Unit 2 Notes

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language.

In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.

Software Design Principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

1. Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

2. Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

Here, there are two common abstraction mechanisms

1. Functional Abstraction
2. Data Abstraction

3. **Modularity**

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

Modular Design

Modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of various parts of a system. We discuss a different section of modular design in detail in this section:

1. Functional Independence

2. Information hiding

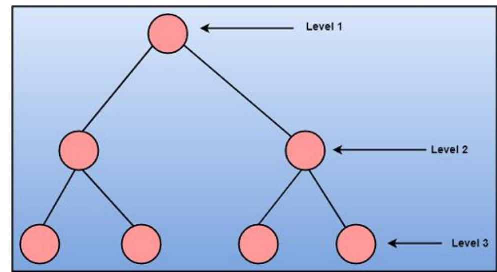
4. Strategy of Design

A good system design strategy is to organize the program modules in such a method that are easy to develop and latter too, change. Structured design methods help developers to deal with the size and complexity of programs. Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program.

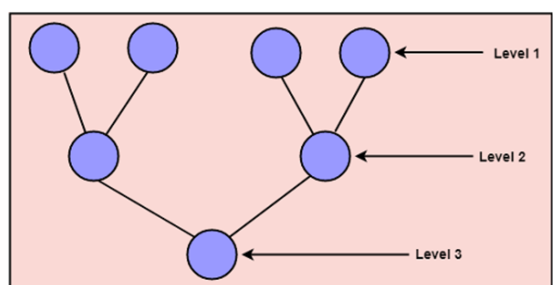
To design a system, there are two possible approaches:

1. Top-down Approach
2. Bottom-up Approach

1. Top-down Approach: This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components.

3. 

2. Bottom-up Approach: A bottom-up approach begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in case of an existing system.

4. 

High-level design

High-level design or **HLD** refers to the overall system, a design that consists description of the system architecture and design and is a generic system design that includes:

1. System architecture
2. Database design
3. Brief description of systems, services, platforms, and relationships among modules.

Low-level design

LLD, or Low-Level Design, is a phase in the software development process where detailed system components and their interactions are specified.

- It describes detailed description of each and every module means it includes actual logic for every system component and it goes deep into each modules specification.
- It is also known as micro level/detailed design.
- It is created by designers and developers.

Modularization: Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering.
Some of these are given below:

- Easy to understand the system.
- System maintenance is easy.
- A module can be used many times as their requirements. No need to write it again and again.

Structure Chart

A structure chart (SC) in software engineering and organizational theory is a chart which shows the breakdown of a system to its lowest manageable levels. A classic "organization chart" for a company is an example of a structure chart. The top of the chart is a box representing the entire problem, the bottom of the chart shows a number of boxes representing the less complicated subproblems. (Left-right on the chart is irrelevant.) A structure chart is NOT a flowchart

.

Module coupling and cohesion

Coupling describes the relationships between modules, and **cohesion** describes the relationships within them. A reduction in interconnectedness between modules (or classes) is therefore achieved via a reduction in coupling.

Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent and changes in one module have little impact on other modules.

1. No Direct Coupling: There is no direct coupling between M1 and M2.

In this case, modules are subordinates to different modules. Therefore, no direct coupling.

2. Data Coupling: When data of one module is passed to another module, this is called data coupling.

3. Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

4. Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

5. External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

6. Common Coupling: Two modules are common coupled if they share information through some global data items.

7. Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module.

1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.

4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.
5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

Pseudo code: It's simply an implementation of an algorithm in the form of annotations and informative text written in plain English. It has no syntax like any of the programming language and thus can't be compiled or interpreted by the computer.

Example:

```
if "1"
   print response
      "I am case 1"
if "2"
   print response
      "I am case 2"
```

What is a Flowchart?

Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.
The process of drawing a flowchart for an algorithm is known as "flowcharting".

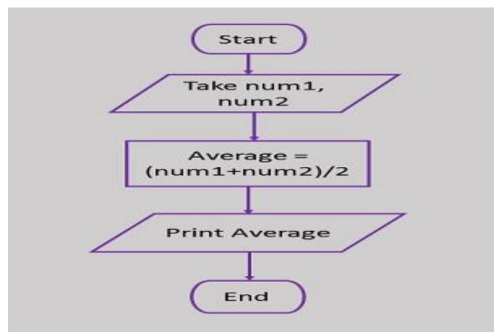
Flowchart Symbols

Here is a chart for some of the common symbols used in drawing flowcharts.

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.

	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

Here is a flowchart to calculate the average of two numbers.



Function Oriented Design

Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.

Object-Oriented Design

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data. For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data. The tasks defined for one purpose cannot refer or change data of other objects.

Objects have their internal data which represent their state. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.

Data Flow Diagram

DFD is the abbreviation for **Data Flow Diagram**. The flow of data of a system or a process is represented by DFD. It also gives insight into the inputs and outputs of each entity and the process itself. DFD does not have control flow and no loops or decision rules are present.

Rules for creating DFD

- The name of the entity should be easy and understandable without any extra assistance(like comments).
- The processes should be numbered or put in ordered list to be referred easily.
- The DFD should maintain consistency across all the DFD levels.
- A single DFD can have a maximum of nine processes and a minimum of three processes.

Symbols Used in DFD

- **Square Box:** A square box defines source or destination of the system. It is also called entity. It is represented by rectangle.
- **Arrow or Line:** An arrow identifies the data flow i.e. it gives information to the data that is in motion.
- **Circle or bubble chart:** It represents as a process that gives us information. It is also called processing box.
- **Open Rectangle:** An open rectangle is a data store. In this data is store either temporary or permanently.

Levels of DFD

DFD uses hierarchy to maintain transparency thus multilevel DFD's can be created. Levels of DFD are as follows:

- 0-level DFD: It represents the entire system as a single bubble and provides an overall picture of the system.
- 1-level DFD: It represents the main functions of the system and how they interact with each other.
- 2-level DFD: It represents the processes within each function of the system and how they interact with each other.
- 3-level DFD: It represents the data flow within each process and how the data is transformed and stored.

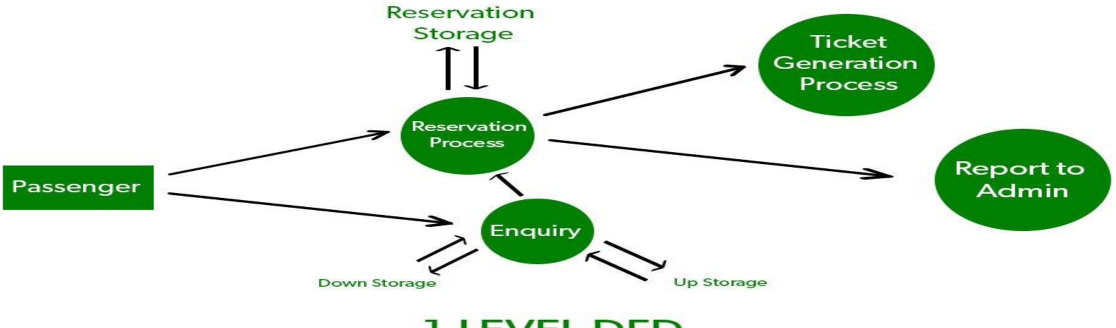
- **0-level DFD:** It is also known as a context diagram. It's designed to be an abstraction view, showing the system as a single process with its relationship to external entities. It represents the entire system as a single bubble with input and output data indicated by

O-LEVEL DFD

incoming/outgoing arrows.

-

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



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

Differences Between DFD and ERD

DFD	ERD
It stands for Data Flow Diagram.	It stands for Entity Relationship Diagram or Model.
Main objective is to represent the processes and data flow between them.	Main objective is to represent the data object or entity and relationship between them.
It explains the flow and process of data input, data output, and storing data.	It explains and represent the relationship between entities stored in a database.
Symbols used in DFD are: rectangles (represent the data entity), circles (represent the process), arrows (represent the flow of data), ovals or parallel lines (represent data storing).	Symbols used in ERD are: rectangles (represent the entity), diamond boxes (represent relationship), lines and standard notations (represent cardinality).

DFD	ERD
Rule followed by DFD is that at least one data flow should be there entering into and leaving the process or store.	Rule followed by ERD is that all entities must represent the set of similar things.
It models the flow of data through a system.	It model entities like people, objects, places and events for which data is stored in a system.

Data dictionary

A data dictionary in Software Engineering means a file or a set of files that ijncludes a database's metadata (hold records about other objects in the database), like data ownership, relationships of the data to another object, and some other data**.**

Entity-Relationship Diagrams

ER-modeling is a data modeling method used in software engineering to produce a conceptual data model of an information system. Diagrams created using this ER-modeling method are called Entity-Relationship Diagrams or ER diagrams or ERDs.

<u>Components of an ER Diagrams</u>

1. Entity

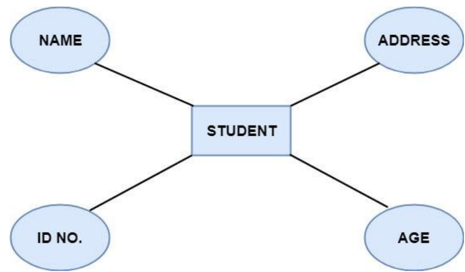
An entity can be a real-world object, either animate or inanimate, that can be merely identifiable. An entity is denoted as a rectangle in an ER diagram. For example, in a school database, students, teachers, classes, and courses offered can be treated as entities. All these entities have some attributes or properties that give them their identity.

Entity Set

An entity set is a collection of related types of entities. An entity set may include entities with attribute sharing similar values. For example, a Student set may contain all the students of a school; likewise, a Teacher set may include all the teachers of a school from all faculties. Entity set need not be disjoint.

2. Attributes

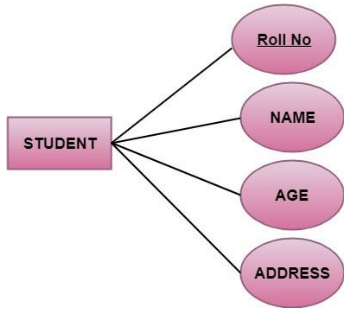
Entities are denoted utilizing their properties, known as attributes. All attributes have values. For example, a student entity may have name, class, and age as attributes.



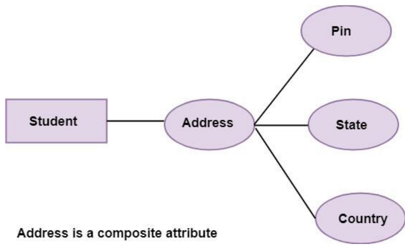
There are four types of Attributes:

1. Key attribute
2. Composite attribute
3. Single-valued attribute
4. Multi-valued attribute
5. Derived attribute

1. Key attribute: Key is an attribute or collection of attributes that uniquely identifies an entity among the entity set. For example, the roll_number of a student makes him identifiable among students.

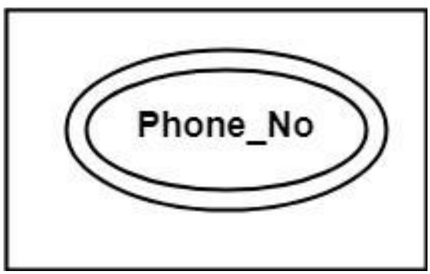


2. Composite attribute: An attribute that is a combination of other attributes is called a composite attribute. For example, In student entity, the student address is a composite attribute as an address is composed of other characteristics such as pin code, state, country.

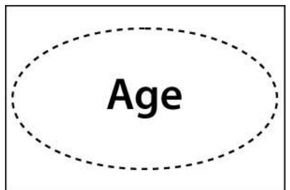


3. Single-valued attribute: Single-valued attribute contain a single value. For example, Social_Security_Number.

4. Multi-valued Attribute: If an attribute can have more than one value, it is known as a multi-valued attribute. Multi-valued attributes are depicted by the double ellipse. For example, a person can have more than one phone number, email-address, etc.



5. Derived attribute: Derived attributes are the attribute that does not exist in the physical database, but their values are derived from other attributes present in the database. For example, age can be derived from date_of_birth. In the ER diagram, Derived attributes are depicted by the dashed ellipse.



3. Relationships

The association among entities is known as relationship. Relationships are represented by the diamond-shaped box. For example, an employee works_at a department, a student enrolls in a course. Here, Works_at and Enrolls are called relationships.

Fig: Relationships in ERD

Cardinality describes the number of entities in one entity set, which can be associated with the number of entities of other sets via relationship set.

Types of Cardinalities

1. One to One: One entity from entity set A can be contained with at most one entity of entity set B and vice versa. Let us assume that each student has only one student ID, and each student ID is assigned to only one person. So, the relationship will be one to one.

2. One to many: When a single instance of an entity is associated with more than one instances of another entity then it is called one to many relationships. For example, a client can place many orders; a order cannot be placed by many customers.

3. Many to One: More than one entity from entity set A can be associated with at most one entity of entity set B, however an entity from entity set B can be associated with more than one entity from entity set A. For example - many students can study in a single college, but a student cannot study in many colleges at the same time.

4. Many to Many: One entity from A can be associated with more than one entity from B and vice-versa. For example, the student can be assigned to many projects, and a project can be assigned to many students.

The user interface is the front-end application view to which the user interacts to use the software. The software becomes more popular if its user interface is:

1. **Attractive**
2. **Simple to use**
3. **Responsive in a short time**
4. **Clear to understand**
5. **Consistent on all interface screens**

Coding Standards and Guidelines

Good software development organizations want their programmers to maintain to some well-defined and standard style of coding called coding standards. They usually make their own coding standards and guidelines depending on what suits their organization best and based on the types of software they develop. It is very important for the programmers to maintain the coding standards otherwise the code will be rejected during code review.

Purpose of Having Coding Standards:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It improves readability, and maintainability of the code and it reduces complexity also.
- It helps in code reuse and helps to detect error easily.
- It promotes sound programming practices and increases efficiency of the programmers.

Some of the coding standards are given below:

1. **Limited use of globals:** These rules tell about which types of data that can be declared global and the data that can't be.

2. **Standard headers for different modules:** For better understanding and maintenance of the code, the header of different modules should follow some standard format and information. The header format must contain below things that is being used in various companies:
   - Name of the module
   - Date of module creation

- Author of the module
- Modification history
- Synopsis of the module about what the module does
- Different functions supported in the module along with their input output parameters
- Global variables accessed or modified by the module

3. **Naming conventions for local variables, global variables, constants and functions:** Some of the naming conventions are given below:
   - Meaningful and understandable variables name helps anyone to understand the reason of using it.
   - Local variables should be named using camel case lettering starting with small letter (e.g. **localData**) whereas Global variables names should start with a capital letter (e.g. **GlobalData**). Constant names should be formed using capital letters only (e.g. **CONSDATA**).
   - It is better to avoid the use of digits in variable names.
   - The names of the function should be written in camel case starting with small letters.
   - The name of the function must describe the reason of using the function clearly and briefly.

4. **Indentation:** Proper indentation is very important to increase the readability of the code. For making the code readable, programmers should use White spaces properly. Some of the spacing conventions are given below:
   - There must be a space after giving a comma between two function arguments.
   - Each nested block should be properly indented and spaced.
   - Proper Indentation should be there at the beginning and at the end of each block in the program.
   - All braces should start from a new line and the code following the end of braces also start from a new line.

5. **Error return values and exception handling conventions:** All functions that encountering an error condition should either return a 0 or 1 for simplifying the debugging. On the other hand, Coding guidelines give some general suggestions regarding the coding style that to be followed for the betterment of understandability and readability of the code. Some of the coding guidelines are given below :

6. **Avoid using a coding style that is too difficult to understand:** Code should be easily understandable. The complex code makes maintenance and debugging difficult and expensive.

7. **Avoid using an identifier for multiple purposes:** Each variable should be given a descriptive and meaningful name indicating the reason behind using it. This is not possible if an identifier is used for multiple purposes and thus it can lead to confusion to the reader. Moreover, it leads to more difficulty during future enhancements.

8. **Code should be well documented:** The code should be properly commented for understanding easily. Comments regarding the statements increase the understandability of the code.

9. **Length of functions should not be very large:** Lengthy functions are very difficult to understand. That's why functions should be small enough to carry out small work and lengthy functions should be broken into small ones for completing small tasks.

Concept of user interface

The user interface is the front-end application view to which the user interacts to use the software.

The user interface (UI) is the point of human-computer interaction and communication in a device. This can include display screens, keyboards, a mouse and the appearance of a desktop. It is also the way through which a user interacts with an application or a website.

User Interface Design Golden Rules

The following are the golden rules stated by Theo Mandel that must be followed during the design of the interface.

1. **Define the interaction modes in such a way that does not force the user into unnecessary or undesired actions:** The user should be able to easily enter and exit the mode with little or no effort.
2. **Provide for flexible interaction:** Different people will use different interaction mechanisms, some might use keyboard commands, some might use mouse, some might use touch screen, etc., Hence all interaction mechanisms should be provided.
3. **Allow user interaction to be interruptible and undoable:** When a user is doing a sequence of actions the user must be able to interrupt the sequence to do some other work without losing the work that had been done. The user should also be able to do undo operation.
4. **Streamline interaction as skill level advances and allow the interaction to be customized:** Advanced or highly skilled user should be provided a chance to customize the interface as user wants which allows different interaction mechanisms so that user doesn't feel bored while using the same interaction mechanism.
5. **Hide technical internals from casual users:** The user should not be aware of the internal technical details of the system. He should interact with the interface just to do his work.
6. **Design for direct interaction with objects that appear on-screen:** The user should be able to use the objects and manipulate the objects that are present on the screen to perform a necessary task. By this, the user feels easy to control over the screen.