerge:

- **Actors and goals.** The actor-goal list enumerates the use cases and specifies the actors for each goal. (The name of the use case is generally the goal.) This table may be extended by giving a brief description of each of the use cases. At this level, the use cases together specify the scope of the system and give an overall view of what it does. Completeness of functionality can be assessed fairly well by reviewing these.

- **Main success scenarios.** For each of the use cases, the main success scenarios are provided at this level. With the main scenarios, the system behavior for each use case is specified. This description can be reviewed to ensure that interests of all the stakeholders are met and that the use case is delivering the desired behavior.

- **Failure conditions**. Once the success scenario is listed, all the possible failure conditions can be identified. At this level, for each step in the main success scenario, the different ways in which a step can fail form the failure conditions. Before deciding what should be done in these failure conditions (which is done at the next level), it is better to enumerate the failure conditions and review for completeness.

- **Failure handling.** This is perhaps the most tricky and difficult part of writing a use case. Often the focus is so much on the main functionality that people do not pay attention to how failures should be handled. Determining what should be the behavior under different failure conditions will often identify new business rules or new actors.

# 9. Four Levels for analysis when employing use case :

- These four levels can also guide the analysis activity. A step-by-step approach for analysis when employing use cases is:

- **Step 1.** Identify the actors and their goals and get an agreement with the concerned stakeholders as to the goals. The actor-goal list will clearly define the scope of the system and will provide an overall view of what the system capabilities are.

- **Step 2**. Understand and specify the main success scenario for each UC, giving more details about the main functions of the system. Interaction and discussion are the primary means to uncover these scenarios though models may be built, if required. During this step, the analyst may uncover that to complete some use case some other use cases are needed, which have not been identified. In this case, the list of use cases will be expanded.

- **Step 3**. When the main success scenario for a use case is agreed upon and the main steps in its execution are specified, then the failure conditions can be examined. Enumerating failure conditions is an excellent method of uncovering special situations that can occur and which must be handled by the system.

- **Step 4**. Finally, specify what should be done for these failure conditions. As details of handling failure scenarios can require a lot of effort and discussion, it is better to first enumerate the different failure conditions and then get the details of these scenarios. Very often, when deciding the failure scenarios, many new business rules of how to deal with these scenarios are uncovered.

# 10. Other Approaches for Problem Analysis :

The basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and the users, what exactly is desired from the software, and what the constraints on the solution are.

1. Divide and conquer
2. State and Projection
3. DFD'S( Data Flow diagrams )
4. ERD'S(Entity Relationship diagram )

- The basic principle used in analysis is the same as in any complex task: **divide and conquer.** That is, partition the problem into sub problems and then try to understand

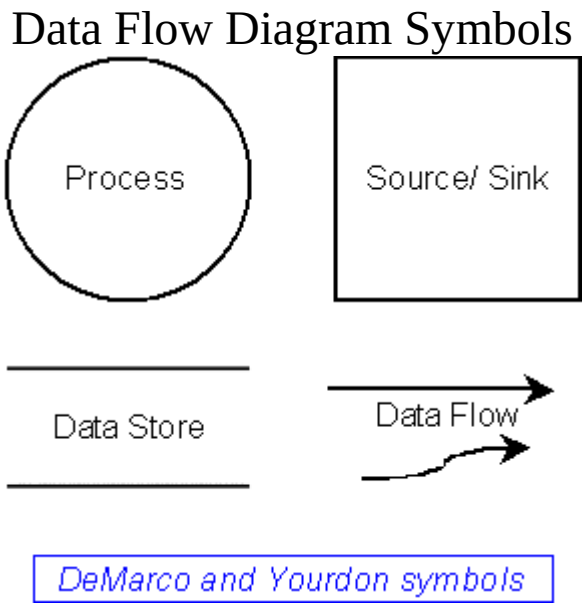
each sub problem and its relationship to other sub problems in an effort to understand the total problem.

- The concepts of **state and projection** can sometimes also be used effectively in the partitioning process. A **state of a system** represents some conditions about the system. Frequently, when using state, a system is first viewed as operating in one of the several possible states, and then a detailed analysis is performed for each state. This approach is sometimes used in real-time software or process-control software.
- **In projection,** a system is defined from multiple points of view . While using projection, different viewpoints of the system are defined and the system is then analyzed from these different perspectives. The different "projections" obtained are combined to form the analysis for the complete system. Analyzing the system from the different perspectives is often easier, as it limits and focuses the scope of the study.

**Data Flow Diagrams :**

# What is a data flow diagram?

A data flow diagram (DFD) maps out the flow of information for any process or system. It uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination. Data flowcharts can range from simple, even hand-drawn process overviews, to in-depth, multi-level DFDs that dig progressively deeper into how the data is handled. They can be used to analyze an existing system or model a new one. Like all the best diagrams and charts, a DFD can often visually "say" things that would be hard to explain in words, and they work for both technical and nontechnical audiences, from developer to CEO. That's why DFDs remain so popular after all these years. While they work well for data flow software and systems, they are less applicable nowadays to visualizing interactive, real-time or database-oriented software or systems.




## DFD RULES :

1. All processes should have unique names. If two data flow lines (or data stores) have the same label, they should both refer to the exact same data flow (or data store).
2. The inputs to a process should differ from the outputs of a process.
3. Any single DFD should not have more than about seven processes.
4. No process can have only outputs. (This would imply that the process is making information from nothing.) If an object has only outputs, then it must be a source.


Incorrect Correct

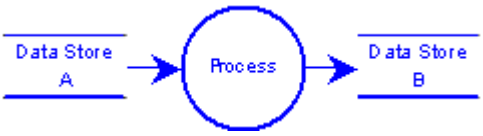
5. No process can have only inputs. (This is referred to as a "black hole".) If an object has only inputs, then it must be a sink.


Incorrect Correct

6. A process has a verb phrase label.

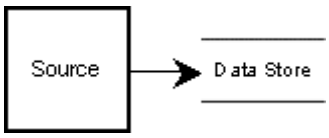
**7** Data cannot move directly from one data store to another data store. Data must be moved by a process.


Incorrect




Correct

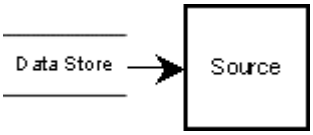
**8** Data cannot move directly from an outside source to a data store. Data must be moved by a process that receives data from the source and places the data in the data store.




Incorrect


Correct

**9** Data cannot move directly to an outside sink from a data store. Data must be moved by a process.

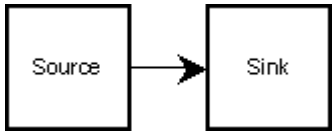



Incorrect


Correct

**10** A data store has a noun phrase label.

**11** Data cannot move directly from a source to a sink. It must be moved by a process if the data are of any concern to the system. If data flows directly from a source to a sink (and does not involved processing) then it is outside the scope of the system and is not shown on the system data flow diagram DFD.

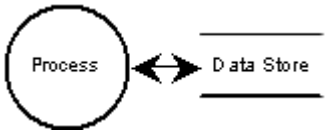



Incorrect

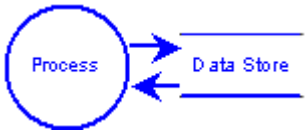

Correct

**12** A source/sink has a noun phrase label.

**13** A data flow has only one direction between symbols. It may flow in both directions between a process and a data store to show a read before an update. To effectively show a read before an update, draw two separate arrows because the two steps (reading and updating) occur at separate times.



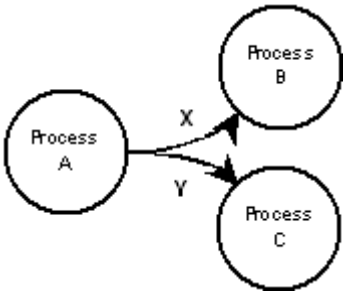

Incorrect



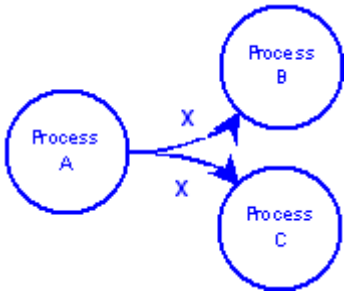

Correct

**14** A fork in a data flow means that exactly the same data goes from a

common location to two or more different processes, data stores, or sources/sinks. (This usually indicates different copies of the same data going to different locations.)

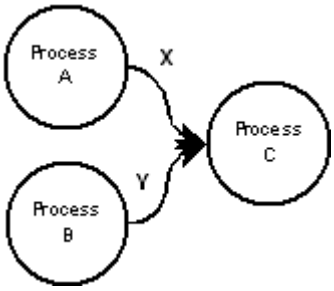



Incorrect

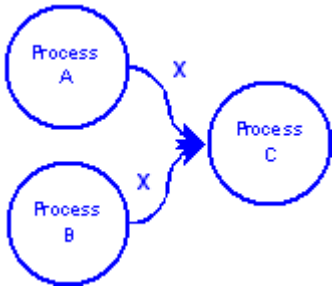



Correct

**15** A join in a data flow means that exactly the same data comes from any of two or more different processes, data stores, or sources/sinks, to a common location.

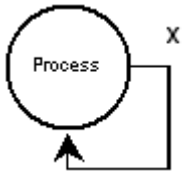



Incorrect

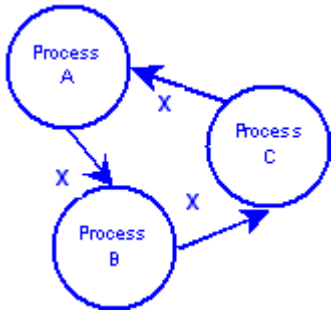



Correct

**16** A data flow cannot go directly back to the same process it leaves. There must be at least one other process that handles the data flow, produces some other data flow, and returns the original data flow to the originating process.




Incorrect




Correct

**17** A data flow to a data store means update (i.e., delete, add, or change).

**18** A data flow from a data store means retrieve or use.

**19** A data flow has a noun phrase label. More than one data flow noun phrase can appear on a single arrow as long as all of the flows on the same arrow move together as one package.

# Entity Relationship Diagrams (ERD'S) :

## What is an Entity Relationship Diagram (ERD)?

- An entity relationship diagram (ERD) shows the relationships of entity sets stored in a database. An entity in this context is a component of data. In other words, ER diagrams illustrate the logical structure of databases.

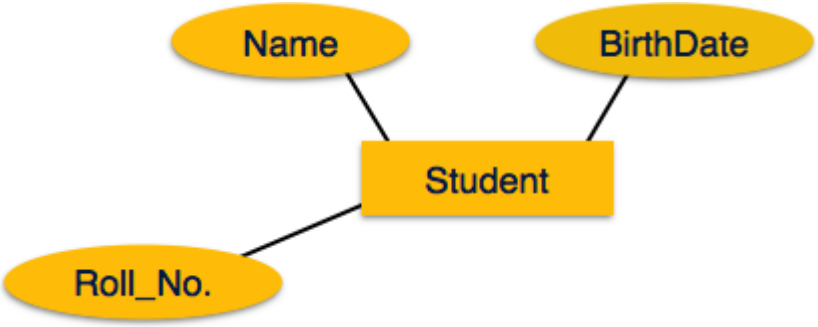
- Any object, for example, entities, attributes of an entity, relationship sets, and attributes of relationship sets, can be represented with the help of an ER diagram.

## Entity

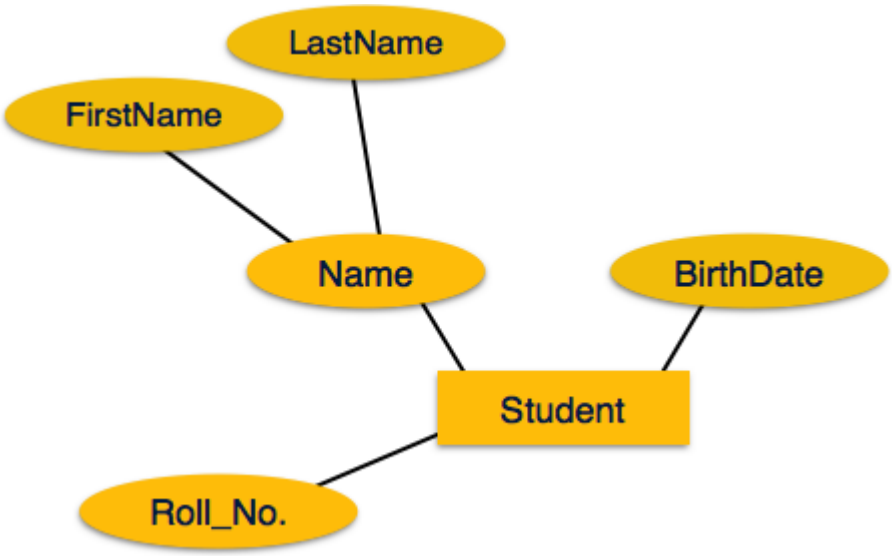
Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.


## Attributes

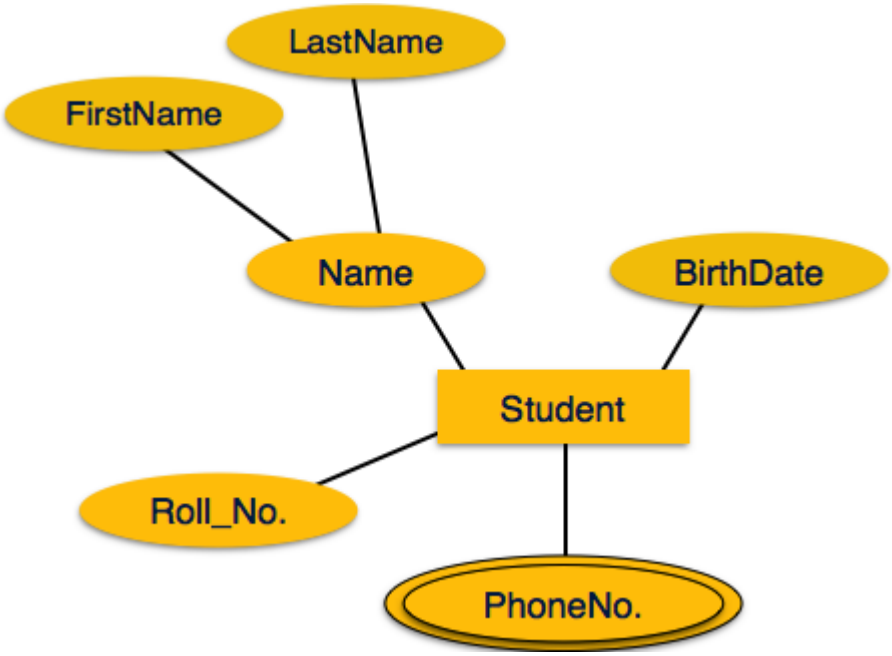
Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity (rectangle).



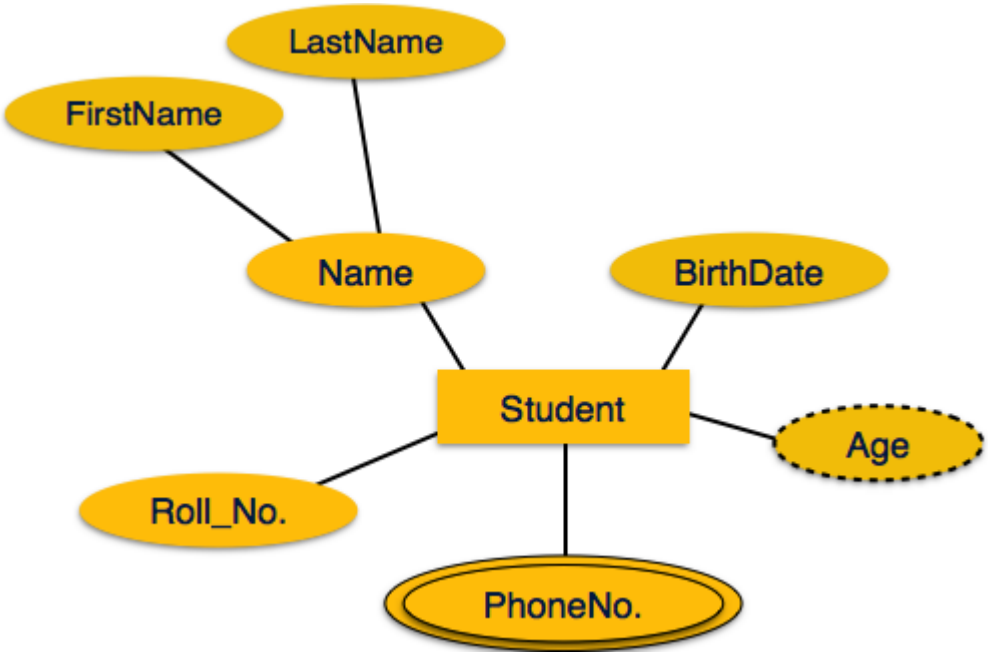

If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is, composite attributes are represented by ellipses that are connected with an ellipse.




**Multivalued** attributes are depicted by double ellipse.



**Derived** attributes are depicted by dashed ellipse.



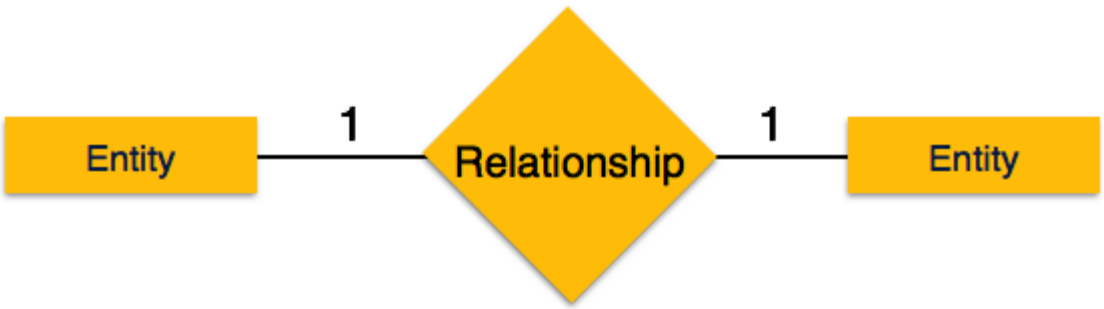

# Relationship

Relationships are represented by diamond-shaped box. Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line.

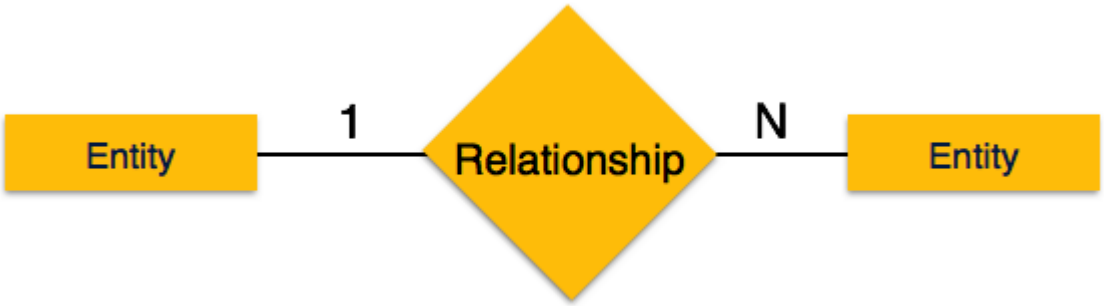
## Binary Relationship and Cardinality

A relationship where two entities are participating is called a **binary relationship**. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

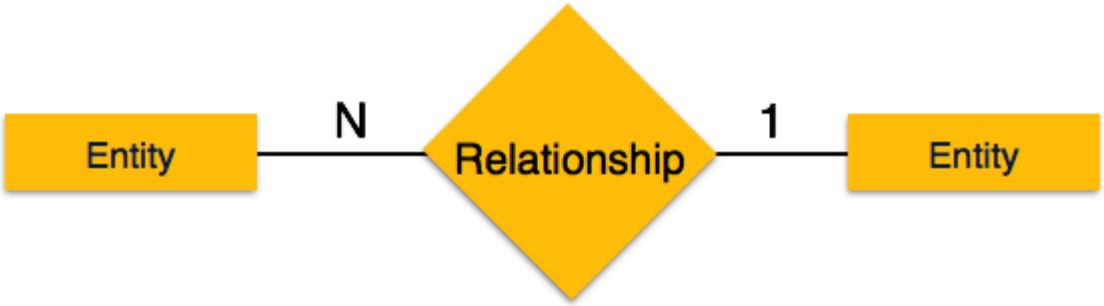
- **One-to-one** – When only one instance of an entity is associated with the relationship, it is marked as '1:1'. The following image reflects that only one instance of each entity should be associated with the relationship. It depicts one-to-one relationship.



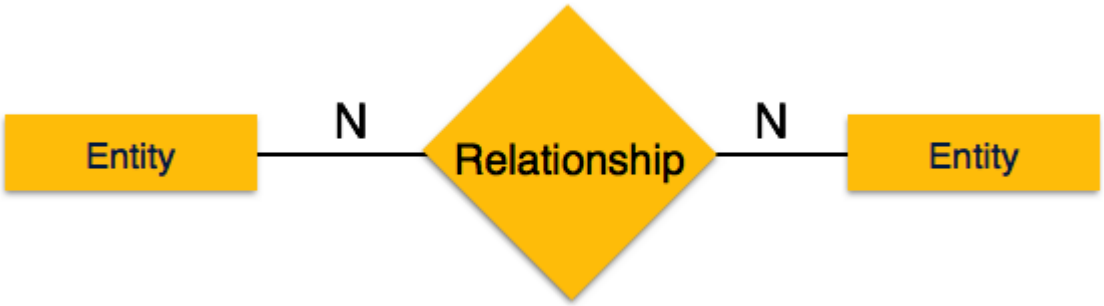
- **One-to-many** – When more than one instance of an entity is associated with a relationship, it is marked as '1:N'. The following image reflects that only one instance of entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts one-to-many relationship.




- **Many-to-one** – When more than one instance of entity is associated with the relationship, it is marked as 'N:1'. The following image reflects that more than one instance of an entity on the left and only one instance of an entity on the right can be associated with the relationship. It depicts many-to-one relationship.




- **Many-to-many** – The following image reflects that more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts many-to-many relationship.



# 11.Validation:

In **software** project management, **software testing**, and **software engineering**, **verification** and**validation** (V&V) is the process of checking that a**software** system meets specifications and that it fulfills its intended purpose. It may also be referred to as **software** quality control.

# 12.Software Architecture:

What is architecture? Generally speaking, architecture of a system provides a very high level view of the parts of the system and how they are related to form the whole system. That is, architecture partitions the system in logical parts such that each part can be comprehended independently, and then describes the system in terms of these parts and the relationship between these parts.

# Role of Software Architecture :

Some of the important uses that software architecture are:

1. Understanding and communication:
    - An architecture description is primarily to communicate the architecture to its various stakeholders, which include the users who will use the system, the clients who commissioned the system, the builders who will build the system, and, of course, the architects.
2. Reuse :
    - The software engineering world has, for a long time, been working toward a discipline where software can be assembled from parts that are developed by different people and are available for others to use. If one wants to build a software product in which existing components may be reused, then architecture becomes the key point at which reuse at the highest level is decided.

3. Construction and Evolution:

   - As architecture partitions the system into parts, some architecture-provided partitioning can naturally be used for constructing the system, which also requires that the system be broken into parts such that di erent teams (or individuals) can separately work on dif erent parts. A suitable partitioning in the architecture can provide the project with the parts that need to be built to build the system. As, almost by definition, the parts specified in an architecture are relatively independent (the dependence between parts coming through their relationship), they can be built independently.

**4.** Analysis :

   - It is highly desirable if some important properties about the behavior of the system can be determined before the system is actually built. This will allow the designers to consider alternatives and select the one that will best suit the needs. Many engineering disciplines use models to analyze design of a product for its cost, reliability, performance, etc. Architecture opens such possibilities for software also.

## 13.Architecture Views:

In software different drawings are called views or structures.

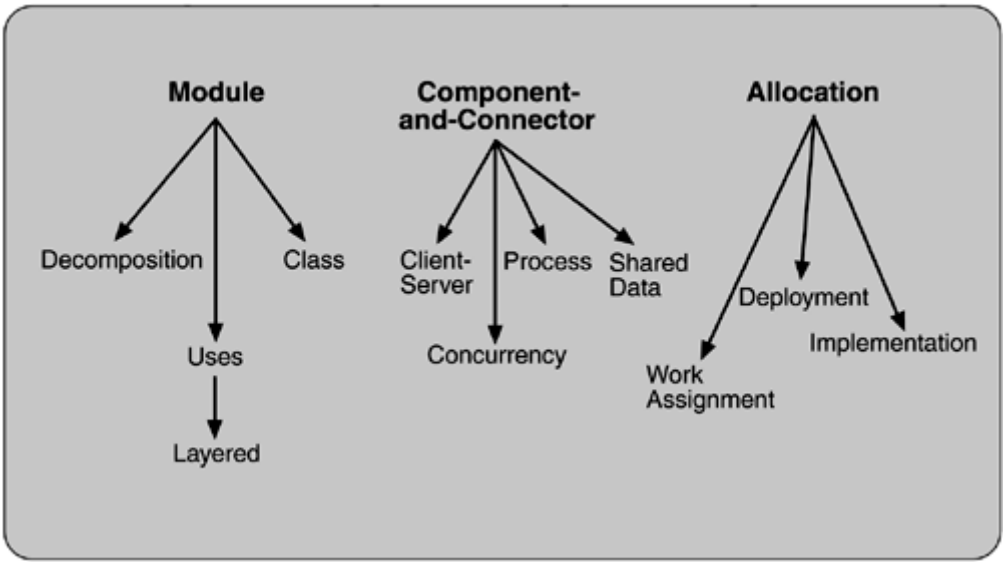
### SOFTWARE STRUCTURES

Some of the most common and useful software structures are :

3 different views / 3 different styles :

1. Module
2. Component and Connector
3. Allocation

*Figure 2-3. Common software architecture structures*




1. In a **module view,** the system is viewed as a collection of code units, each implementing some part of the system functionality. That is, the main elements in this view are modules. These views are code-based and do not explicitly represent any runtime structure of the system. Examples of modules are packages, a class, a procedure, a method, a collection of functions, and a collection of classes.

2. In a **component and connector (C&C) view,** the system is viewed as a collection of runtime entities called components. That is, a component is a unit which has an identity in the executing system. Objects (not classes), a collection of objects, and a process are examples of components. While executing, components need to interact with others to support the system services. Connectors provide means for this interaction. Examples of connectors are pipes and sockets. Shared data can also act as a connector.

3. An **allocation view** focuses on how the different software units are allocated to resources like the hardware, file systems, and people. That is, an allocation view specifies the relationship between software elements and elements of the environments in which the software system is executed. They expose structural properties like which processes run on which processor, and how the system files are organized on a file system.

# 14.Components and Connector View (C&C View):

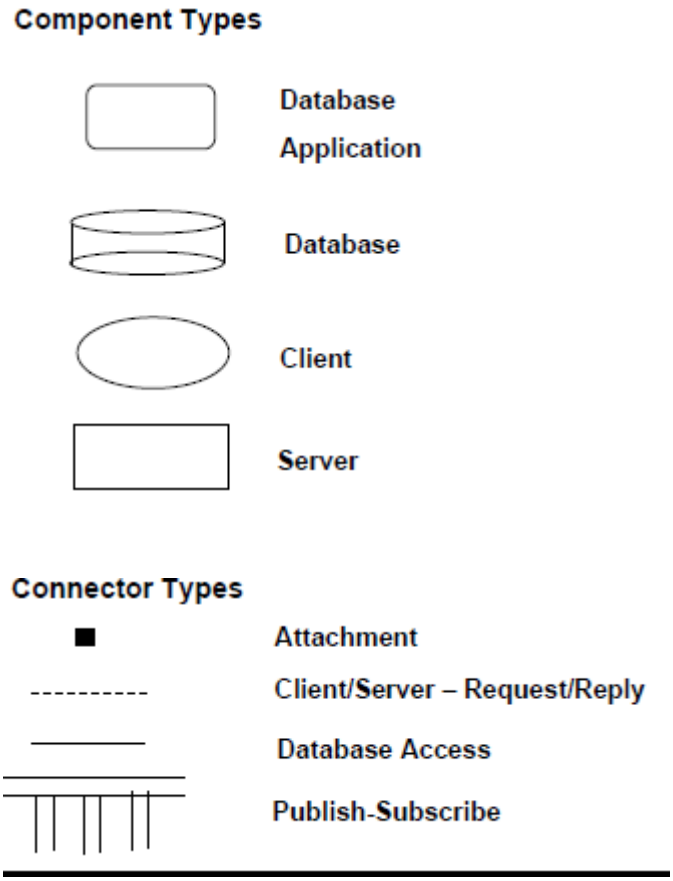
## Component:

Components are generally units of computation or data stores in the system.

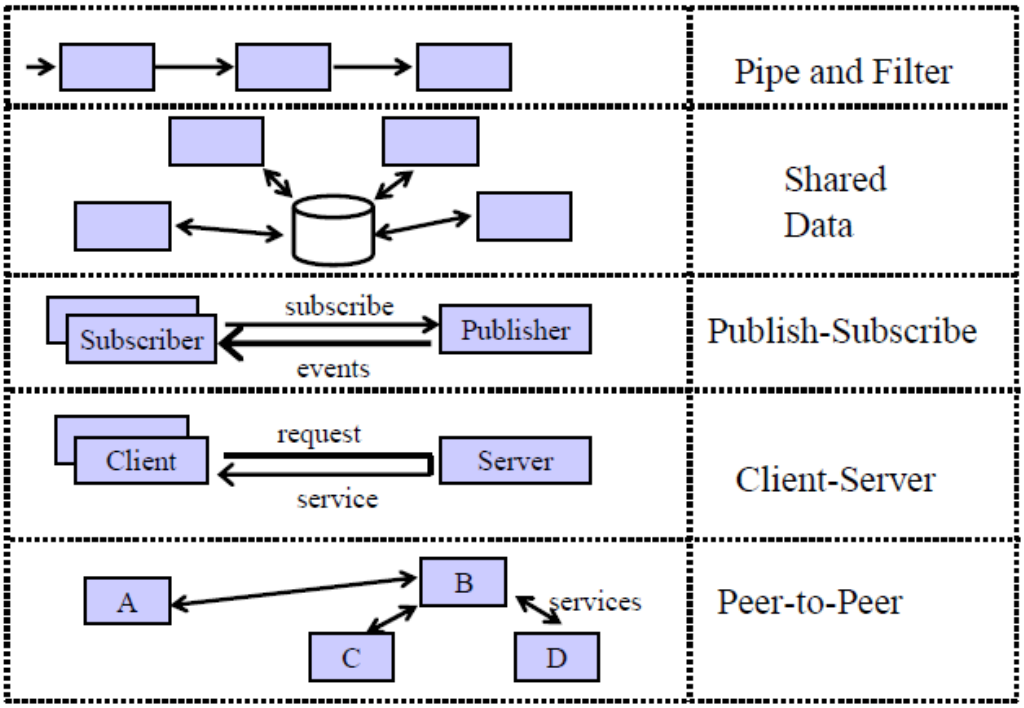
## Connector:

Connectors define the means of interaction between these components.

A **C&C View** describes a runtime structure of the system – what components exist when the system is executing and how they interact during the execution.



# C&C View: Architectural Styles

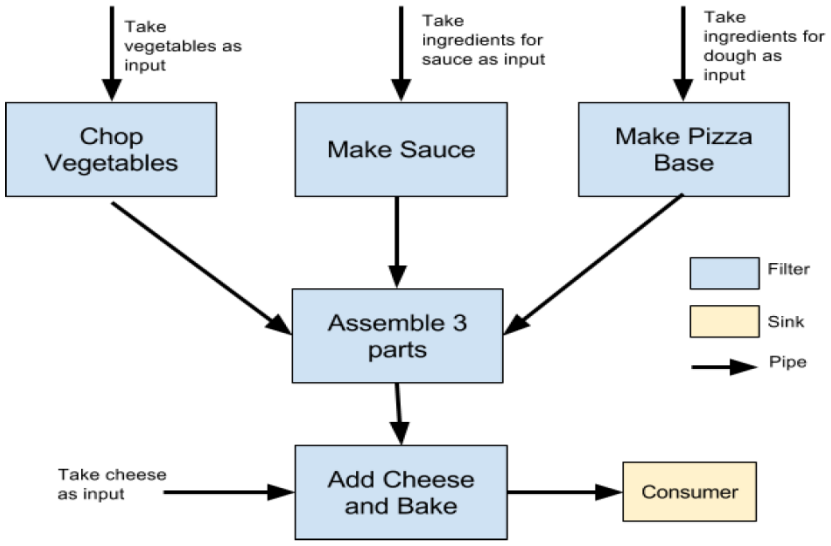



- Pipe-and-Filter
  - Component: Filter
  - Connector: Pipe
  - Data Transformation
- Shared Data
  - Component: Repositories
  - Connector: Data Reader/Writer
  - Multiple Accessors & Persistence
- Publish-Subscribe
  - Component: any
  - Connector: publish-subscribe
  - Send events & msgs. to unknown set of recipients
- Client-Server
  - Component: Client und Server
  - Connector: request/reply
  - Decoupling Apps. from Services
- Peer-to-Peer
  - Component: Peer
  - Connector: bidirectional RPC
  - Collaboration
- Communicating Processes
  - Component: any concurrent unit
  - Connector: data exchg., msg. passing, sync., ctrl.
  - Concurrent Systems

## Pipe and Filter Architectural Style

The Pipe and Filter is an architectural design pattern that allows for stream/asynchronous processing. In this pattern, there are many components, which are referred to as filters, and connectors between the filters that are called pipes. Each filter is responsible for applying a function to the given data; this is known as filtering. Filters can work asynchronously. The final output is given to the consumer, known as a sink.

The diagram below shows a simple representation of the pipe and filter architectural style. In this example we demonstrate how a pizza is made using this style. We are using 5 filters, with 3 of them working asynchronously. We implement the first few filters to process the raw ingredients to create the basic elements for the pizza(the vegetables, the sauce, and the pizza base). When all three of these have been completed, we can assemble them. After it has been assembled, we can then add the cheese to the pizza, and bake it, before delivering it to our customer, the sink.




## Shared Data Architectural Style

Data is shared between components through shared storage. Communication between the computational components and shared data is an unconstrained read-write protocol.

## Types of Components

There are two types of components −

- A **central data** structure or data store or data repository, which is responsible for providing permanent data storage. It represents the current state.

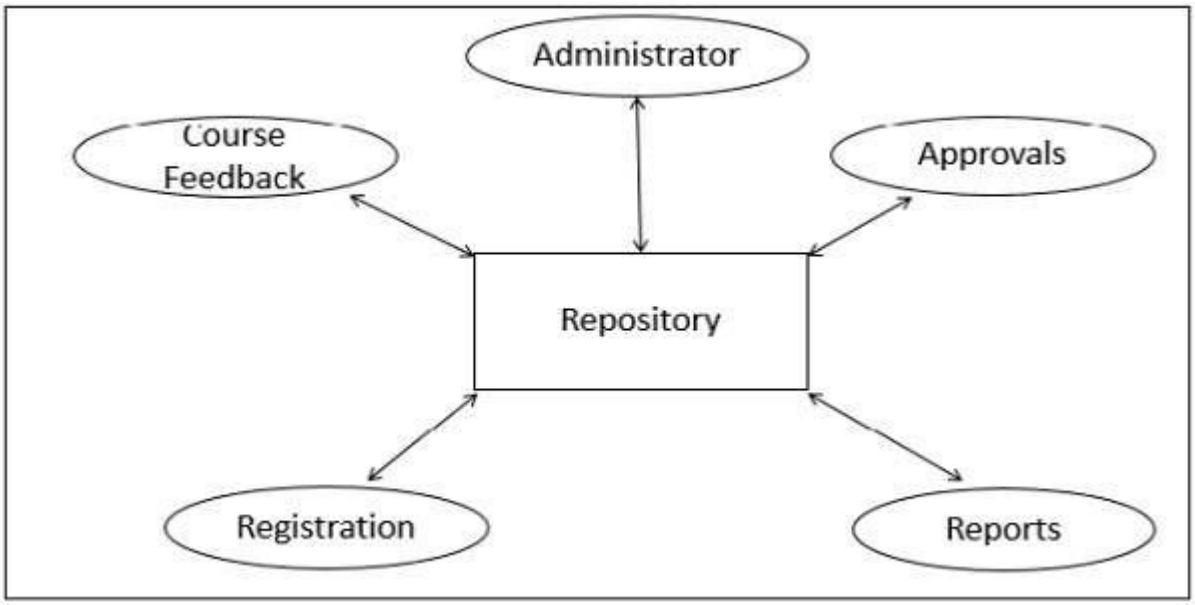
- A **data accessor** or a collection of independent components that operate on the central data store, perform computations, and might put back the results.

Interactions or communication between the data accessors is only through the data store. The data is the only means of communication among clients. The flow of control differentiates the architecture into two categories as **Repository Architecture Style** and **Blackboard Architecture Style**. A brief detail about both the categories is given below –

## Repository Architecture Style

In Repository Architecture Style, the data store is passive and the clients (software components or agents) of the data store are active, which control the logic flow. The participating components check the data-store for changes.

A client sends a request to the system to perform actions (e.g. insert data). The computational processes are independent and triggered by incoming requests. If the types of transactions in an input stream of transactions trigger selection of processes to execute, then it is traditional database or repository architecture, or passive repository. This approach is widely used in DBMS, library information system, the interface repository in CORBA, compilers, and CASE (computer aided software engineering) environments.




## Advantages

Repository Architecture Style has following advantages −

- Provides data integrity, backup and restore features.

- Provides scalability and reusability of agents as they do not have direct communication with each other.
- Reduces overhead of transient data between software components.

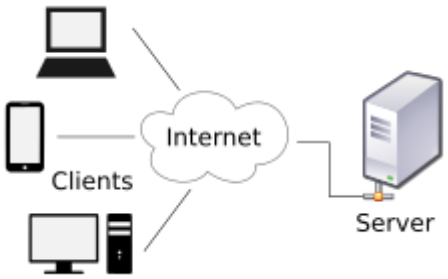
## Disadvantages

Because of being more vulnerable to failure and data replication or duplication, Repository Architecture Style has following disadvantages −

- High dependency between data structure of data store and its agents.
- Changes in data structure highly affect the clients.
- Evolution of data is difficult and expensive.
- Cost of moving data on network for distributed data.

### Client-server Architectural Style

The **client–server model** is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.[1] Often clients and servers communicate over a computer networkon separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests. Examples of computer applications that use the client–server model are Email, network printing, and the World Wide Web.



# 15.Documenting the architecture Design

As I've already mentioned, the design and documentation are not the same. On the other hand, these issues are complementary. Designing the larger part of the application which won't be documented is pointless. Today, I'll think about what is, what for and why do we need good documentation.

## What is the documentation of architecture?

The first association – a diagram. Certainly, to some extent correct, but only partially.

Firstly, we can't describe one diagram as documentation. Architecture, even in a simple project, is a complex and multifaceted notion, therefore it can't be presented on one diagram. Instead, we should create multiple views, presenting architecture from different perspectives and with different accuracy.

Secondly, despite the fact that one picture is worth a thousand words, information such as standards and explanation of taken decisions, is hard to present in a picture. In that case we should use other artefacts e.g., text or code snippets.

## Objectives of documentation

Before we eagerly start preparing documentation of architecture, because that is what best practices say, it is worth considering what the point of doing this is.
We can enumerate a few basic objectives of documentation:

- Communication with the client – it can be the basis for presentation of how quality requirements are going to be fulfilled, and how complex and expensive will be the solution.
- Creating a guide for developers – the main purpose will be to create the source of truth, the reference point for programmers, in case they have any doubts regarding solution they would like to use.
- Team education – good documentation will help to promote information about changes in the architecture, and will be a good foundation for discussions concerning the system development.

## How much documentation and for whom

There is no right or wrong number of artefacts documenting the architecture. Once again, it depends on the team, its dynamics and the environment. It's always worth considering the reasonableness of creating another element of documentation regarding its purpose. Not always more documentation is better. While deciding how much and what kind of artefacts we are going to create, it's better to bear in mind that we won't be able to solve all the communication problems with the use of documentation:

The primary problem with documentation is the difference between context and content. Documentation can provide content, but understanding the context requires domain expertise.

Existing documentation, even if good and valid, must be understood. To achieve this we need good relations, interactions and cognitive empathy among team members as well as between the team and the client.

# Form of documentation

We all know that the value of diagrams presenting relations and behaviours of architectural elements is invaluable. However, diagrams aren't equal, and I don't mean differences between class diagrams and sequences diagrams. Some of them are explicit and understandable, however there are also the ones we look at, and don't understand what the author had in mind.

As we all know the most common standard in the industry is UML. However, for many, not without reason, this standard is too complicated to illustrate the majority of solutions. The problem is so old and common that at the turn of the century Martin Fowler wrote the guide: "UML Distilled: A Brief Guide to the Standard Object Modeling Language". In his book he describes how to pragmatically use UML. How to use its essence to make our diagrams clear and understandable, without wasting time on tunning them to make compliant with standard.

To summarize the above deliberation, diagrams documenting the application architecture can be divided into:

- Formal – compatible with standard (usually UML), which makes them explicit. They are a good choice if we decide to create a complex documentation with the use of CASE tools. It makes sense, especially if we develop a project of the system with a closed specification.
- Informal – "rectangles and arrows" usually drawn on a piece of paper, whiteboard or flip chart to illustrate the idea we are talking about, or to back up our thinking processes. They are highly unclear and without context of people involved in their creation, they may be more confusing than understood.
- Hybrid – a merger of the two mentioned methods, also known as "Arbitrary Modelling Language" i.e. using 20% of standard to create 80% of uniqueness. Diagrams have a good value for money spent on creating them. Definitely, it is the best way of creating documentation to support the above objectives: communication with the client, creating a guide for developers and education.

# The golden thought of software documentation

The above deliberations about documentation may be summed up in two sentences: Documentation has to have its purpose, otherwise it is pointless. What is more, the amount of documents should be barely sufficient, but never insufficient.

# 16.Evaluating Architectures :

## Evaluating an architecture

**Why evaluate an architecture?**

- The earlier you find a problem in a SW project, the better off you are (the cost to fix an error in early design phase is much smaller than the cost to fix the same error in implementation/testing)
- Architecture is the earliest point in the project where trade-offs are visible
- Architecture determines the structure of the project: schedules, budgets, performance indicators, team structure, testing and maintenance activities
- Risk management

## Evaluating an architecture

**SAAM (Software Architecture Analysis Method)**

- Based on scenarios
  - A scenario represents a description of a stakeholder's interaction with the system
- Scenarios are created depending on the point of view of each stakeholder:
  - **Developer** – interested in reusability, implementation, maintenance
  - **Project Manager** – interested in time, cost, quality, extensibility
  - **Tester** – interested in usability, mapping to requirements

## Evaluating an architecture

**Examples**

- Direct scenarios
  - Confronting the architecture with regular use cases
  - Use logic that is provided by the interfaces
  - Stress testing – behavior of components in case of intensive usage
  - Corruption of data/components after long-term usage
  - Data integrity when sending it through communication channels
  - Scenarios regarding functionality found in the requirements
  - Ease of test – how easy is it to test a requirement

## Evaluating an architecture

**Steps of a SAAM evaluation**

Identify and assemble stakeholders
↓
Develop and prioritize scenarios
↓
Describe architecture (actual review)
↓
Classify scenarios as direct or indirect
↓
Perform scenario evaluation
↓
Reveal scenario interactions
↓
Generate overall evaluation

- In this way Architecture is Evaluated. Architecture is evaluated to see that it satisfies the requirements. A common approach is to do a subjective evaluation with respect to the desired properties.

# UNIT 3
# [QUESTIONS]

1.Explain about Value of good SRS .

2. What is Requirement Process ?

3.a)Explain Requirement Specification

b)What are desirable characteristics of an SRS ?

4. WHAT ARE COMPONENTS OF AN SRS

5. Explain Structure of a Requirements Document.

6. Discuss Functional Specifications With Use Cases

7. USE CASE , DFD'S , ERD DIAGRAMS FROM ASSAIGNMENT .

8. How do you Develop Use Cases ?

9.What are Four Levels for analysis when employing use case ?

10.What are Other Approaches for Problem Analysis ?

11.What is DFD?Explain with example.

12.Explain about ERD diagrams.

13.What is Validation.

14.what is software Architecture? Explain role of software architecture.

15.What is an architecture View.What are different architecture views.

[or]

What are different architecture structures.

16.Explain about C&C View.

17.What are different architectural styles.

18.Why and how do you document architecture design.

19.How do you evaluate architecture.

## Repeated & previous papers Questions

1. a) What are the characteristics of a good SRS? 8 M
   b) Describe the architecture styles for C and C view.

2. a) Describe about requirement process and requirement specification of SRS. 8 M
   b) Describe about components and connector view.

3. Explain the ways and means for collecting the software requirements and how are they organized and represented? 16

4.Explain about Shared Data Architecture Style.