Software Design

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- Software design provides a **design plan** that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system.
- To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.
- It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.

Software Design Level

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design-** The high-level design breaks the ‘single entity-multiple component’ concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Different types of Designs Models

- **Data Design –**

  ✓ The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software.

  ✓ The 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.

- **Architectural Design –**

  ✓ 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, and the constraints that affect the way in which architectural design patterns can be applied .

  ✓ 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.

- **Interface design –**

  ✓ describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. ✓ An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design.

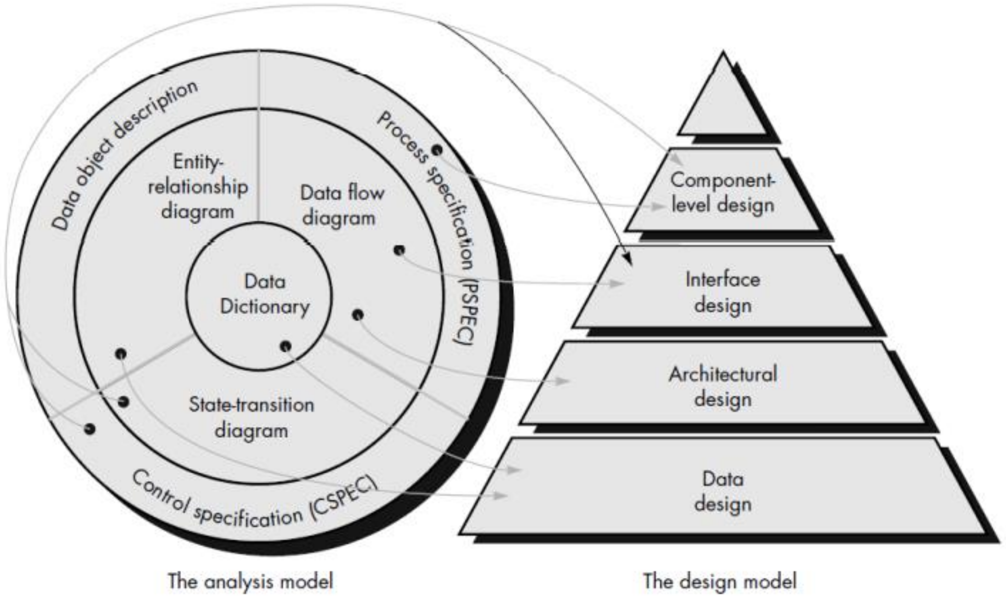


FIGURE 13.1 Translating the analysis model into a software design

- **Component-level design –**

  ✓ transforms structural elements of the software architecture into a procedural description of software components.

  ✓ Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design.

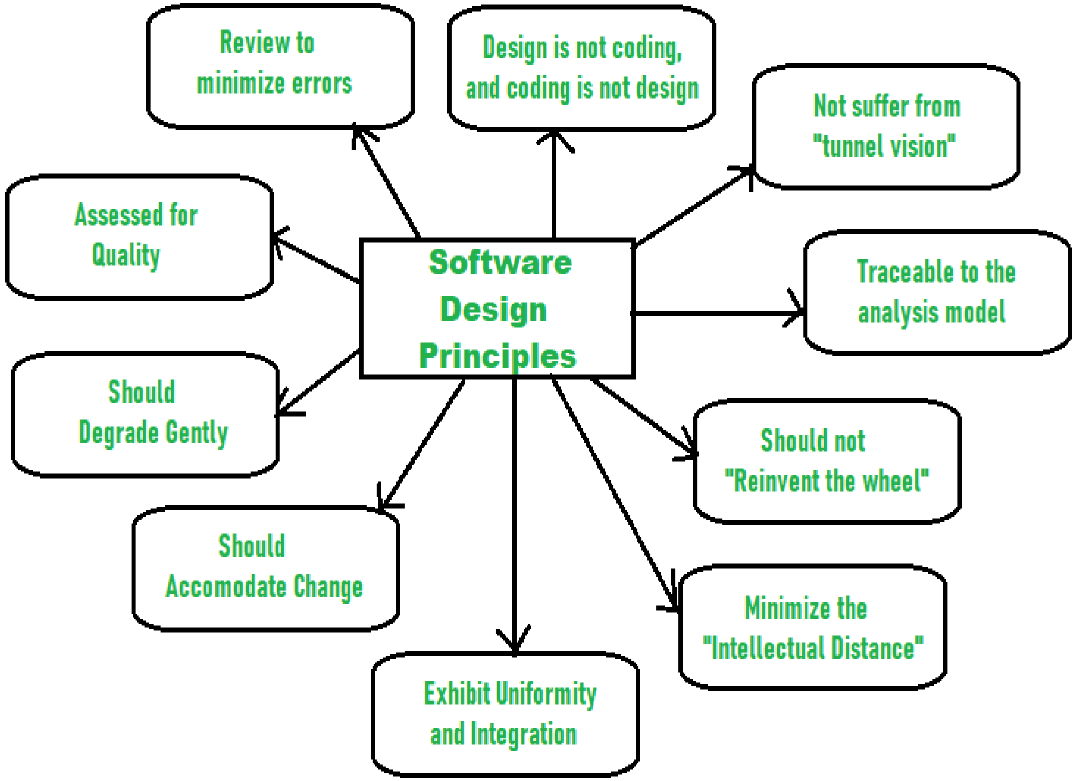
What is Good Software Design? What are the characteristics of Good software Design?

- A good design is the key to successful product.
- For good quality software to be produced, the software design must also be of good quality.
- Now, the matter of concern is how the quality of good software design is measured? This is done by observing certain characteristic or factor in software design. These characteristics are:
  - Correctness: A good design should correctly implement all the functionalities of the system.
  - Understandability: A good design should be easily understandable.
  - Efficiency: A good design should be efficient.
  - Maintainability: It should be easily amenable to change.
  - A well design software is easy to implement and reliable and allows for smooth evolution.
- In a good design the change is one part may not require the change is other part of the software.
- Good design relies on a combination of high-level systems thinking and low-level component knowledge.
- In modern software design, best practice revolves around creating modular components that you can call and deploy as needed.
- In doing this, you build software that is reusable, extensible, and easy to test.
- But before you can create these components, you need to consider what functionality users (or other software) will need out of the software you're creating.

Software Design Principles

- Including above property following are the main principles of software design:

1. The design process should not suffer from ‘tunnel vision.’
2. The design should be traceable to the analysis model.
3. The design should not reinvent the wheel.
4. The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world.
5. The design should exhibit uniformity and integration.
6. The design should be structured to accommodate change.
7. The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
8. Design is not coding, coding is not design.
9. The design should be assessed for quality as it is being created, not after the fact.
10. The design should be reviewed to minimize conceptual (semantic) errors.



Software Design Principles

Review to minimize errors

Design is not coding, and coding is not design

Not suffer from "tunnel vision"

Assessed for Quality

Traceable to the analysis model

Should Degrade Gently

Should not "Reinvent the wheel"

Should Accomodate Change

Minimize the "Intellectual Distance"

Exhibit Uniformity and Integration

Software Design Concept

- **1. Abstraction**
  - A solution is stated in large terms using the language of the problem environment at the highest level abstraction.
  - The lower level of abstraction provides a more detail description of the solution.
  - A sequence of instruction that contain a specific and limited function refers in a procedural abstraction.
  - A collection of data that describes a data object is a data abstraction.
  - Wasserman: “Abstraction permits one to concentrate on a problem at some level of abstraction without regard to low level details”
  - **Types of Abstraction**
  - **Functional Abstraction/Procedural Abstraction**
    - A module is specified by the method it performs.
    - Instructions are given in a named sequence
    - Each instruction has a limited function
    - The details of the algorithm to accomplish the functions are not visible to the user of the function.
    - Functional abstraction forms the basis for **Function oriented design approaches**.
  - **Data Abstraction**
    - Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for **Object Oriented design approaches**.
  - **Control Abstractions;**
    - A program control mechanism without specifying internal details, e.g., semaphore, rendezvous

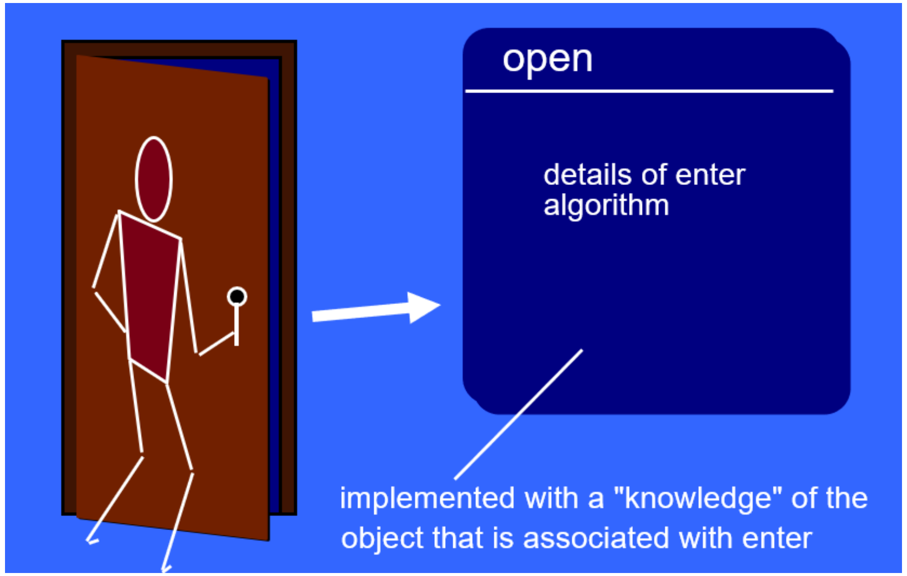

open
details of enter algorithm
implemented with a "knowledge" of the object that is associated with enter

Fig:Procedural Abstraction

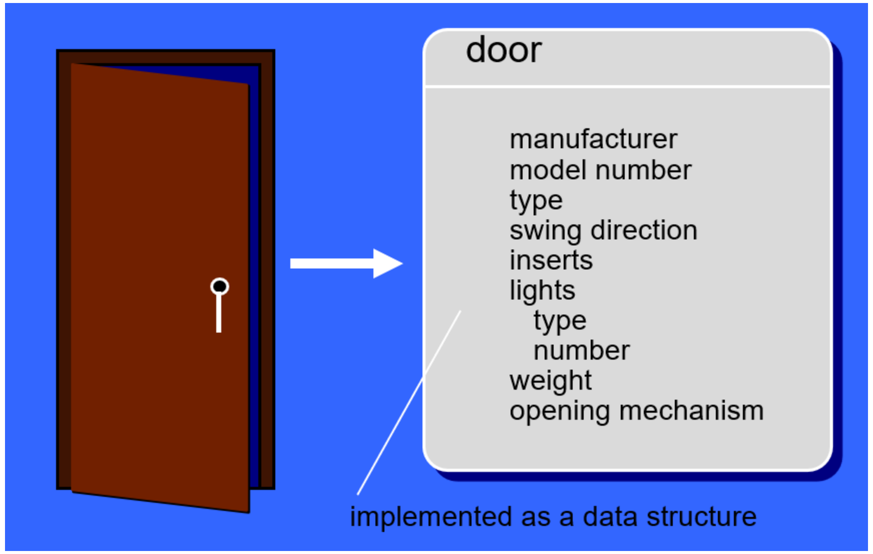
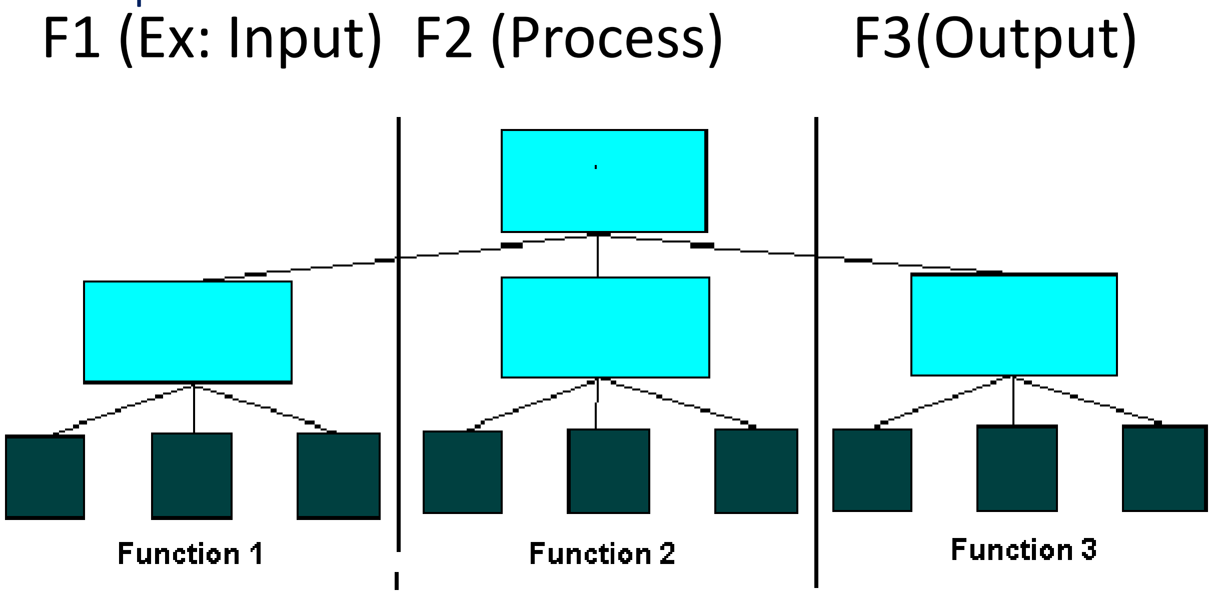


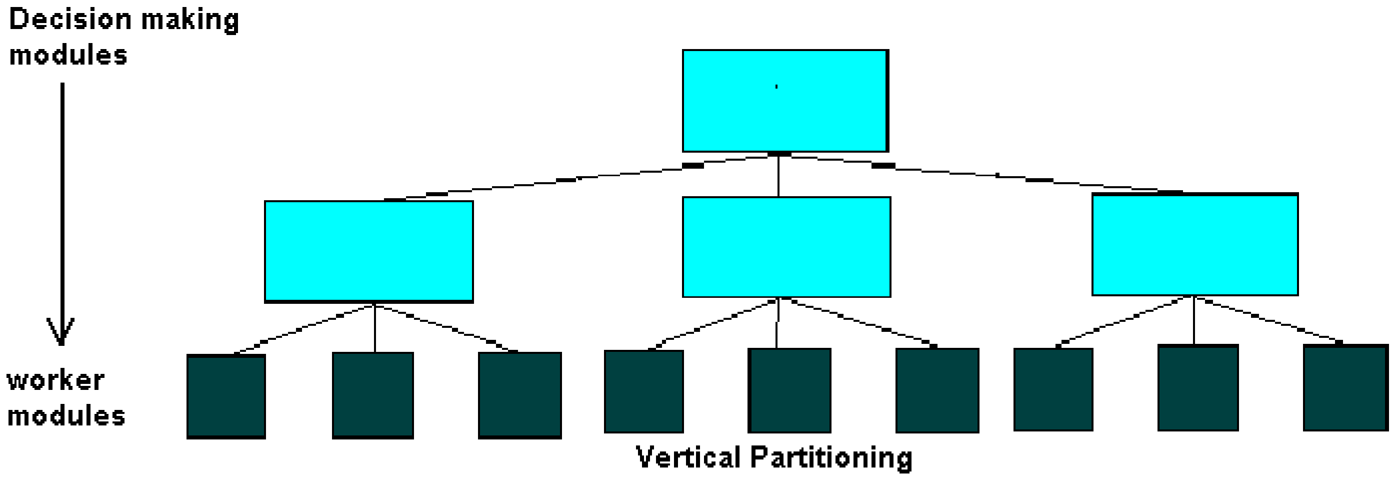
Fig: Data Abstraction

- **2.Architecture**
  - The complete structure of the software is known as software architecture.
  - Structure provides conceptual integrity for a system in a number of ways.
  - The architecture is the structure of program modules where they interact with each other in a specialized way.
  - The components use the structure of data.
  - The aim of the software design is to obtain an architectural framework of a system.
  - The more detailed design activities are conducted from the framework.
- Desired properties of an architectural design
  - **Structural Properties**
    - This defines the components of a system and the manner in which these interact with one another.
  - **Extra Functional Properties**
    - This addresses how the design architecture achieves requirements for performance, reliability and security
  - **Families of Related Systems**
    - The ability to reuse architectural building blocks
- **Structural Partitioning:**
  - **Horizontal Partitioning**
    - Easier to test
    - Easier to maintain (questionable)
    - Propagation of fewer side effects (questionable)
    - Easier to add new features



Structural Partitioning contd…

- **Vertical Partitioning**
  - Control and work modules are distributed top down
  - Top level modules perform control functions
  - Lower modules perform computations
    - Less susceptible to side effects
    - Also very maintainable

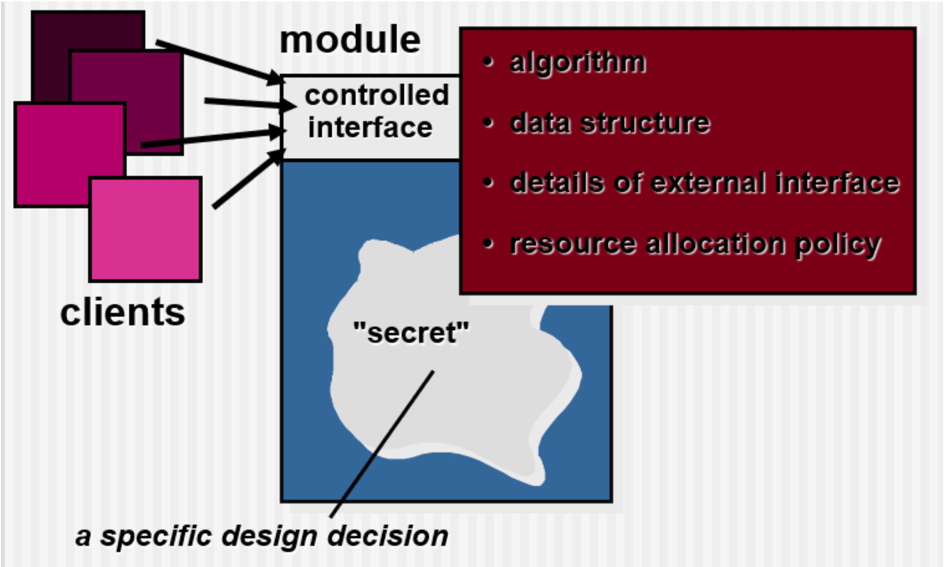


- **3. Patterns**
  - A design pattern describes a design structure and that structure solves a particular design problem in a specified content.
- **4. Modularity**
  - A software is separately divided into name and addressable components. Sometime they are called as modules which integrate to satisfy the problem requirements.
  - Modularity is the single attribute of a software that permits a program to be managed easily.

    Modules are characterized by design decisions that are hidden from others

    Modules communicate only through well defined interfaces

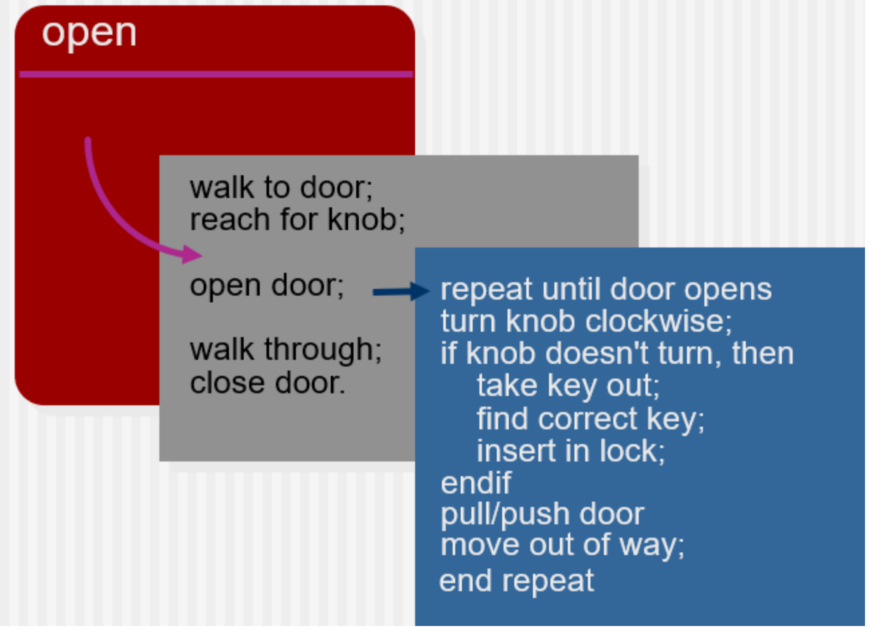
- **5. Information hiding**
  - Modules must be specified and designed so that the information like algorithm and data presented in a module is not accessible for other modules not requiring that information.
  - Modules are characterized by design decisions that are hidden from others
  - Modules communicate ***only*** through well defined interfaces
  - Enforce access constraints to local entities and those visible through interfaces
  - Very important for accommodating change and reducing coupling
  - Why Information Hiding?
    - reduces the likelihood of “side effects”
    - limits the global impact of local design decisions
    - emphasizes communication through controlled interfaces
    - discourages the use of global data
    - leads to encapsulation—an attribute of high quality design
    - results in higher quality software


module
controlled interface
• algorithm
• data structure
• details of external interface
• resource allocation policy
clients
"secret"
a specific design decision

- **6. Functional independence**
  - The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.
  - The functional independence is accessed using two criteria i.e Cohesion and coupling.

Software Design Concept contd…

- **7. Refinement**
  - Refinement is a top-down design approach.
  - It is a process of elaboration.
  - A program is established for refining levels of procedural details.
  - A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement are reached.
  - ***Fig:Stepwise Refinement***



open

```
walk to door;
reach for knob;

open door;

walk through;
close door.
```

```
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
end repeat
```

- **8. Refactoring**
  - It is a reorganization technique which simplifies the design of components without changing its function behavior.
  - Refactoring is the process of changing the software system in a way that it does not change the external behavior of the code still improves its internal structure.
- **9. Design classes**
  - The model of software is defined as a set of design classes.
  - Every class describes the elements of problem domain and that focus on features of the problem which are user visible.

Design strategy

- In general, there are two types of designing strategies that are mainly followed in the designing phase:
- 1.Top-down Design Strategy
- 2.Bottom-up Design Strategy
- **Top-down Design Strategy:**
  - Top-down strategy is the informal design strategy for breaking the problems into smaller problems.
  - The design activity must begin with the analysis of the requirements definitions and should not consider implementation details at first.
  - The top-down strategy uses the modular approach to develop the design of a system.
  - It is called so because it starts from the top or the highest-level module and moves towards the lowest level modules.
  - In this technique, the highest-level module or main module for developing the software is identified.
  - The main module is divided into several smaller and simpler submodules or segments based on the task performed by each module.
  - Then, each submodule is further subdivided into several submodules of next lower level forming the hierarchical structure.
  - This process of dividing each module into several submodules continues until the lowest level modules, which cannot be further subdivided, are not identified.
  - This strategy is suitable if the specifications are clear and development is from the scratch that is if the solutions of the software need to be developed from the ground level, top-down design best suits the purpose.
  - In top-down design system components are derived from *the project specifications.*

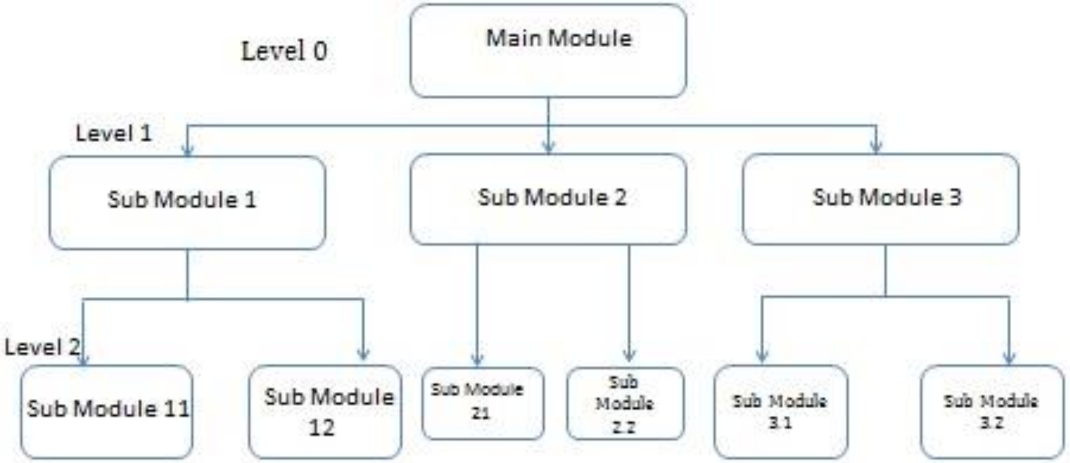


Fig: Top-down Design Strategy

Advantages:

•The main advantage of top down approach is that its strong focus on requirements helps to make a design responsive according to its requirements.

Disadvantages:

•Project and system boundaries tends to be application specification-oriented. Thus it is more likely that advantages of component reuse will be missed.

•The system is likely to miss, the benefits of a well-structured, simple architecture.

- **Bottom-up Design Strategy:**
  - Bottom-Up Strategy follows the modular approach to develop the design of the system.
  - It is called so because it starts from the bottom or the most basic level modules and moves towards the highest level modules.
  - In this techniques:
    - The modules at the most basic or the lowest level are identified.
    - These modules are then grouped together based on the function performed by each module to form the next higher level modules.
    - Then, these modules are further combined to form the next higher-level modules.
    - This process of grouping several simpler modules to form higher level modules continues until the main module of system development process is achieved.
    - The amount of abstraction grows high as the design moves to more high levels.
    - By using the basic information existing system, when a new system needs to be created, the bottom up strategy suits the purpose.
  - Thus in bottom-up design, design start from the bottom with problems that we already know who to solve. From there we can work upwards a solution to the overall problem.
  - Bottom-up techniques is used where objectives are not clear.

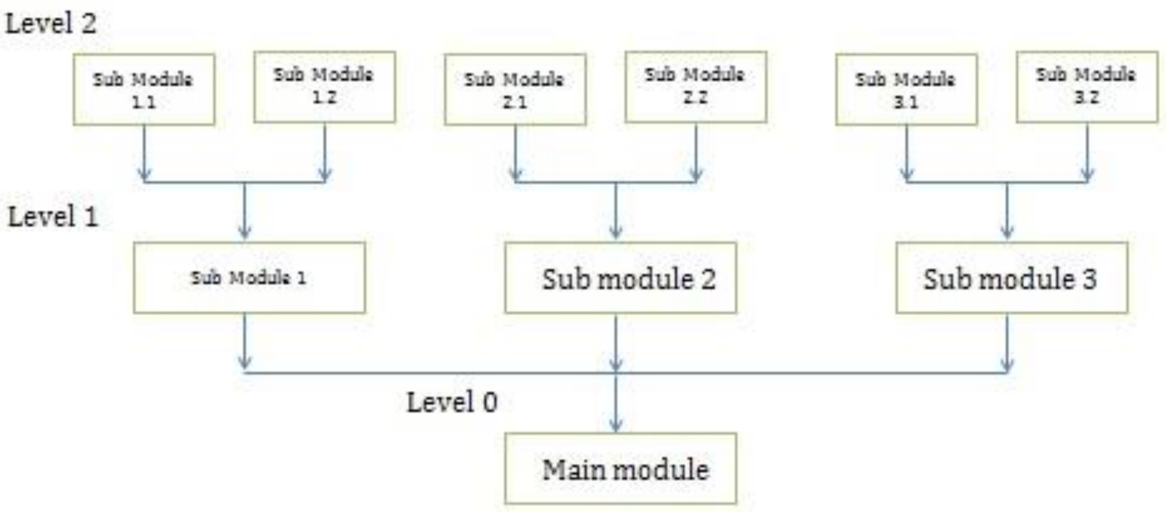


Fig: Bottom-Up Strategy

- **Advantages:**
  - The economics can result when general solutions can be reused.
  - It can be used to hide the low-level details of implementation and be merged with top-down technique.
- **Disadvantages:**
  - It is not so closely related to the structure of the problem.
  - High quality bottom-up solutions are very hard to construct.
  - It leads to proliferation of 'potentially useful' functions rather than most appropriate ones.

Architectural Design

- **Architectural design** is a process for identifying the sub-systems making up a system and the framework for sub-system control and communication.
- The output of this design process is a description of the software architecture.
- Architectural design is an early stage of the system design process. It represents the link between specification and design processes and is often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.
- Software architectures can be designed at **two levels of abstraction**:
- **Architecture in the small** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- **Architecture in the large** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

- **Advantages** :
  - **Stakeholder communication**: Architecture may be used as a focus of discussion by system stakeholders.
  - **System analysis**: Well-documented architecture enables the analysis of whether the system can meet its non-functional requirements.
  - **Large-scale reuse**: The architecture may be reusable across a range of systems or entire lines of products.
- Software architecture is most often represented using simple, informal **block diagrams** showing entities and relationships.
  - **Pros**: simple, useful for communication with stakeholders, great for project planning.
  - **Cons**: lack of semantics, types of relationships between entities, visible properties of entities in the architecture.
- **Uses of architectural models**:
  - **As a way of facilitating discussion about the system design**
    - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
  - **As a way of documenting an architecture that has been designed**
    - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

- **Goals of Software Architecture Design**
- Software Architecture Design has following goals:
  - The main important goal of software architecture design is to build the bridge between business and technical requirements.
  - To identify the requirements which affect the basic structure of the application.
  - Consider the overall effect of design decisions.
  - Design should be flexible to handle the changes in user scenarios and requirements.
  - To find the different ways to implement the use cases in the software.
- **Principles of Software Architecture Design**
- Following are the important principles considered while designing an architecture:
  - Application should be flexible to support new changes over time to address requirements and challenges.
  - Use UML (Unified Modeling Language) & visualizations to capture requirements, architectural & design to analyze their impact and to reduce risk.
  - Use models, views, visualizations, collaboration tool of the architecture to communicate & share design efficiently with all the stakeholders and to enable rapid communication of changes to the design.
  - Use proper information to understand the key engineering decisions and the areas where mistakes are most often made. If the decisions are right with use of proper information then the design is more flexible and less likely to be broken by changes.
  - Use Incremental and Iterative testing to improve the architecture.

Importance of Architectural Design

1. Performance:

- The right architecture covers the way for system success and the wrong architecture usually causes some form of disaster.
- Performance depends on raw processing power of its hardware and is important to make the system successful.
- It helps to validate software architectural design choices with respect to performance indices.

2. Scalability:

- Scalability of the system is related to performance.
- It considers how quickly the system performs its current workload.
- It focuses on the predictability of the system's performance as the workload increases.

3. Reliability:

- Reliability is the probability of failure free software architecture design.
- It is an important factor affecting on system reliability.

4. Security:

- Security provides method to deal with requirements.
- It helps to manage the risk which may be arises in architectural design.

5. Modifiability:

- It is possible to make minor changes in the architectural design.

6. Reusability:

- Reusability makes easy to reuse the code in a new system.

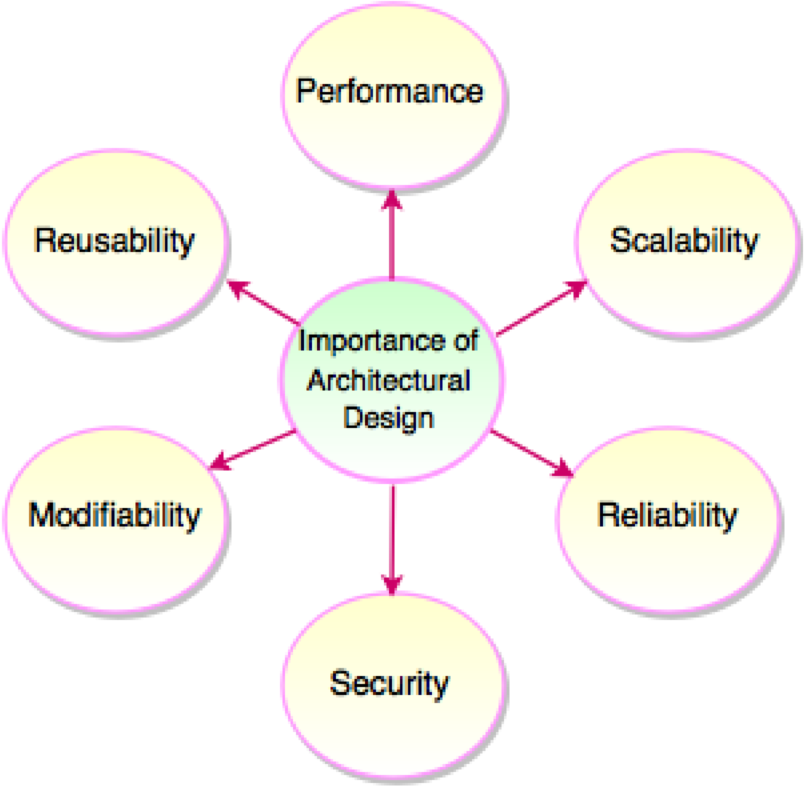


Fig. Importance of Architectural Design

Architectural views

- Each architectural model only shows one view or perspective of the system.
- It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network.
- For both design and documentation, you usually need to present multiple views of the software architecture.
- Architectural views are representations of the overall architecture.
- Architectural views are meaningful to one or more stakeholders in the system.
- It is based on the use of multiple, concurrent views.
- **4+1 view model of software architecture:**
  - 4+1 view model is used for describing the architecture of software intensive system.
  - This model is based on the use of multiple concurrent views.
  - These views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers.
- **Why is it called 4 + 1 instead of just 5?**
- Software Architecture have four views:
  - Logical view
  - Process View
  - Development view
  - Physical View
- In addition to these four views,use cases are used to illustrate the architecture serving as the + 1 view. Hence, the model contains 4 + 1 views.
- When all the views are finished, use cases are effectively redundant.
- This model gives the detail of high requirements of the system.

1. Logical View:

- Logical view contains the information about the various parts of the system.
- It is concerned with the functionality that the system provides to end-users.
- UML diagrams are used to represent logical view, including class diagram, communication diagram, sequence diagram.

2. Development View:

- Development view is also known as the **Implementation view**.
- It illustrates a system from a programmer's perspective.
- It is concerned with the software management.
- It focuses on software modules and subsystems.
- UML diagrams are used to represent development view, including package diagram, component diagram.

3. Process View:

- Process view deals with the dynamic aspects of the system.
- It explains the system process, how they communicate and focuses on the runtime behavior of the system.
- It describes the concurrent process within the system.
- UML diagrams are used to represent process view, included in the activity diagram.

4. Physical View:

- Physical view is also known as the **Deployment view**.
- It describes the physical deployment of the system.
- It depicts the system from a system engineer's point of view.
- UML diagrams are used to represent physical view, included in the deployment diagram.

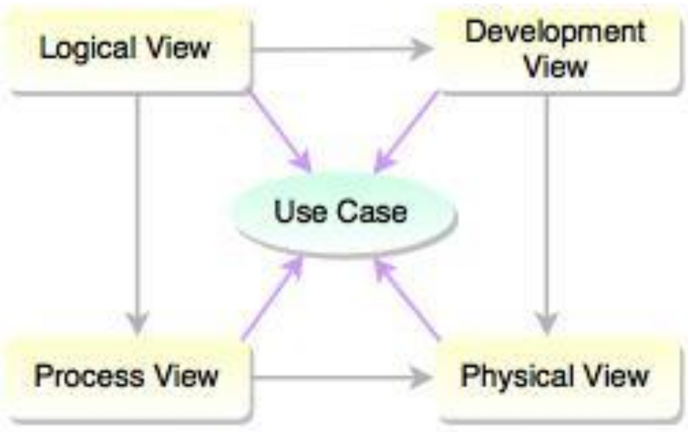


Fig. 4 + 1 View Architecture

5. Use Case View :

- Use case view describes the functionality of the system from the perspective of the outside world.
- It contains the diagram describing what the system is supposed to do from a black-box perspective.
- It contains use case diagrams.
- This view can guide all other views.

Architectural patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a **stylized description of a good design practice**, which has been tried and tested in different environments. Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.
- **Model-View-Controller(MVC) Architecture Patterns**
  - Serves as a basis of interaction management in many web-based systems.
  - Decouples three major interconnected components:
    - The model is the central component of the pattern that directly manages the data, logic and rules of the application. It is the application's dynamic data structure, independent of the user interface.
    - A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible.
    - The controller accepts input and converts it to commands for the model or view.
  - Supported by most language frameworks.

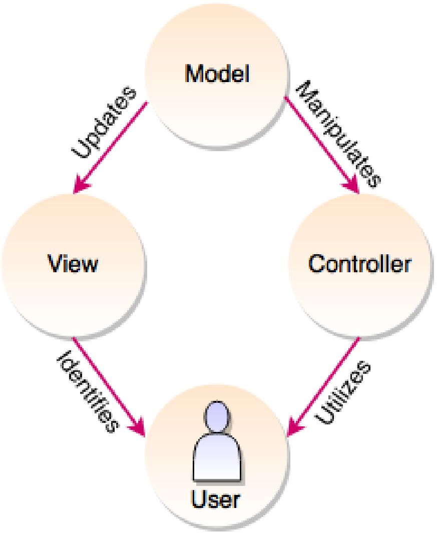
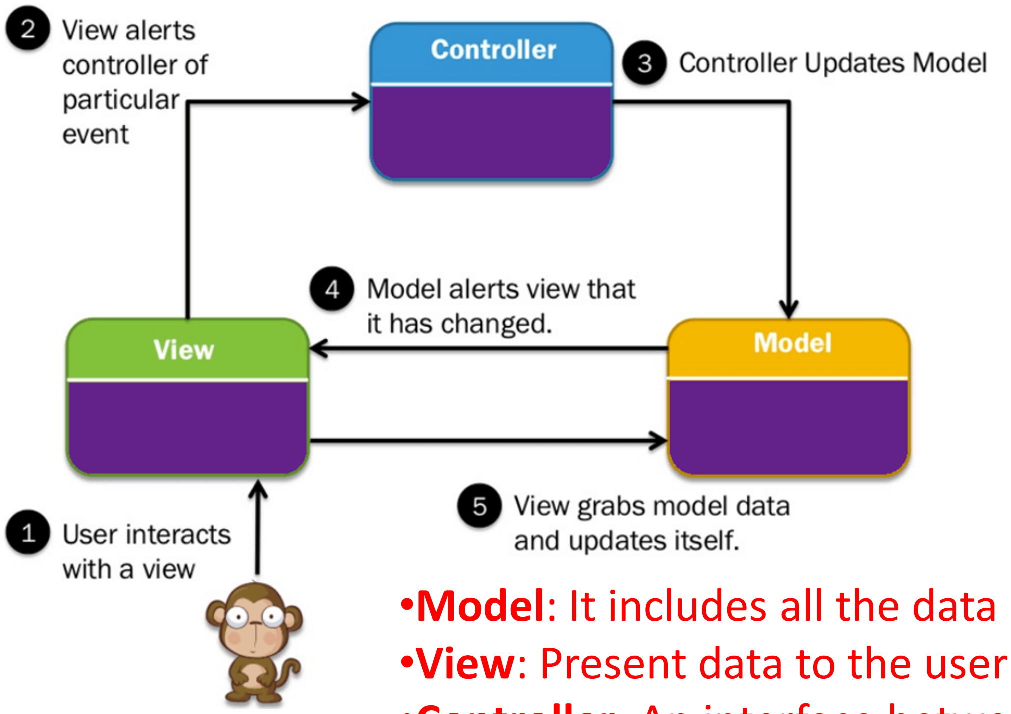
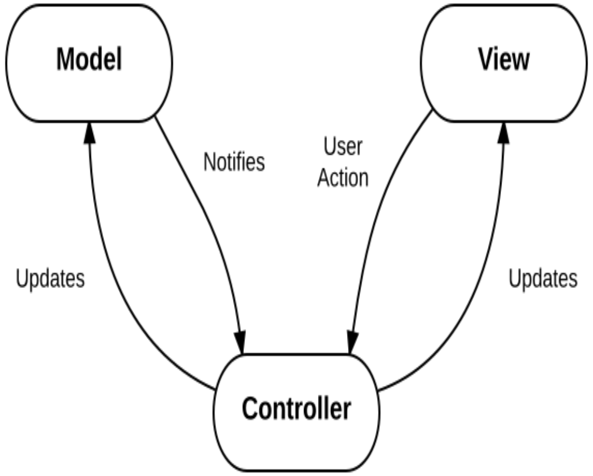


Fig. MVC Architecture


2 View alerts controller of particular event
Controller
3 Controller Updates Model
4 Model alerts view that it has changed.
View
Model
5 View grabs model data and updates itself.
1 User interacts with a view



- **Model**: It includes all the data and its related logic
- **View**: Present data to the user or handles user interaction
- **Controller**: An interface between Model and View components

- The controller receives all the request for the application and then works with the Model to prepare the data needed by the View.
- The View uses the data prepared by the Controller to generate the final presentable response.

Example:

Car driving mechanism is an example of the MVC model.

- Every car consist of three main parts.
- View= User interface : (Gear lever, panels, steering wheel, brake, etc.)
- Controller- Mechanism (Engine)
- Model- Storage (Petrol or Diesel tank)

Components	Description
Model	• Model is the lowest level of the pattern responsible for maintaining the data. • It is a central component of MVC architecture. • It manages the data, logic and constraints of an application. • It captures the behavior of an application domain problem. • It is the domain-specific software implementation of the application's central structure. • If there is any change in its state then it gives notification to its associated view to produce updated output and the controller to change the available set of commands. • It is an independent user interface.
View	• View is responsible for displaying all or a portion of the data to the user. • It represents any output of information in a graphical form such as diagram or chart. • View consists of presentation components which provide the visual representations of data. • View formats and presents the data from model to user.
Controller	• Controller controls the interactions between the Model and View. • It accepts an input and converts it into the commands for model or view. • Controller acts as an interface between the associated models, views and the input devices. • It sends the commands to the model to update the model's state and to it's associated view to change the view's presentation of the model.

- Examples of MVC Web Frameworks
  - Ruby on Rails
  - Django
  - CakePHP
  - Yii
  - CherryPy
  - Spring MVC
  - Catalyst
  - Rails
  - Zend Framework
  - CodeIgniter
  - Laravel
  - Fuel PHP
  - Symphony

Advantages/Disadvantages of MVC

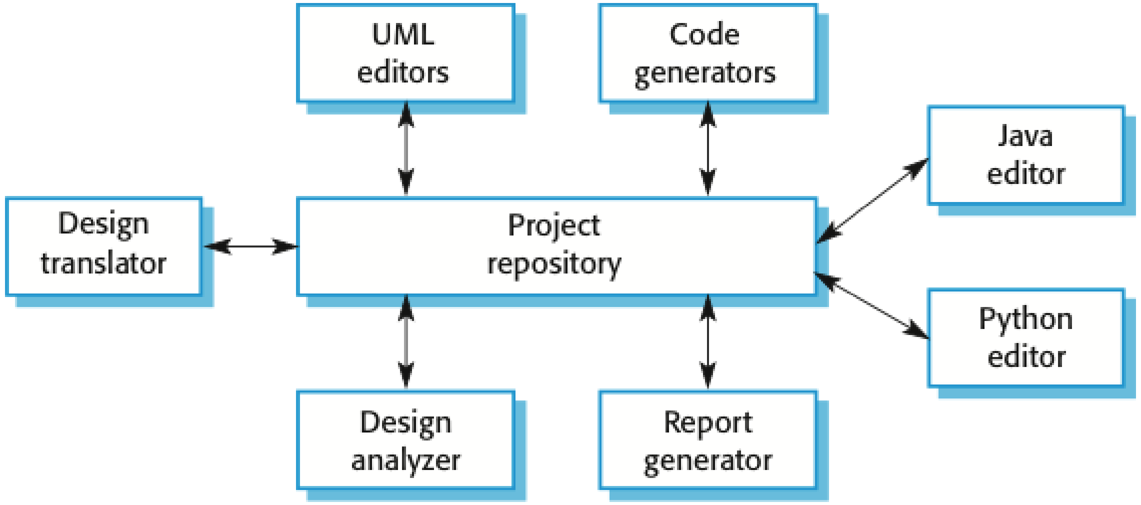
- **Advantages of MVC architecture:**
  - Development of the application becomes fast.
  - Easy for multiple developers to collaborate and work together.
  - Easier to Update the application.
  - Easier to Debug as we have multiple levels properly written in the application.
- **Disadvantages of MVC architecture:**
  - It is hard to understand the MVC architecture.
  - The complexity is high.
  - Not suitable for small applications.

Architectural Styles

- The **architectural style** is a very specific solution to a particular software, which typically focuses on how to organize the code created for the software.
- It focuses on creating the layers and modules of the software and allowing an appropriate interaction between the various modules for giving the right results upon implementation.
- Each style will describe a system category that consists of :
  - A set of components(eg: a database, computational modules) that will perform a function required by the system.
  - The set of connectors will help in coordination, communication, and cooperation between the components.
  - Conditions that how components can be integrated to form the system.
  - Semantic models that help the designer to understand the overall properties of the system.

Repository Architecture

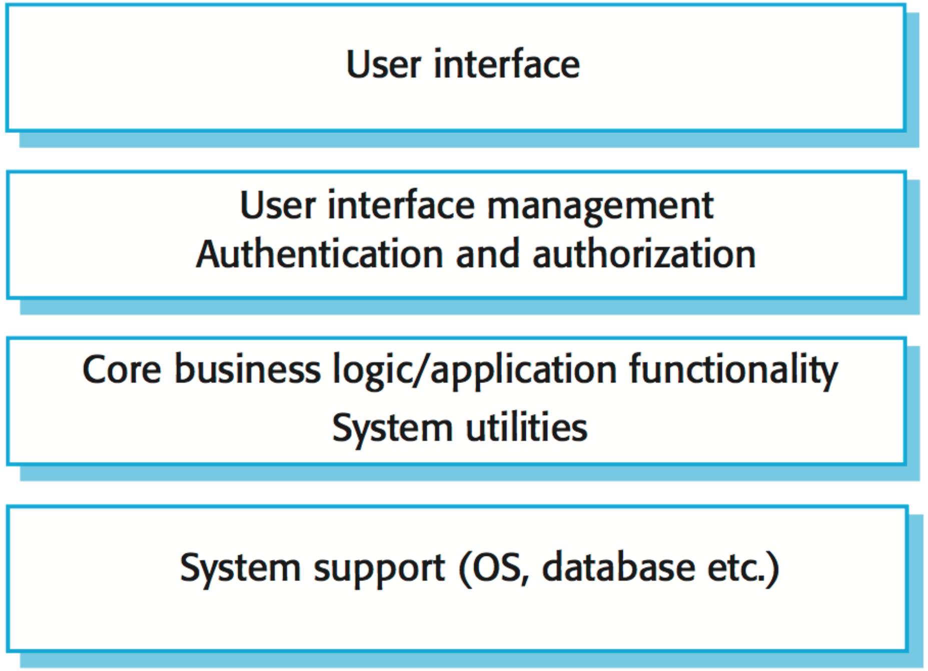
- In Repository Architecture Style, the data store is passive and the clients (software components or agents) of the data store are active, which control the logic flow.
- The participating components check the data-store for changes.
  - The client sends a request to the system to perform actions (e.g. 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.



Name	Repository Architecture
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent--they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Layered Architecture

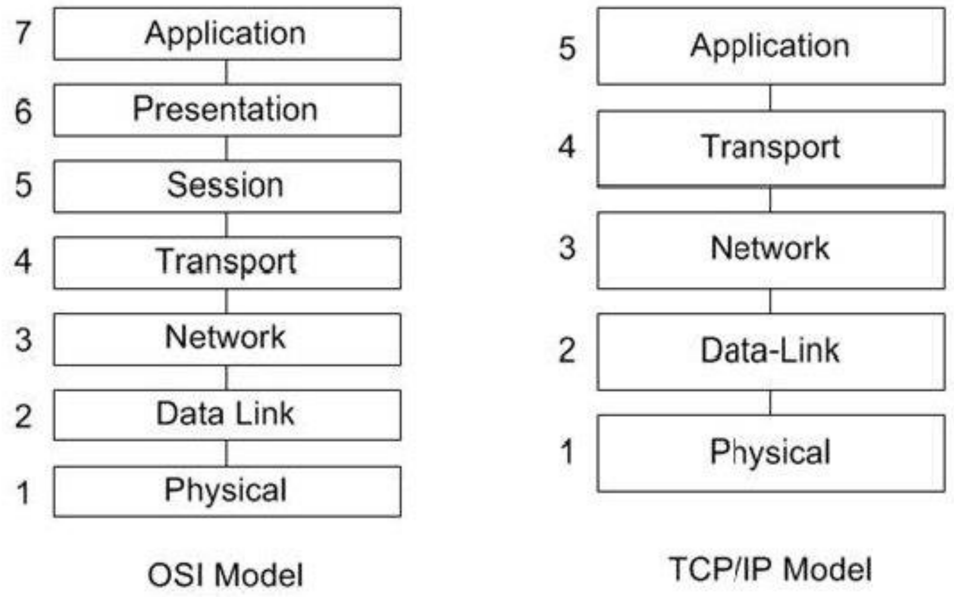
- Used to model the interfacing of sub-systems.
- Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.


User interface
User interface management
Authentication and authorization
Core business logic/application functionality
System utilities
System support (OS, database etc.)

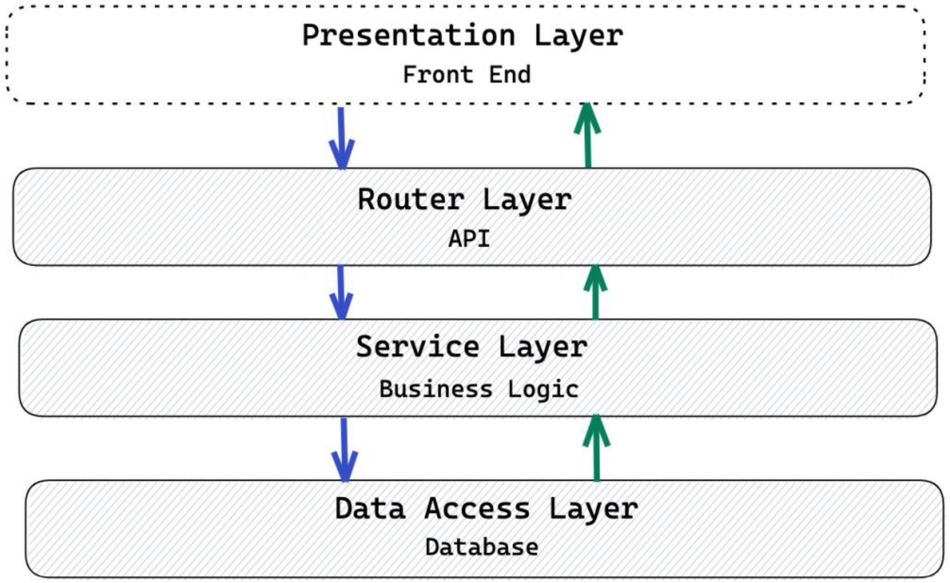
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

tiered/layered architecture example:

Open Systems Interconnection (OSI) & Transmission Control Protocol/Internet Protocol (TCP/IP)

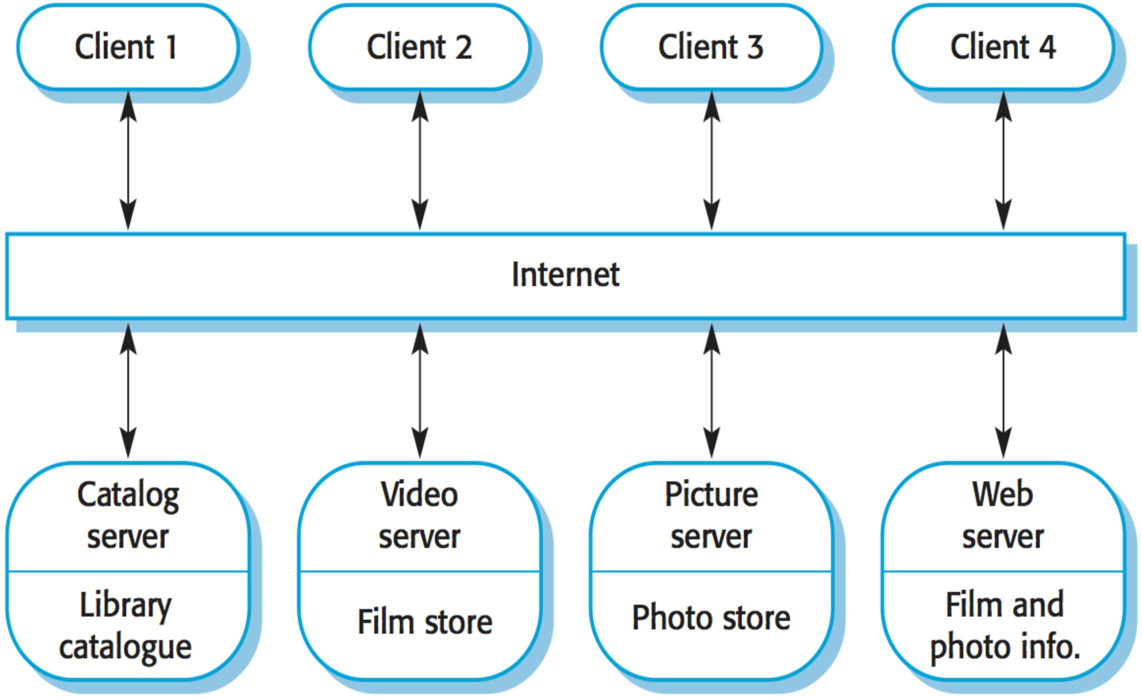


NodeJS Layered Architecture



Client-server architecture

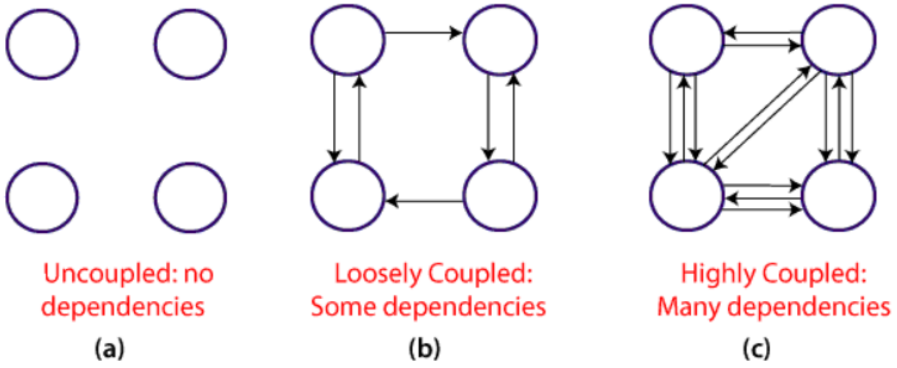
- Distributed system model which shows how data and processing is distributed across a range of components, but can also be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.



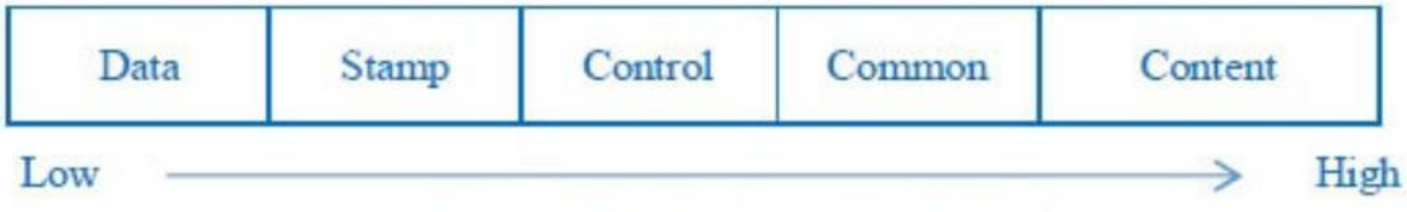
Name	Client-server
Description	In a client-server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

Coupling and Cohesion

- Coupling:
  - Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.
  - Coupling is used to define the factors of dependency and independence of each module of the software with other modules.
  - It is used as an indicator of interdependency amongst the modules, and the lower the coupling value will be, the higher the quality of the software will be.
  - The name coupling is applied for this process, as it is typically deliberated between two modules at a time.
  - The first step taken in the coupling process is to evaluate the association between the two modules and to define the functionally dependent areas in the modules.


Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

Types of Coupling



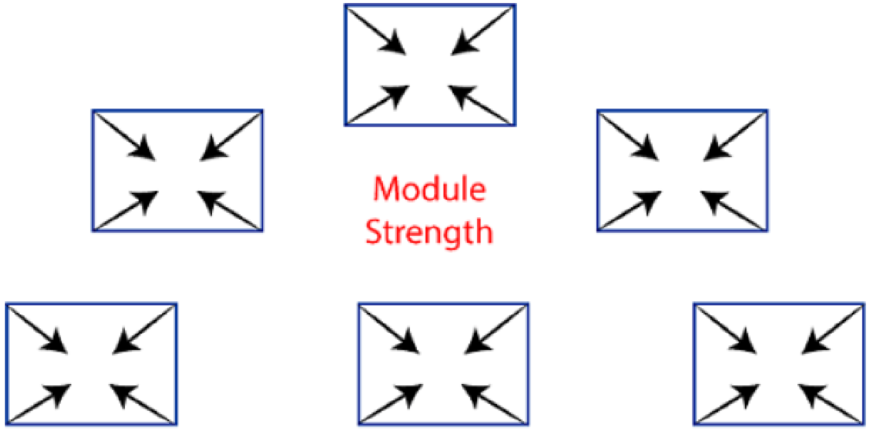
- **Data Coupling**:
  - Two modules are data coupled if they communicate through a parameter.
  - An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc.
  - This data item should be problem-related and not used for the control purpose.

- **Stamp Coupling**:
  - Two modules are stamp coupled if they communicate using a composite data item such as a record in PASCAL or a structure in C.
- **Control Coupling**:
  - Control coupling exists between two modules if data from one module is used to direct the order of instructions executed in another.
  - An example of control coupling is a flag set in one module and tested in another module.

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

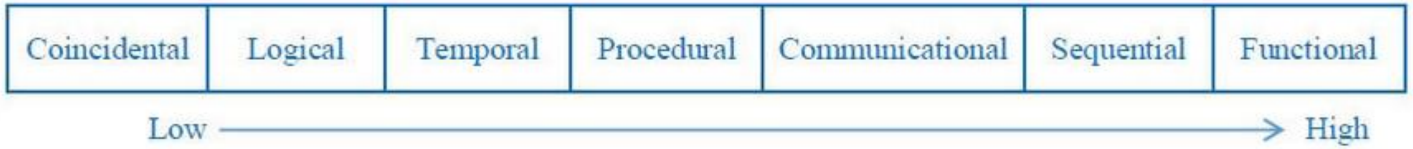
Content Coupling: Content coupling exists between two modules if they share code, e.g. a branch from one module into another module.

- Cohesion:
  - **Cohesion** is a measure of functional strength of a module.
  - Cohesion of a module is the degree to which the different functions of the module co-operate to work towards a single objective.
  - It is a measure that defines the degree of intra-dependability within an element of a module.
  - It is the degree to which all elements directed towards performing a single task are contained in the component.
  - Basically, cohesion is the internal glue that keeps the module together.
  - A good software design will have high cohesion.



Cohesion= Strength of relations within Modules

Types of Cohesion:



Coincidental | Logical | Temporal | Procedural | Communicational | Sequential | Functional

Low → High

1. **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
2. **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
3. **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.
4. **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
5. **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.
6. **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
7. **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

Coupling Vs Cohesion

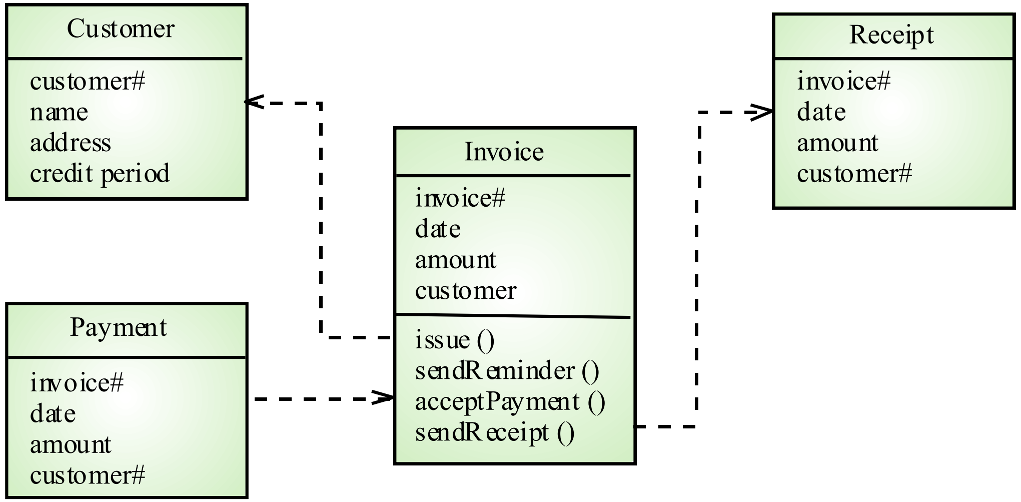
Coupling	Cohesion
• Coupling is also called Inter-Module Binding.	• Cohesion is also called Intra-Module Binding.
• Coupling shows the relationships between modules.	• Cohesion shows the relationship within the module.
• Coupling shows the relative **independence** between the modules.	• Cohesion shows the module's relative **functional** strength.
• While creating, you should aim for low coupling, i.e., dependency among modules should be less.	• While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
• In coupling, modules are linked to the other modules.	• In cohesion, the module focuses on a single thing.

Modularity and Modular Decomposition

- Modularity:
  - A technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently.
  - These modules may work as basic constructs for the entire software.
  - Designers tend to design modules such that they can be executed and/or compiled separately and independently.
  - Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.
  - Advantages
    - Smaller components are easier to maintain
    - Program can be divided based on functional aspects
    - Desired level of abstraction can be brought in the program
    - Components with high cohesion can be re-used again
    - Concurrent execution can be made possible
    - Desired from security aspect
- Modular Decomposition
  - Another structural level where sub-systems are decomposed into modules
  - This construction is based on assigning functions to components. Here the designer begins with a high level description of the function that are to be implemented and builds lower level explanations of how each components will be organized and related to other components.
  - Two modular decomposition models covered
    - **An object model** where the system is decomposed into interacting objects
    - **A data-flow model** where the system is decomposed into functional modules which transform inputs to outputs. Also known as the pipeline model
  - If possible, decisions about concurrency should be delayed until modules are implemented
  - Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle.
  - If different modules have either no interactions or little interactions with each other, then each module can be understood separately.
  - This reduces the perceived complexity of the design solution greatly.

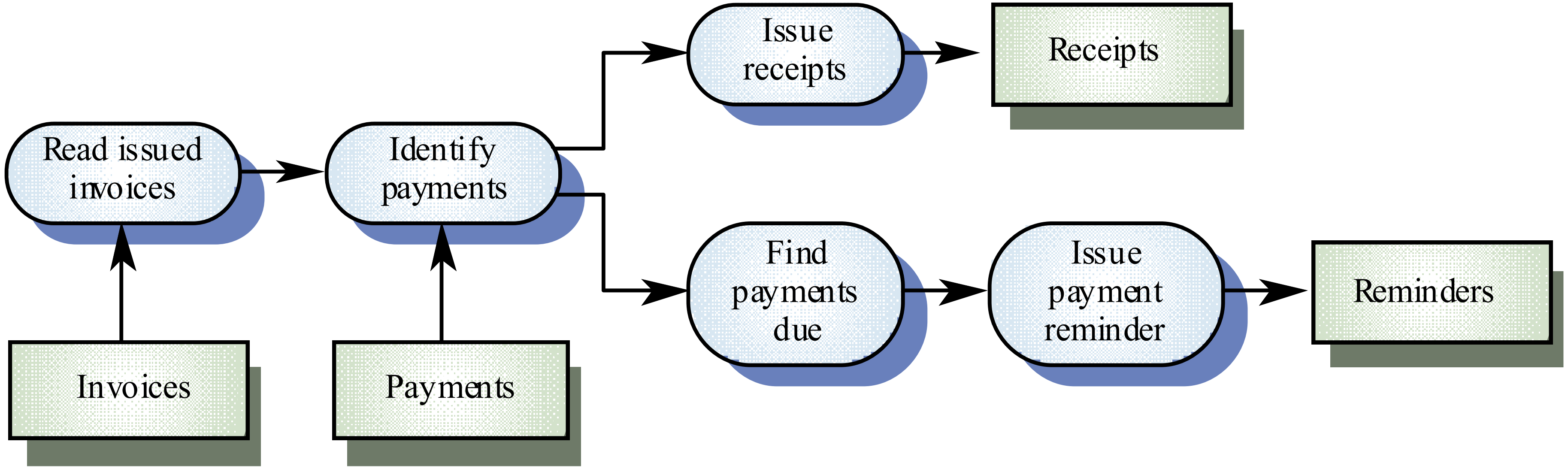
Object Model

- Structure the system into a set of loosely coupled objects with well-defined interfaces
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations
- When implemented, objects are created from these classes and some control model used to coordinate object operations
- Example :*Object Model of Invoice Processing System*



Data Flow Model

- Functional transformations process their inputs to produce outputs
- May be referred to as a pipe and filter model (as in UNIX shell)
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems
- Not really suitable for interactive systems
- Example :*Object Model of Invoice Processing System*



Procedural Design using Structured Methods

- A more methodical approach to software design is proposed by structured methods which are sets of notations and guidelines for software design.
- Rules
  - Programs were to be broken into functions and subroutines
  - There was only a single entry point and a single exit point for any function or routine.
- Structured methods often support some or all of the following models of a system:
  - □ A data-flow model
  - □ An Entity-relationship model
  - □ A structural model
  - □ An object-oriented model

Procedural Design

- The objective in procedural design is to transform structural components into a procedural description of the software.
- The step occurs after the data and program structures have been established, i.e. after architectural design. Procedural details can be represented in different ways:
- **1.Graphical Design Notation:** The most widely used notation is the flowchart. Some notation used in flowcharts are
  (i) Boxes to indicate processing steps.
  (ii) Diamond to indicate logical condition.
  (iii) Arrows to indicate flow of control.
  (iv) Two boxes connected by a line of control will indicate a Sequence

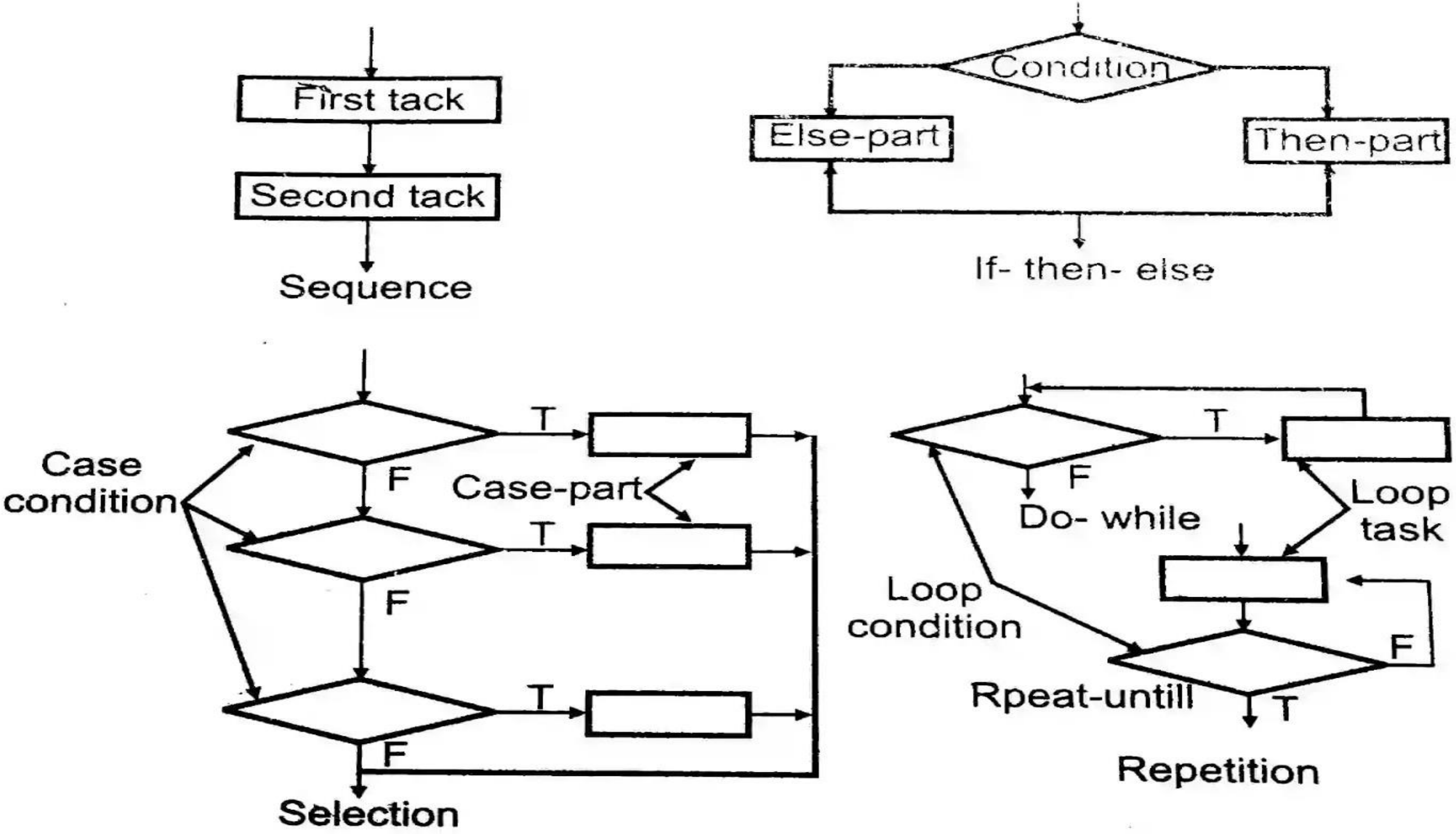
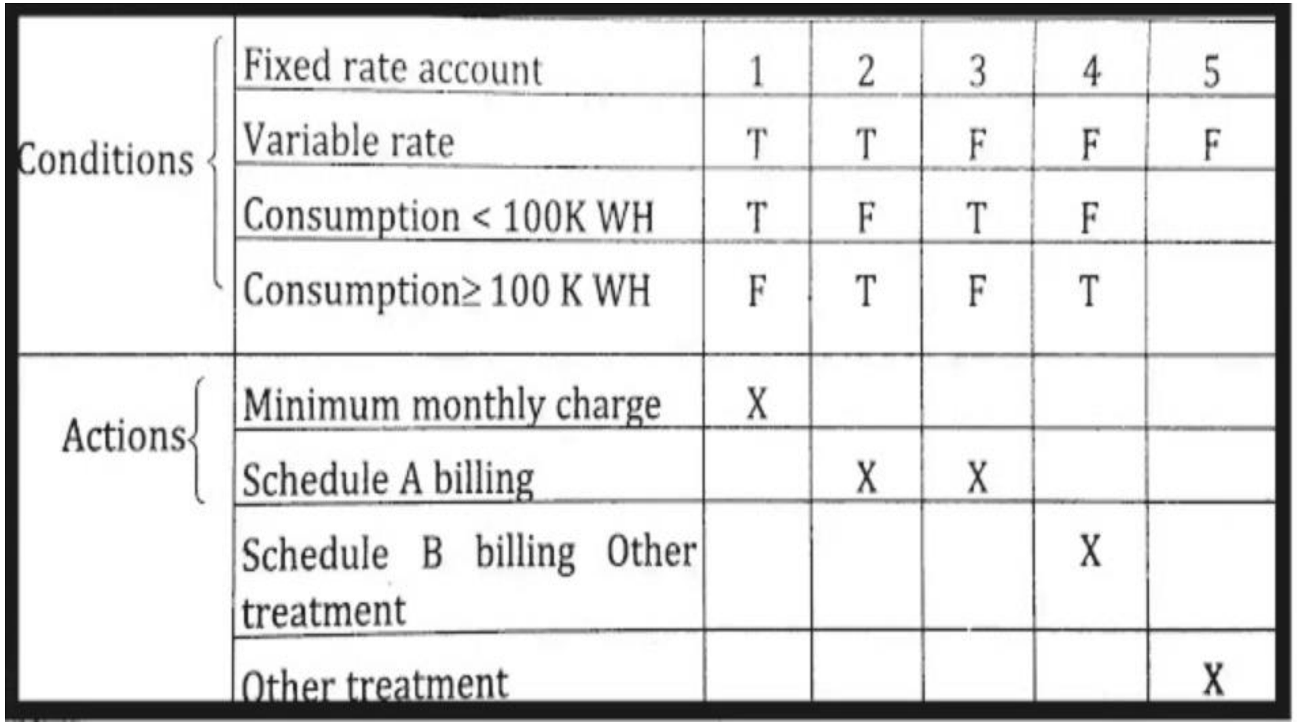

First tack
Second tack
Sequence
Condition
Else-part
Then-part
If- then- else
Case
condition
T
F
Case-part
T
F
T
F
Selection
T
F
Do- while
Loop
task
Loop
condition
F
Rpeat-untill
T
Repetition

Fig.

2.Tabular Design Notations

(i) Decision tables provide a notation that translates actions and conditions (described a processing narrative) into a tabular form.

(ii) The upper left-hard section contains a list of all conditions. The lower left-hand section lists all actions that are possible based on the conditions. The right-hand sections form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination.



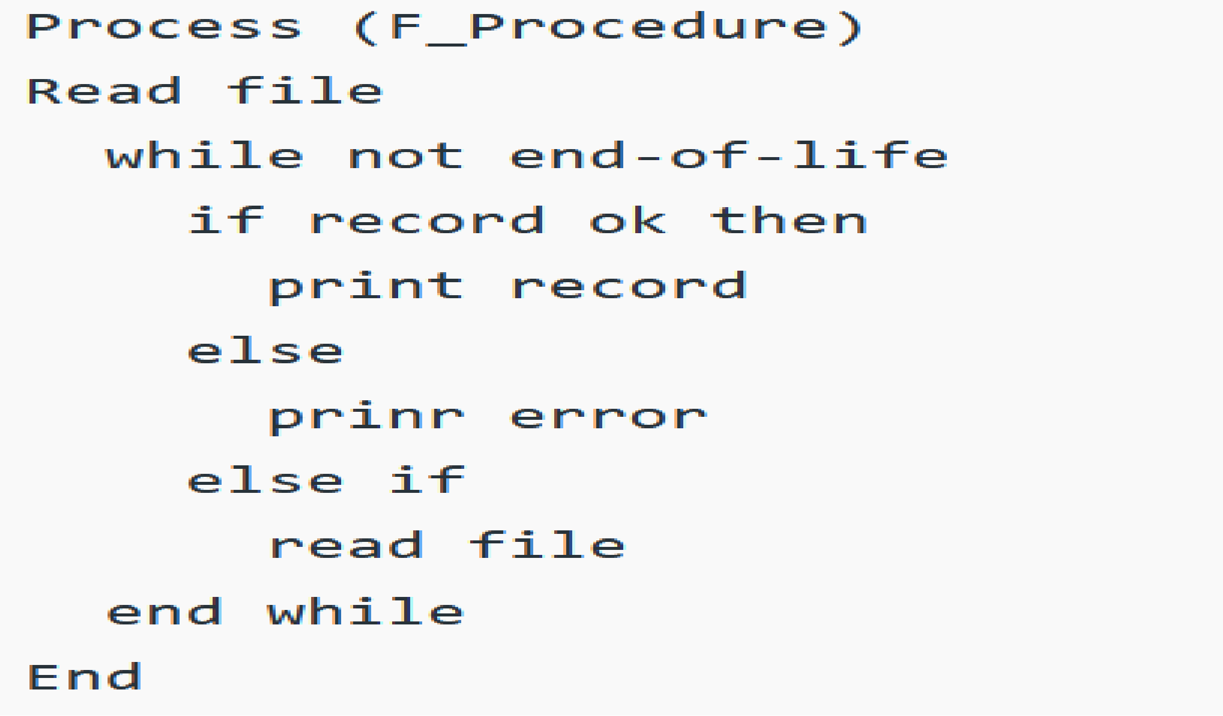
Conditions	Fixed rate account	1	2	3	4	5
	Variable rate	T	T	F	F	F
	Consumption < 100K WH	T	F	T	F	
	Consumption≥ 100 K WH	F	T	F	T	
Actions	Minimum monthly charge	X				
	Schedule A billing		X	X		
	Schedule B billing Other treatment				X	
	Other treatment					X

3. Program Design Language:

It is a method designing and documenting methods and procedures in software. It is related to pseudocode, but unlike pseudocode, it is written in plain language without any terms that could suggest the use of any programming language or library.

PDL Example

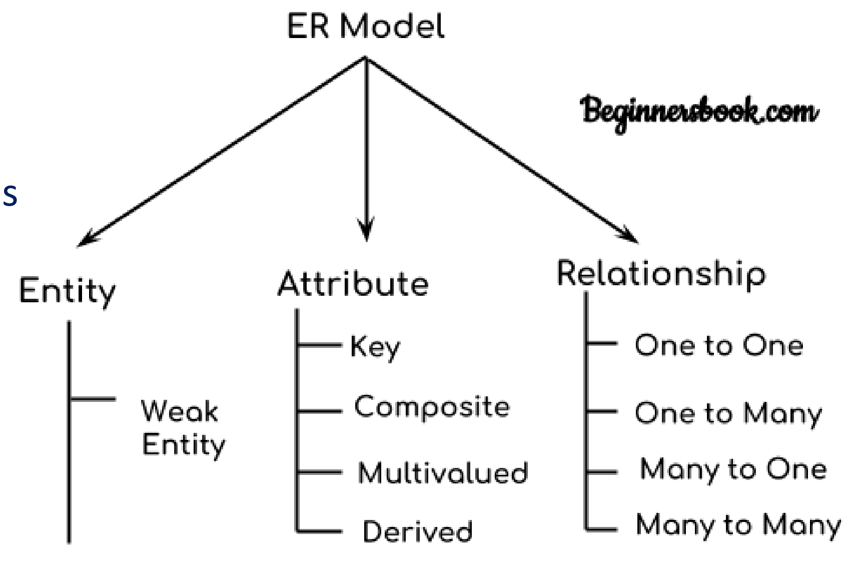
- Consider the problem of reading the record from the file. If file reading is not completed and there is no error in the record then print the information of the record otherwise print that there is an error in reading of the record. This process will continue till the whole file is completed:



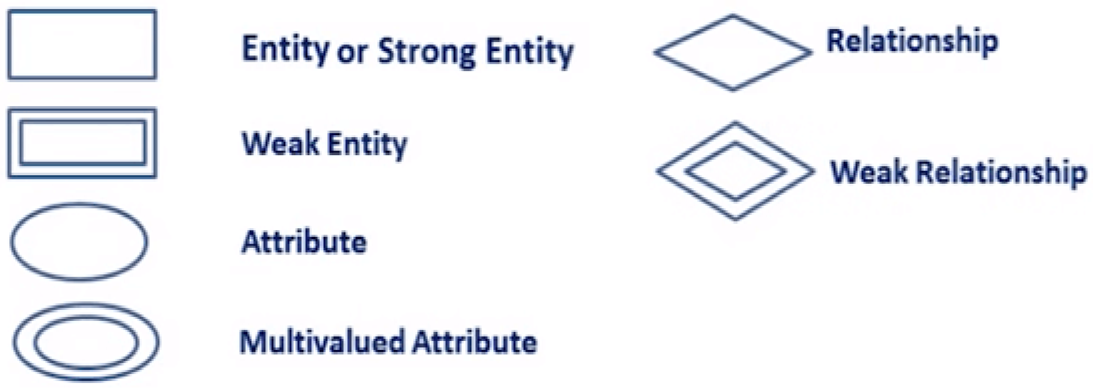
```
Process (F_Procedure)
Read file
  while not end-of-life
    if record ok then
      print record
    else
      prinr error
    else if
      read file
  end while
End
```

Entity Relationship Diagram(ERD)

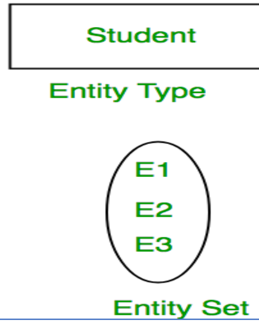
- A detailed, logical representation of the entities, associations and data elements for an organization or business area.
- **ER Diagram** stands for Entity Relationship Diagram, also known as ERD is a diagram that displays the relationship of entity sets stored in a database. In other words, ER diagrams help to explain the logical structure of databases. ER diagrams are created based on three basic concepts: entities, attributes and relationships.
- ER Diagrams contain different symbols that use rectangles to represent entities, ovals to define attributes and diamond shapes to represent relationships.
- **Components of ERD**
  - **Rectangles:** This Entity Relationship Diagram symbol represents entity types
  - **Ellipses :** Symbol represent attributes
  - **Diamonds:** This symbol represents relationship types
  - **Lines:** It links attributes to entity types and entity types with other relationship types
  - **Primary key:** attributes are underlined
  - **Double Ellipses:** Represent multi-valued attributes



Components of ER Diagram



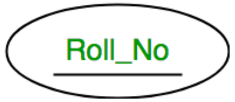
- An **Entity** may be an object with a physical existence – a particular person, car, house, or employee – or it may be an object with a conceptual existence – a company, a job, or a university course.
- An **Entity** is an object of Entity Type and set of all entities is called as entity set. e.g.; E1 is an entity having Entity Type Student and set of all students is called Entity Set.Eg:



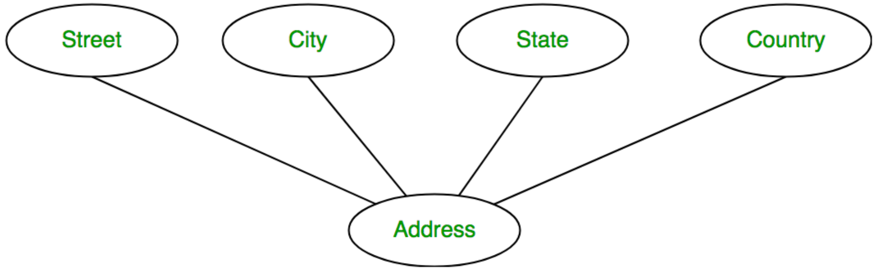
- **Attribute(s):**
  Attributes are the **properties which define the entity type**. For example, Roll_No, Name, DOB, Age, Address, Mobile_No are the attributes which defines entity type Student. In ER diagram, attribute is represented by an oval.

Types of Attributes

- **Key Attribute**
  - The attribute which **uniquely identifies each entity** in the entity set is called key attribute.For example, Roll_No will be unique for each student. In ER diagram, key attribute is represented by an oval with underlying lines.

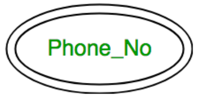


- **Composite Attribute**
  - An attribute **composed of many other attribute** is called as composite attribute. For example, Address attribute of student Entity type consists of Street, City, State, and Country. In ER diagram, composite attribute is represented by an oval comprising of ovals.



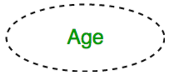
Multivalued Attribute

- An attribute consisting **more than one value** for a given entity. For example, Phone_No (can be more than one for a given student). In ER diagram, multivalued attribute is represented by double oval.



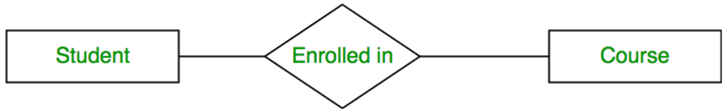
Derived Attribute

- An attribute which can be **derived from other attributes** of the entity type is known as derived attribute. e.g.; Age (can be derived from DOB). In ER diagram, derived attribute is represented by dashed oval.

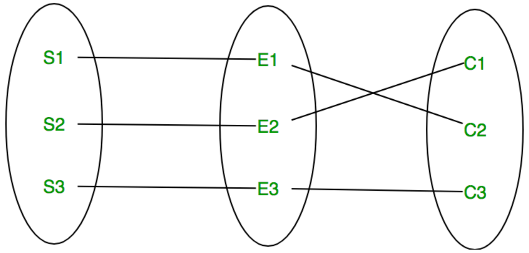


Relationship Type and Relationship Set:

- A relationship type represents the **association between entity types**. For example,'Enrolled in' is a relationship type that exists between entity type Student and Course. In ER diagram, relationship type is represented by a diamond and connecting the entities with lines.



A set of relationships of **same type** is known as **relationship set**. The following relationship set depicts S1 is enrolled in C2, S2 is enrolled in C1 and S3 is enrolled in C3.

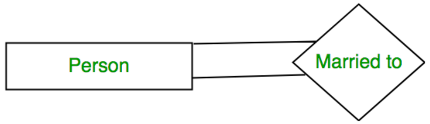


Degree of a relationship set:

The number of different entity sets **participating in a relationship** set is called as degree of a relationship set.

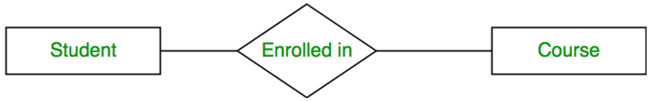
Unary Relationship

- When there is **only ONE entity set participating in a relation**, the relationship is called as unary relationship. For example, one person is married to only one person.



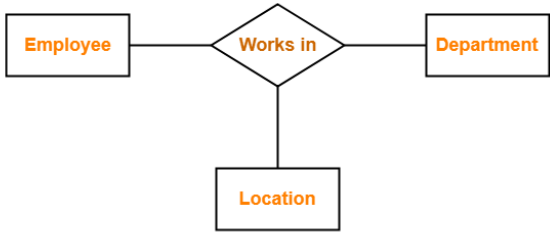
Binary Relationship

- When there are **TWO entities set participating in a relation**, the relationship is called as binary relationship.For example, Student is enrolled in Course



Ternary Relationships:

- Ternary relationship set is a relationship set where three entity sets participate in a relationship set.



Ternary Relationship Set

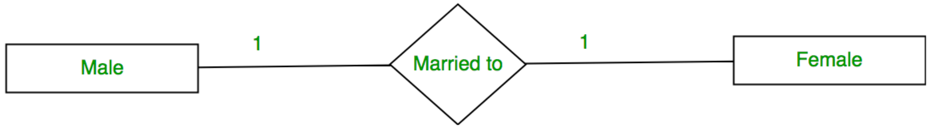
n-ary Relationship

- When there are n entities set participating in a relation, the relationship is called as n-ary relationship.

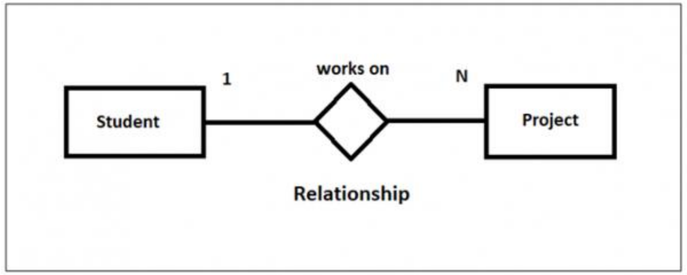
Cardinality:

- The **number of times an entity of an entity set participates in a relationship** set is known as cardinality. Cardinality can be of different types:

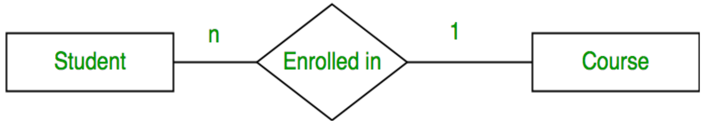
1. **One to one –** When each entity in each entity set can take part **only once in the relationship**, the cardinality is one to one. Let us assume that a male can marry to one female and a female can marry to one male. So the relationship will be one to one.



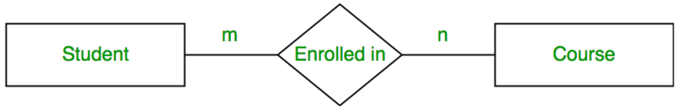
2. **One to Many-**When each entity in each entity can **take part more than once in the relationship** cardinality is one to many



2. **Many to one –** When entities in one entity set **can take part only once in the relationship set and entities in other entity set can take part more than once in the relationship set,** cardinality is many to one. Let us assume that a student can take only one course but one course can be taken by many students. So the cardinality will be n to 1. It means that for one course there can be n students but for one student, there will be only one course.



3. **Many to many –** When entities in all entity sets can **take part more than once in the relationship** cardinality is many to many. Let us assume that a student can take more than one course and one course can be taken by many students. So the relationship will be many to many.

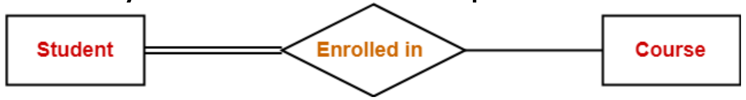


Participation Constraint:

Participation Constraint is applied on the entity participating in the relationship set.

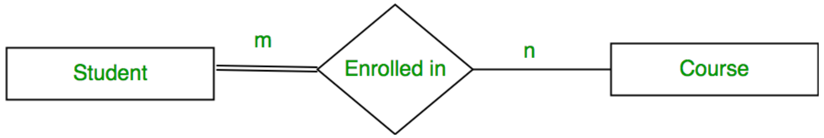
Total Participation –

- It specifies that each entity in the entity set must compulsorily participate in at least one relationship instance in that relationship set.
- That is why, it is also called as **mandatory participation.**
- Total participation is represented using a double line between the entity set and relationship set.



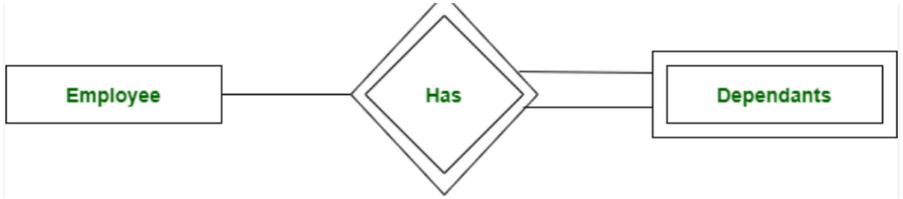
Partial Participation –

- It specifies that each entity in the entity set may or may not participate in the relationship instance in that relationship set.
- That is why, it is also called as **optional participation.**
- Partial participation is represented using a single line between the entity set and relationship set.



Weak Entity Type and Identifying Relationship:

weak entity type is represented by a double rectangle. The participation of weak entity type is always total. The relationship between weak entity type and its identifying strong entity type is called identifying relationship and it is represented by double diamond.



How to Create ERD(Designing concept on ERD)

- 1.Identification and listing of the Entities exists in the System
- 2.Identification and listing of relationships exists within the system
- 3.Identification and assignment of the cardinality exists in the system
- 4.Identification of the attributes exists in the system and assign them with proper symbol example primary key attribute,composite key attributes etc.
- 5.Create the ER diagram on the basis of above identifications.
- Example case:Create ERD for following case

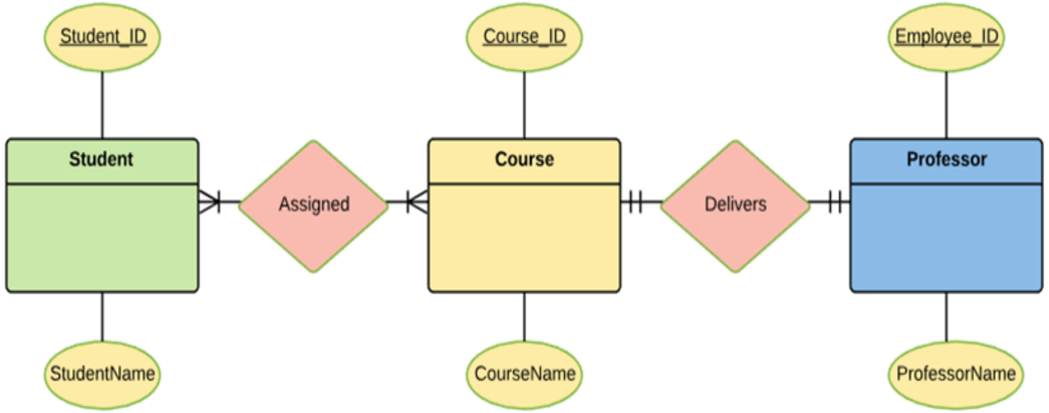
In a university, a **Student** enrolls in **Courses**.

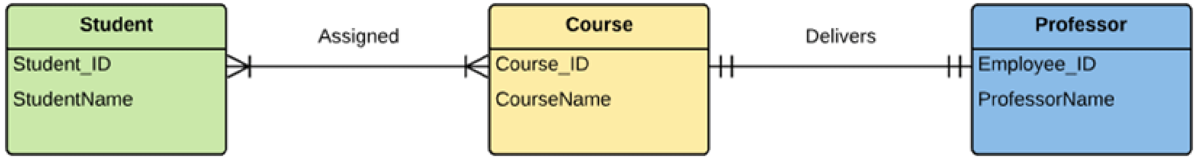
A student must be assigned to at least one or more Courses.

Each course is taught by a single P**rofessor**.

To maintain instruction quality, a Professor can deliver only one course

- **Solutions:**
- 1.List of Entities:Student,Course,Professor
- 2.List of Relationships:
  - student **assigned** to course.
  - A professor can **deliver** only one course.
- 3.Cardinalities:
  - A student can be assigned **multiple** courses
  - A Professor can deliver only **one** course
- 4.Attributes Identification: student,professor and Course can be characterized by their name,id etc.
- 5.Prepare the ERD



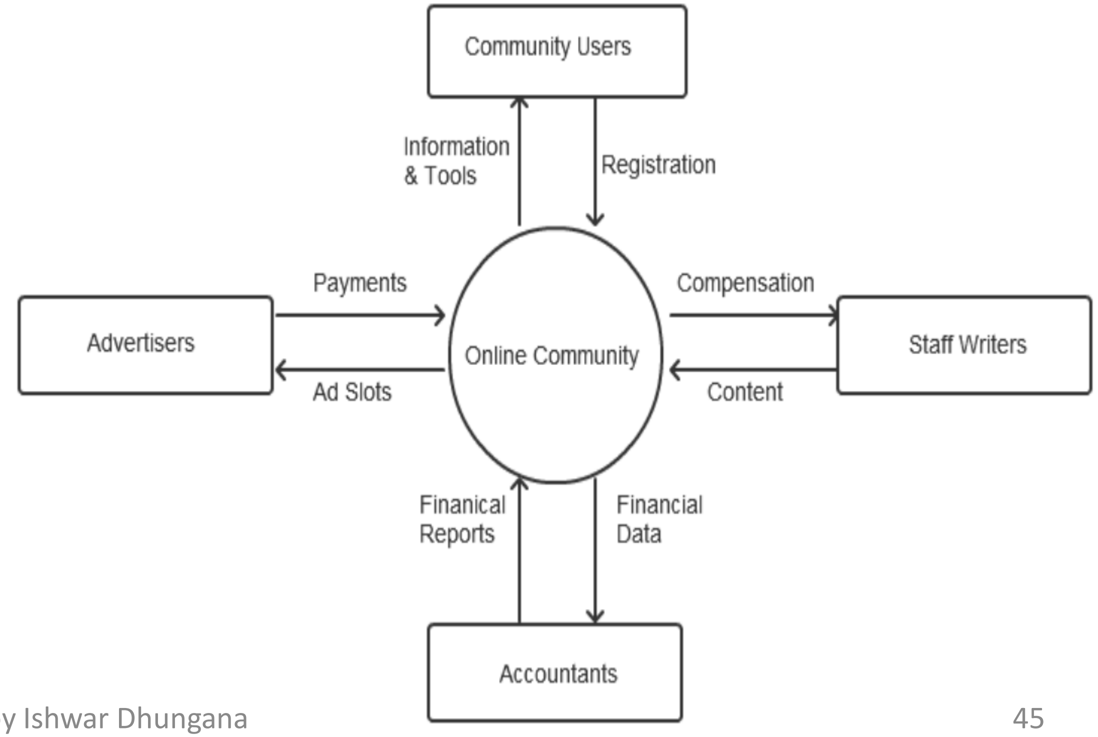


Context Diagram

- The context diagram is used to establish the context and boundaries of the system to be modelled: which things are inside and outside of the system being modelled, and what is the relationship of the system with these external entities. (systems, organizational groups, external data stores, etc.)
- A context diagram, sometimes called a level 0 data-flow diagram, is drawn in order to define and clarify the boundaries of the software system. It identifies the flows of information between the system and external entities. The entire software system is shown as a single process.
- A Context Diagram (and a DFD for that matter) provides no information about the timing, sequencing, or synchronization of processes such as which processes occur in sequence or in parallel. Therefore it should not be confused with a flowchart or process flow which can show these things.
- **Benefits:**
- Shows the scope and boundaries of a system at a glance including the other systems that interface with it
- No technical knowledge is assumed or required to understand the diagram
- Easy to draw and amend due to its limited notation
- Easy to expand by adding different levels of DFDs
- Can benefit a wide audience including stakeholders, business analyst, data analysts, developers
- **Drawback:**
- fail to give details about the sequence or timing of project processes.

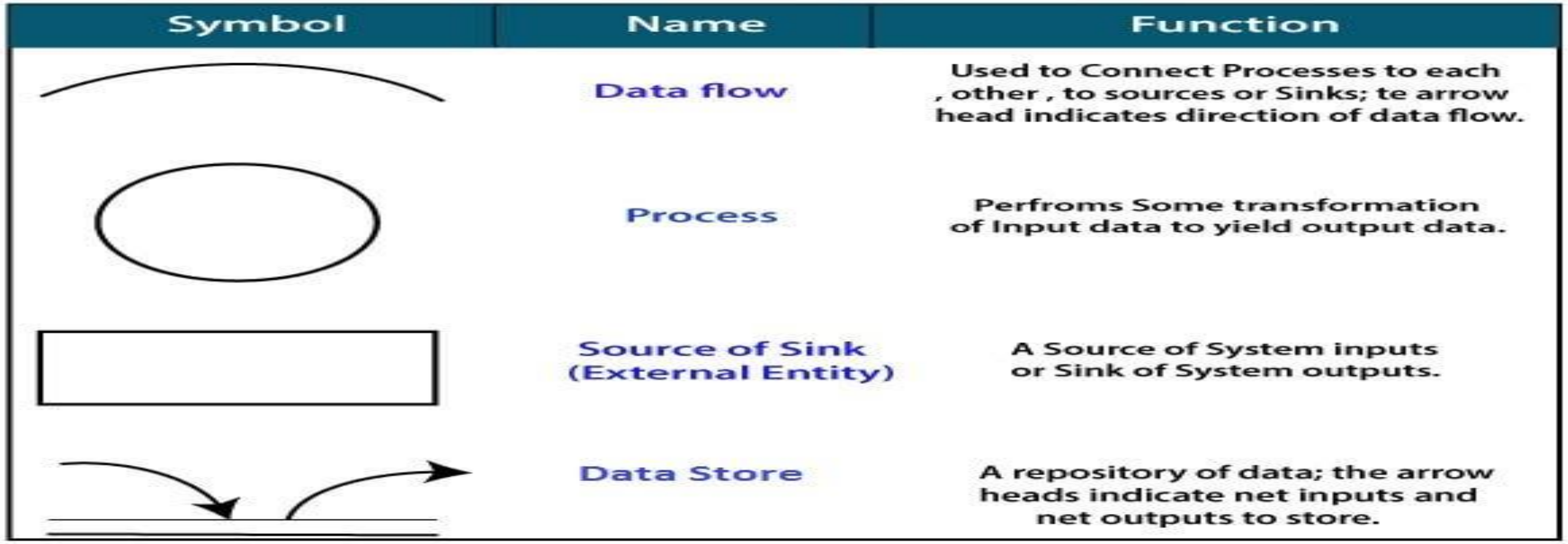
• How to construct Context diagram

- Identify data-flows by listing the major documents and information flows associated with the system
- Identify external entities by identifying sources and recipients of the data-flows, which lie outside of the system under investigation.
- Draw and label a process box representing the entire system.
- Draw and label the external entities around the outside of the process box.
- Add the data-flows between the external entities and the system box.
- Example:**online community**



Data Flow Diagram(DFD)

- **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. Specific operations depending on the type of data can be explained by a flowchart.
- The DFD belongs to structured-analysis modeling tools. Data Flow diagrams are very popular because they help us to visualize the major steps and data involved in software-system processes.DFDs depict logical data flow independent of technology but Flowcharts depict details of physical systems
- **Components of DFD**
  - **Process:** An activity or function performed for a specific business reason. Manual or computerized
  - **Data flow:** A single piece of data or a logical collection of data. Always starts or ends at a process
  - **Data Store:** A collection of data that is stored in some way. Data flowing out is retrieved from the data store. Data flowing in updates or is added to the data store
  - **External entity:** A person, organization, or system that is **external** to the system but interacts with it.



Symbol	Name	Function
⌒	Data flow	Used to Connect Processes to each , other , to sources or Sinks; te arrow head indicates direction of data flow.
○	Process	Perfroms Some transformation of Input data to yield output data.
▭	Source of Sink (External Entity)	A Source of System inputs or Sink of System outputs.
⇘═⇗	Data Store	A repository of data; the arrow heads indicate net inputs and net outputs to store.

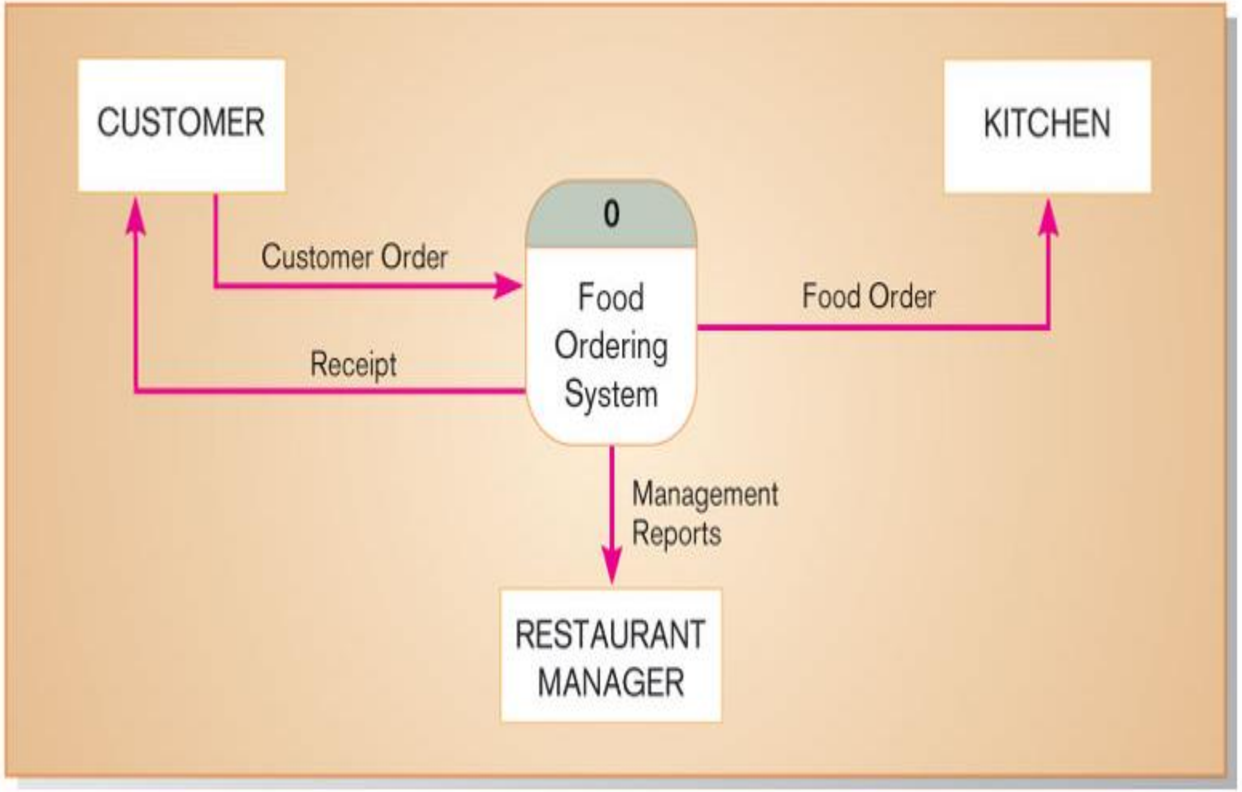
Symbols for Data Flow Diagrams

How to draw DFD?

- 1.Identify major inputs and outputs in your system
- 2.Build a context diagram
- 3.Expand the context diagram into a level 1 DFD
- 4.Expand to a level 2+ DFD
- 5.Confirm the accuracy of your final diagram
- **Some Additional Tips:**
  - Data flow from a process to a data store means update (insert, delete or change).
  - Data flow from a data store to a process means retrieve or use.
  - Data flow labels should be noun phrases.
  - Process labels should be verb phrases; data stores are represented by nouns
  - A data store must be associated to at least a process
  - An external entity must be associated to at least a process
  - Don't let it get too complex; normally 5 - 7 average people can manage processes
  - DFD is non-deterministic - The numbering does not necessarily indicate sequence, it's useful in identifying the processes when discussing with users
  - Data stores should not be connected to an external entity, otherwise, it would mean that you're giving an external entity direct access to your data files
  - Data flows should not exist between 2 external entities without going through a process
  - A process that has inputs but no outputs is considered to be a black-hole process

- **Functional Decomposition:**
  - An iterative process of breaking a system description down into finer and finer detail
  - High-level processes described in terms of lower-level sub-processes
  - DFD charts created for each level of detail
- DFD Levels
  - Context DFD
    - Overview of the organizational system
  - Level-1 DFD
    - Representation of system’s major processes at high level of abstraction
  - Level-2 DFD
    - Results from decomposition of Level 1 diagram
  - Level-*n* DFD
    - Results from decomposition of Level *n*-1 diagram

Example of DFD:Food Ordering System



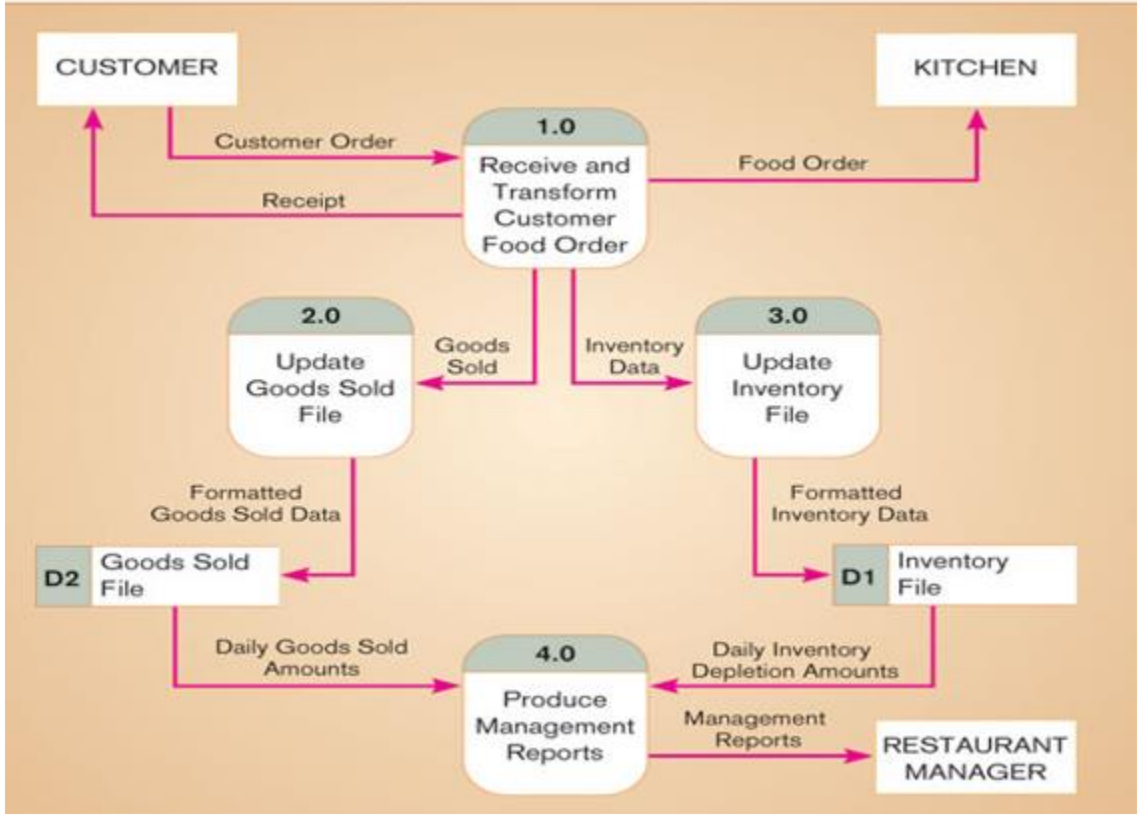
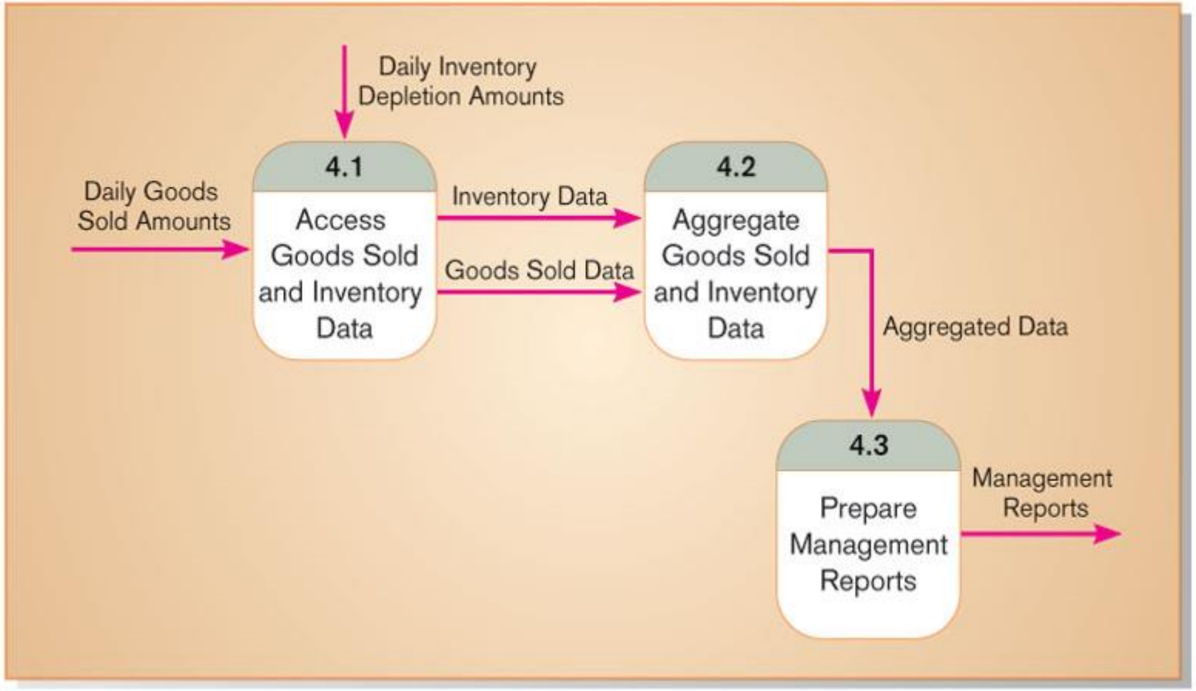


Fig:Context diagram of Food Ordering System
NOTE: only one process symbol, and no data stores shown.

Fig:Level-1 diagram of Food Ordering System
NOTE:Processes are labeled 1.0, 2.0, etc. These will be decomposed into more primitive (lower-level) DFDs.

Example of DFD:Food Ordering System


Daily Inventory Depletion Amounts
Daily Goods Sold Amounts
4.1
Access Goods Sold and Inventory Data
Inventory Data
Goods Sold Data
4.2
Aggregate Goods Sold and Inventory Data
Aggregated Data
4.3
Prepare Management Reports
Management Reports

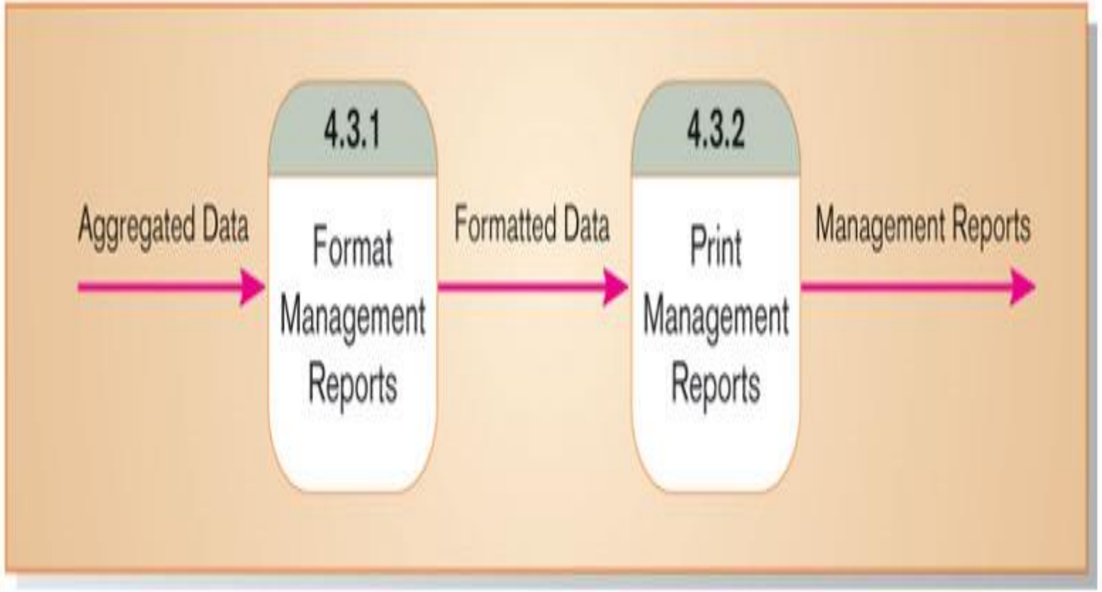


Fig:level-2 DFD for process 4.0 of food ordering system Processes are labeled 4.1, 4.2, etc. These can be further decomposed in more primitive (lower-level) DFDs if necessary.

Fig:Level-n diagram of Food Ordering System NOTE:Processes are labeled 4.3.1, 4.3.2, etc. If this is the lowest level of the hierarchy, it is called a *primitive DFD.*

User Interface Design

- Objectives of UID
  - ➢Understand a number of user interface design principles which have been introduced to serveral interaction styles and understand when these are most appropriate.
  - ➢Understand when to user graphical and textual persentation of information.
  - ➢Know what is involved in the principal activities in the user interface design process.
  - ➢Understand usability attributes and have been introduced to different approaches to interface evaluation.
- Designing effective interfaces for software systems
- System users often judge a system by its interface rather than its functionality
- A poorly designed interface can cause a user to make catastrophic errors
  - Poor user interface design is the reason why so many software systems are never used

UID Principles

- User familiarity
  - The interface should be based on user-oriented terms and concepts rather than computer concepts
  - E.g., an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.
- Consistency
  - The system should display an appropriate level of consistency
  - Commands and menus should have the same format, command punctuation should be similar, etc.
- Minimal surprise
  - If a command operates in a known way, the user should be able to predict the operation of comparable commands
- Recoverability
  - The system should provide some resilience to user errors and allow the user to recover from errors
  - This might include an undo facility, confirmation of destructive actions, 'soft' deletes, etc.
- User guidance
  - Some user guidance such as help systems, on-line manuals, etc. should be supplied
- User diversity
  - Interaction facilities for different types of user should be supported
  - E.g., some users have seeing difficulties and so larger text should be available

Human Computer Interactions(HCI)

The designer of a user interface to a computer is faced with two key questions:

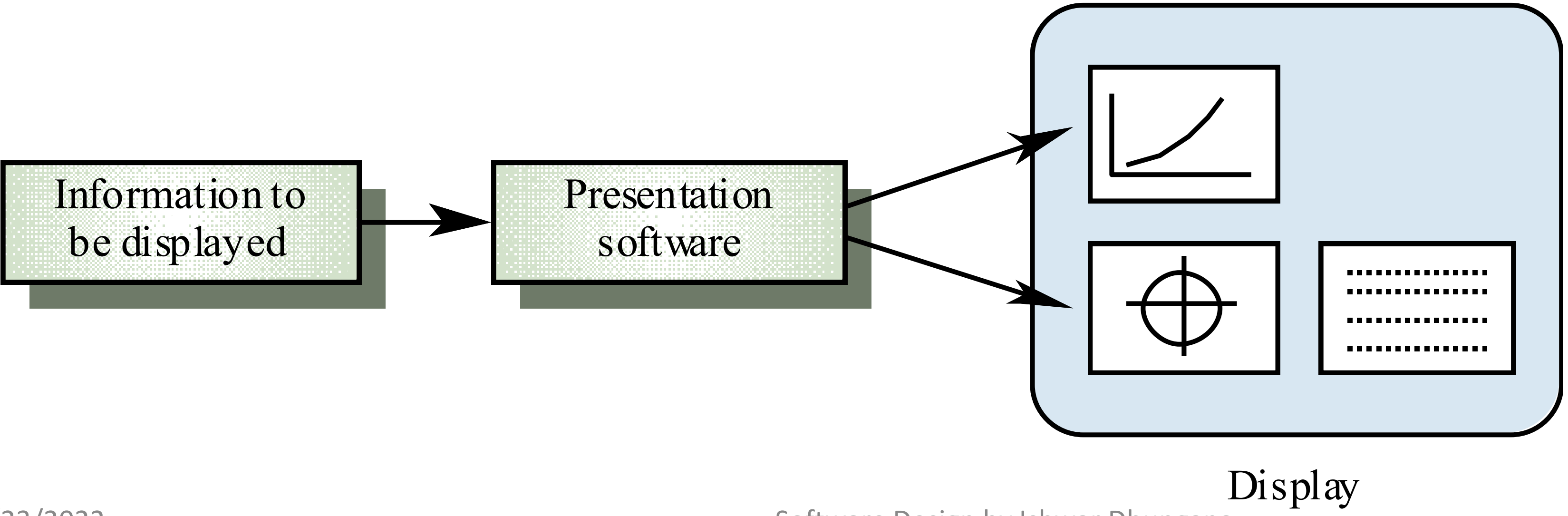
1. How should the user interact with the computer system?
2. How should information from the computer system be presented to the user?

Human Computer Interface (HCI) was previously known as the man-machine studies or man-machine interaction. It deals with the design, execution and assessment of computer systems and related phenomenon that are for human use.

- Areas where HCI can be implemented with distinctive importance are mentioned below
  - **Computer Science** – For application design and engineering.
  - **Psychology** – For application of theories and analytical purpose.
  - **Sociology** – For interaction between technology and organization.
  - **Industrial Design** – For interactive products like mobile phones, microwave oven, etc.

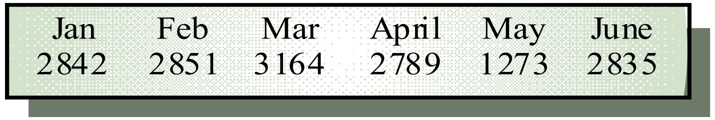
- Interaction styles
  - Direct manipulation
    - Easiest to grasp with immediate feedback
    - Difficult to program
  - Menu selection
    - User effort and errors minimized
    - Large numbers and combinations of choices a problem
  - Form fill-in
    - Ease of use, simple data entry
    - Tedious, takes a lot of screen space
  - Command language
    - Easy to program and process
    - Difficult to master for casual users
  - Natural language
    - Great for casual users
    - Tedious for expert users

- Information presentation
  - Information presentation is concerned with presenting system information to system users
  - The information may be presented directly or may be transformed in some way for presentation
  - The Model-View-Controller approach is a way of supporting multiple presentations of data
  - Information can be presented as:
    - Static information
      - Initialized at the beginning of a session. It does not change during the session
      - May be either numeric or textual
    - Dynamic information
      - Changes during a session and the changes must be communicated to the system user
      - May be either numeric or textual

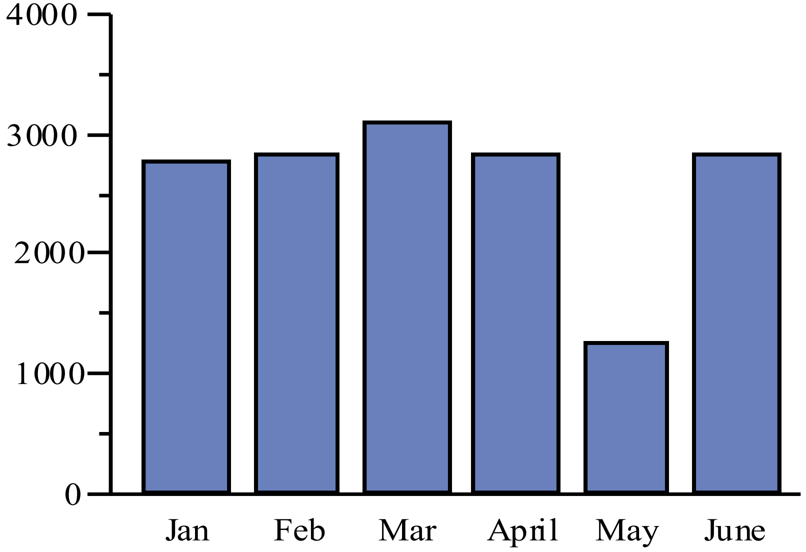


- Information Display Factors
  - Is the user interested in precise information or data relationships?
  - How quickly do information values change?
    Must the change be indicated immediately?
  - Must the user take some action in response to a change?
  - Is there a direct manipulation interface?
  - Is the information textual or numeric? Are relative values important?

Alternative Information Presentation



Jan	Feb	Mar	April	May	June
2842	2851	3164	2789	1273	2835



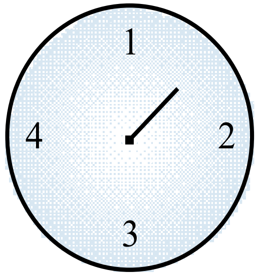
Digital presentation

- Compact - takes up little screen space
- Precise values can be communicated

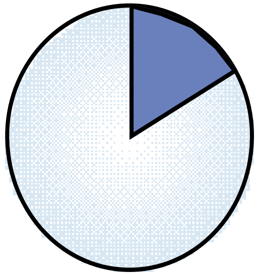
Analogue presentation

- Easier to get an 'at a glance' impression of a value
- Possible to show relative values
- Easier to see exceptional data values

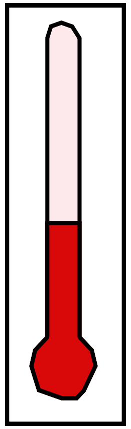
Information Display



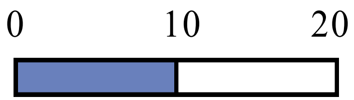
Dial with needle



Pie chart

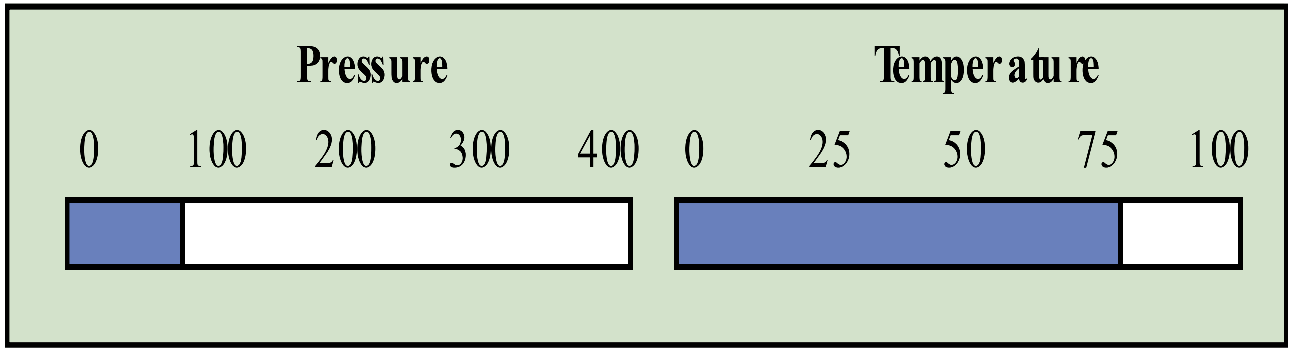


Thermometer



Horizontal bar

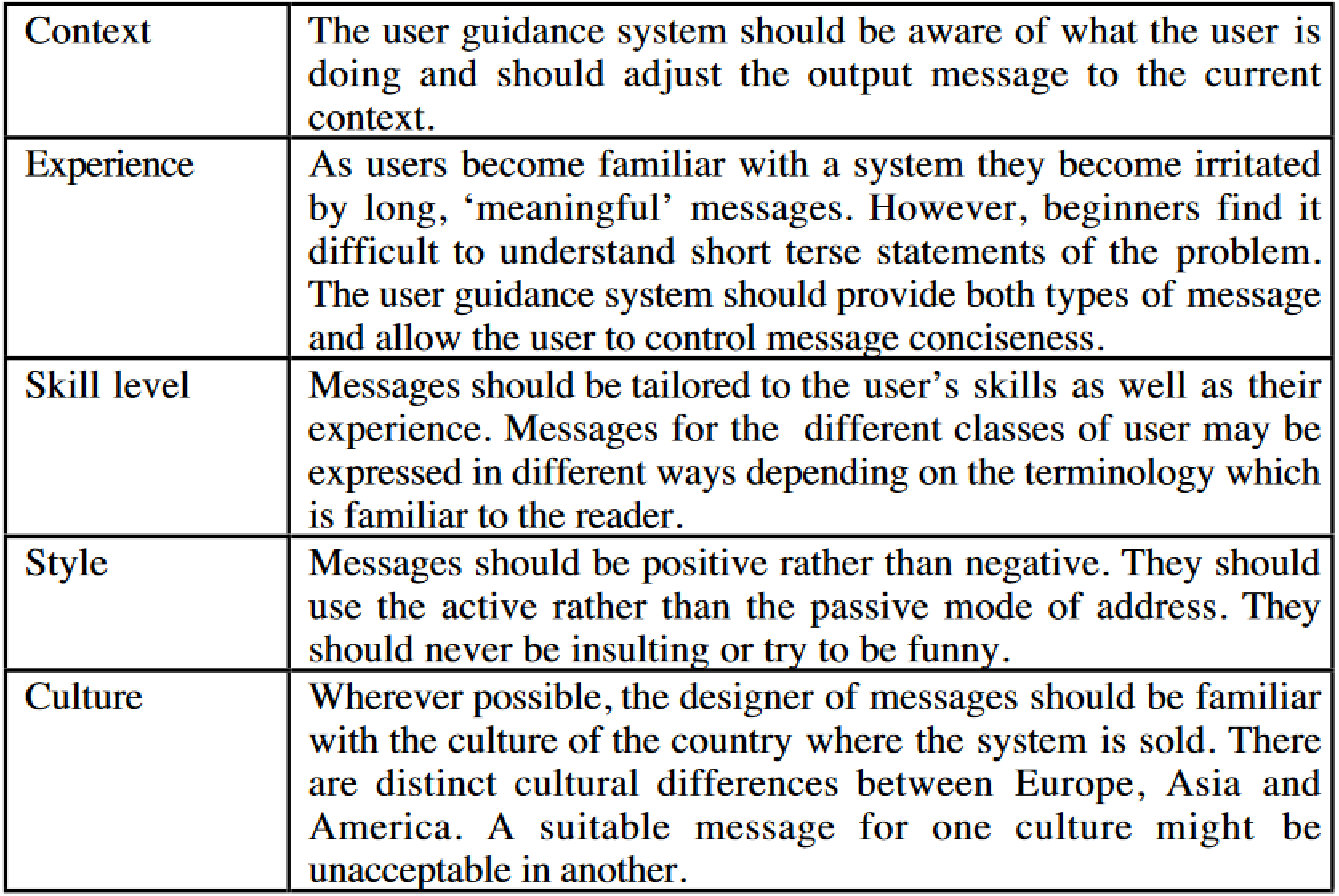
Displaying relative values



•Information Display Guidelines

- Exhibit only that information that is applicable to the present context.
- Don't burden the user with data, use a presentation layout that allows rapid integration of information.
- Use standard labels, standard abbreviations and probable colors.
- Permit the user to maintain visual context.
- Generate meaningful error messages.
- Use upper and lower case, indentation and text grouping to aid in understanding.
- Use windows (if available) to classify different types of information.
- Use analog displays to characterize information that is more easily integrated with this form of representation.
- Consider the available geography of the display screen and use it efficiently.

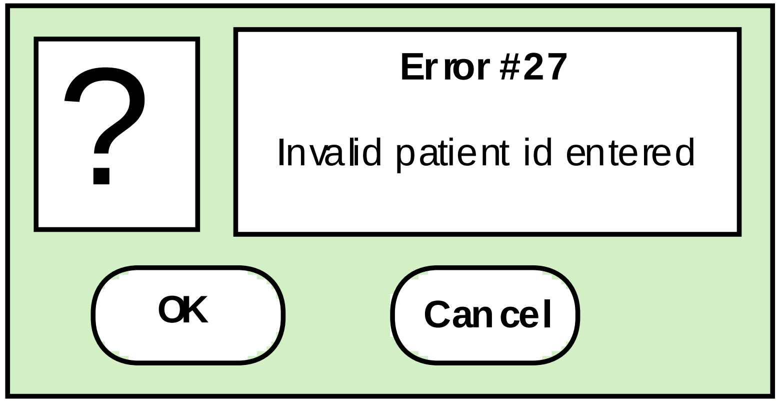
Design Factors in Message Wording



Context	The user guidance system should be aware of what the user is doing and should adjust the output message to the current context.
Experience	As users become familiar with a system they become irritated by long, 'meaningful' messages. However, beginners find it difficult to understand short terse statements of the problem. The user guidance system should provide both types of message and allow the user to control message conciseness.
Skill level	Messages should be tailored to the user's skills as well as their experience. Messages for the different classes of user may be expressed in different ways depending on the terminology which is familiar to the reader.
Style	Messages should be positive rather than negative. They should use the active rather than the passive mode of address. They should never be insulting or try to be funny.
Culture	Wherever possible, the designer of messages should be familiar with the culture of the country where the system is sold. There are distinct cultural differences between Europe, Asia and America. A suitable message for one culture might be unacceptable in another.

System and user-oriented error messages

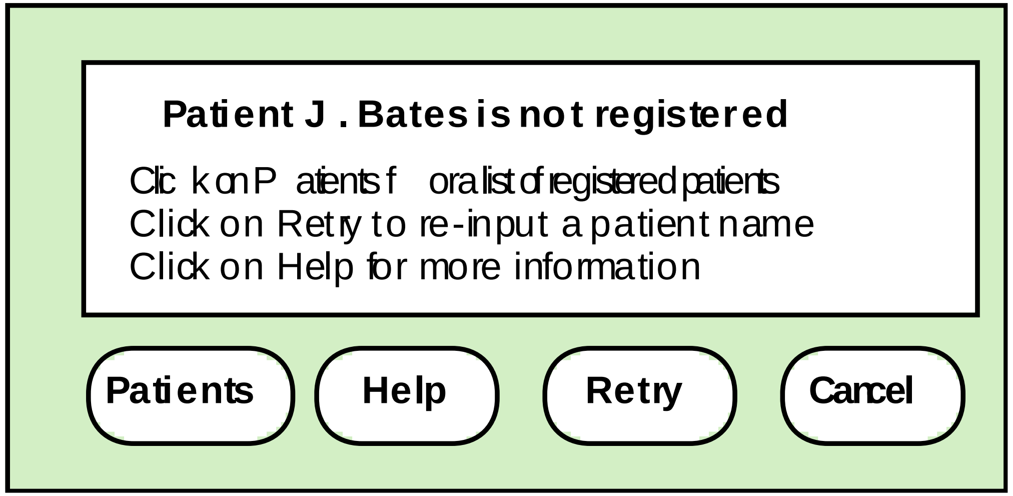
System-oriented error message



Error #27

Invalid patient id entered

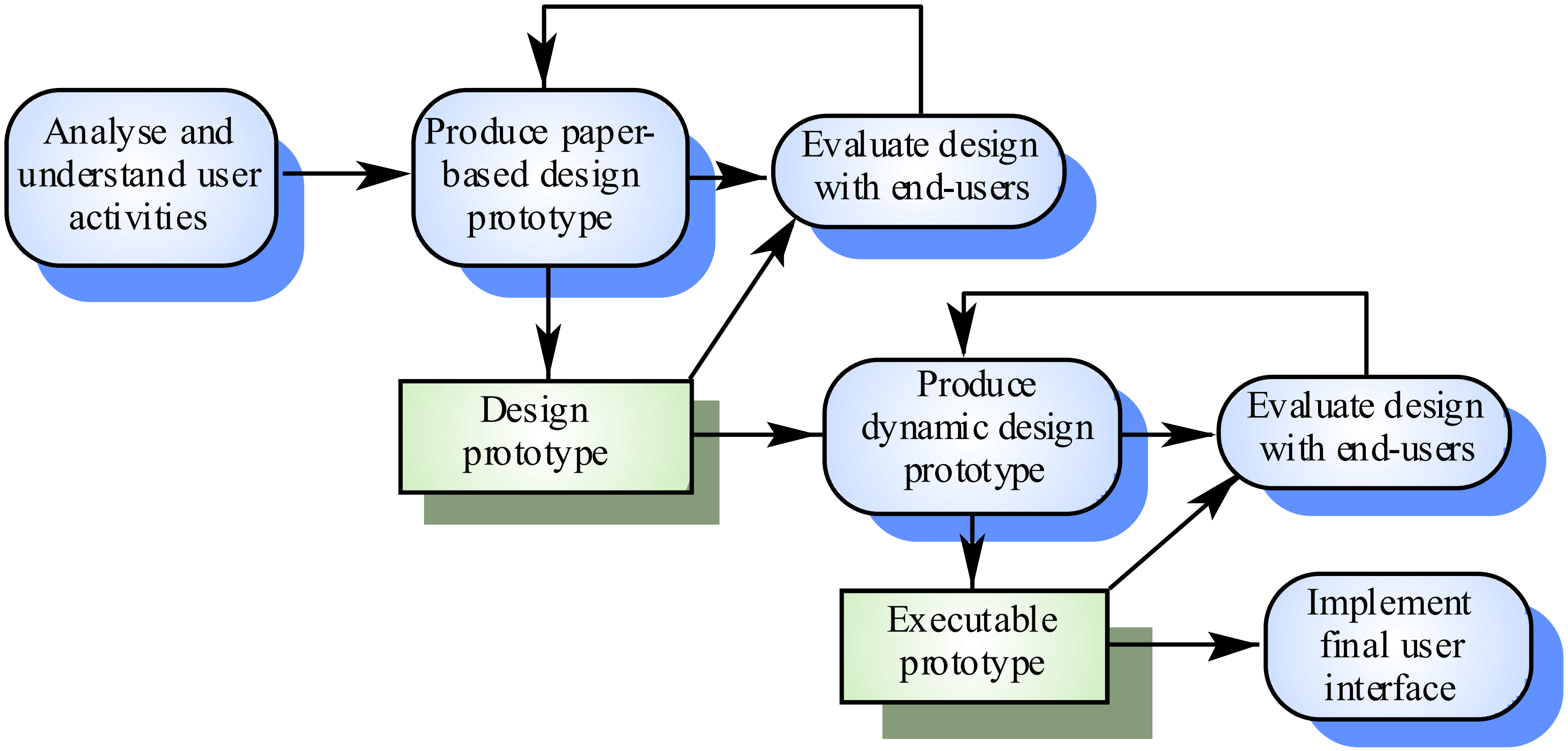
User-oriented error message



Patient J. Bates is not registered

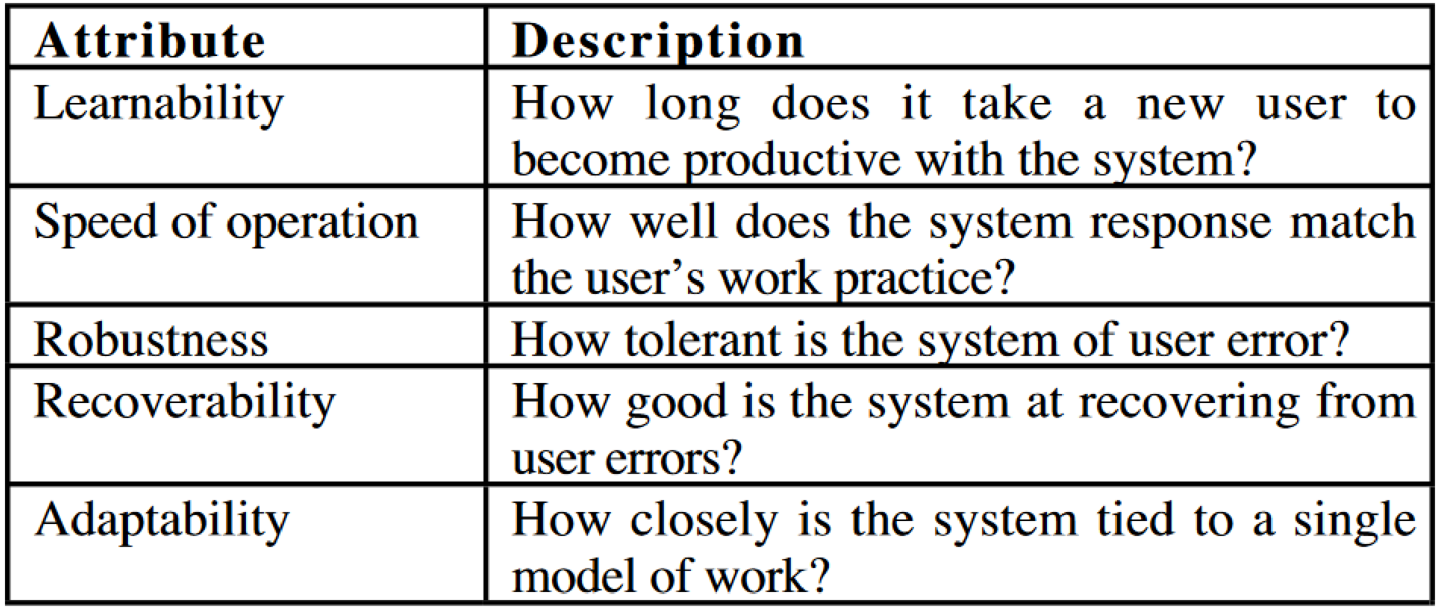
Click on Patients for a list of registered patients
Click on Retry to re-input a patient name
Click on Help for more information

User Interface Design Process


Analyse and understand user activities
Produce paper-based design prototype
Evaluate design with end-users
Design prototype
Produce dynamic design prototype
Evaluate design with end-users
Executable prototype
Implement final user interface

Interface Evaluation

- It is the process of assessing the usability of an interface and checking that it meets user requirements.
- Therefore, it should be part of the normal verification and validation process for software system.
- Some evaluation of a user interface design should be carried out to access its suitability.
- Full Scale evaluation is very expensive and impractical for most system.
- User Interface Evaluation Techniques
  - Questionnaires for user feedback
  - Video recording of system use and subsequent tape evaluation.
  - Instrumentation of code to collect information about facility use and user errors.
  - The provision of a “gripe” button for on-line user feedback
- **Various type of usability attributes:**



Attribute	Description
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user’s work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

Design Notations

- Design notation should lead to a procedural representation that is easy to understand and review. In addition, the notation should enhance "code to" ability so that code does, in fact, become a natural by-product of design.
- Finally, the design representation must be easily maintainable so that design always represents the program correctly.
- Many notations and languages exist to represent software design artifacts. Some are used mainly to describe a design’s structural organization, others to represent software behavior.
- Certain notations are used mostly during architectural design and others mainly during detailed design, although some notations can be used in both steps.
- In addition, some notations are used mostly in the context of specific. Here, they are categorized into notations for describing the structural (static) view vs. the behavioral (dynamic) view.

- **Structural Descriptions (static view)**- describe the major components and how they are interconnected (static view):
  - **Architecture description languages (ADLs**): textual, often formal, languages used to describe a software architecture in terms of components and connectors.
  - **Class and object diagrams**: used to represent a set of classes (and objects) and their interrelationships.
  - **Component diagrams**: used to represent a set of components ("physical and replaceable part[s] of a system that [conform] to and [provide] the realization of a set of interfaces") and their interrelationships.
  - **Class responsibility collaborator cards (CRCs):** used to denote the names of components (class), their responsibilities, and their collaborating components' names.
  - **Deployment diagrams**: used to represent a set of (physical) nodes and their interrelationships, and, thus, to model the physical aspects of a system.
  - **Entity-relationship diagrams (ERDs**): used to represent conceptual models of data stored in information systems.
  - **Interface description languages (IDLs)**: programming-like languages used to define the interfaces (names and types of exported operations) of software components.
  - **Structure charts**: used to describe the calling structure of programs (which module calls, and is called by, which other module).
- **Behavioral Descriptions (dynamic view):** describe the dynamic behavior of software and components.
  - **Activity diagrams**: used to show the control flow from activity ("ongoing non-atomic execution within a state machine") to activity.
  - **Collaboration diagrams**: used to show the interactions that occur among a group of objects, where the emphasis is on the objects, their links, and the messages they exchange on these links.
  - **Data flow diagrams (DFDs):** used to show data flow among a set of processes.
  - **Decision tables and diagrams**: used to represent complex combinations of conditions and actions.
  - **Flowcharts and structured flowcharts**: used to represent the flow of control and the associated actions to be performed.
  - **Sequence diagrams**: used to show the interactions among a group of objects, with emphasis on the time-ordering of messages.
  - **State transition and state-chart diagrams**: used to show the control flow from state to state in a state machine.
  - **Formal specification languages**: textual languages that use basic notions from mathematics (for example, logic, set, sequence) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and post-conditions.
  - **Pseudocode and program design languages (PDLs):** structured-programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method.