UNIT III SOFTWARE DESIGN

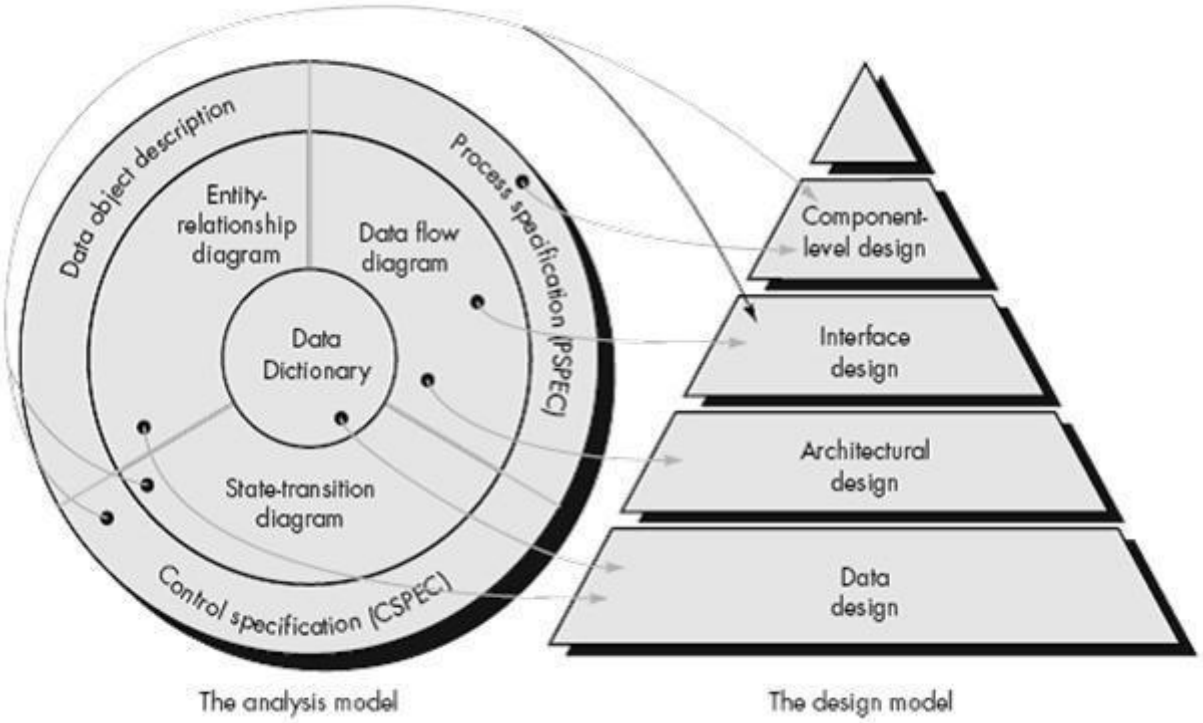
Design process: Design Concepts-Design Model, Design Heuristic. **Architectural Design**: Architectural styles, Architectural Design, Architectural Mapping using Data Flow. **User Interface Design:** Interface analysis, Interface Design. **Component level Design:** Designing Class based components, traditional Components.

3. Design process:

Software design:

Software design is model of software which translates the requirements into finished software products in which the details about software data structures, architectures, interfaces and components that are necessary to implement the system are given.

Levels of design:



Software Quality Guidelines and Attributes(characteristics of good design):

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

Quality Guidelines

a. A design should exhibit an architecture that
   - Has been created using recognizable architectural styles or patterns,
   - Is composed of components that exhibit good design characteristics,
   - Can be implemented in an evolutionary fashion.

b. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

c. A design should contain distinct representations of data, architecture, interfaces, and components.

d. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
e. A design should lead to components that exhibit independent functional characteristics.
f. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
g. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
h. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes

- Functionality
- Usability
- Reliability
- Performance
- Supportability

3.1 Design Concepts

The software Design Concepts provides a framework for implementing the right software.

1.Abstraction
2.Modularity
3.Architecture
4.Refinement
5.Pattern
6.Information hiding
7.Functional independence
8.Refactoring
9.Design Class

1. Abstraction:

- When you consider a modular solution to any problem, many levels of abstraction can be posed.
- Highest level of abstraction, a solution is stated in broad terms.
- Lower levels of abstraction, a more detailed description of the solution is provided.
- Lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

Two types:

- ✓ *Procedural abstraction*
- ✓ *Data abstraction*

Procedural abstraction refers to a sequence of instructions that have a specific and limited function.
Data abstraction is a named collection of data that describes a data object.

2. Architecture:

- Architecture is the structure or organization of program components (Modules), the manner in which these components interact, and the structure of data that are used by the components.

3. Patterns:

- A design pattern describes a design structure that solves a particular design problem within a
- **specific context and amid forces that may have** an impact on the manner in which the pattern is applied and used.
- The intent of each design pattern is to provide a description that enables a designer to determine

- ✓ Whether the pattern is applicable to the current work,
- ✓ Whether the pattern can be reused (hence, saving design time), and
- ✓ Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. Modularity:

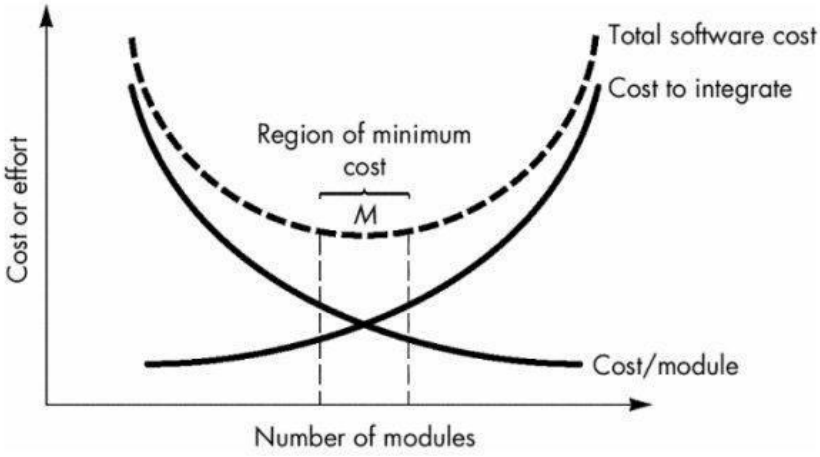
- Modularity is the most common example of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules that* are integrated to satisfy problem requirements.
- It has been stated that modularity is the single attribute of software that allows a program to be intellectually manageable
- Follows "divide and conquer" concept, a complex problem is broken down into several manageable pieces

Let p1 and p2 be two problems.
Let E1 and E2 be the effort required to solve them,
If c(p1)>c(p2)
Hence E(p1)>E(p2)
Now, Complexity of a problem that combines p1 and p2 is greater than complexity when each problem is consider

$$C(p1+p2) > C(p1)+C(p2)$$

Hence, $E(p1+p2) > E(p1)+E(p2)$

It is easier to solve a complex problem when you break it into manageable pieces



- If an error occurs within a module then those errors are localized and not spread to other modules.
- The 5 criteria to evaluate a design method with respect to its modularity,

✓ **Modular understandability**

Module should be understandable as a standalone unit (no need to refer to other modules)

✓ **Modular continuity**

If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of side effects will be minimized.

✓ **Modular protection**

If an error occurs within a module then those errors are localized and not spread to other modules.

✓ **Modular Composability**

Design method should enable reuse of existing components.

✓ **Modular Decomposability**

Complexity of the overall problem can be reduced if the design method provides a systematic mechanism to decompose a problem into sub problems.

5.Information Hiding:

- Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
- The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance.
- Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

6.Functional Independence:

- The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.
- You should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure. **Why independence is important?**
- Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified.
- Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.
- Functional independence is a key to good design, and design is the key to software quality.
- Independence is assessed using two qualitative criteria:
  - cohesion
  - coupling

Cohesion is an indication of the relative functional strength of a module.
Coupling is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding.A cohesive module performs a single task, requiring little interaction with other components in other parts of a program.

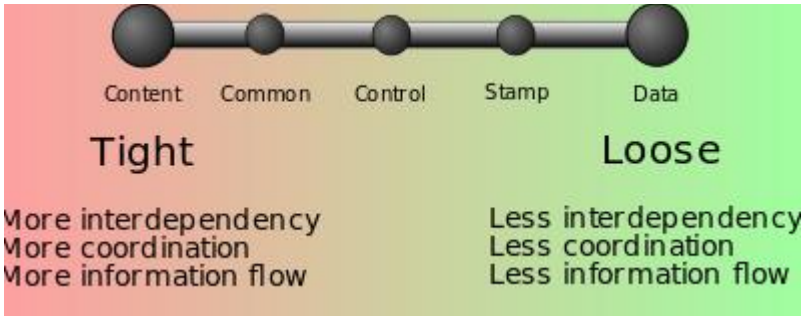
Types of Cohesion:

- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.
- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

- **Sequential cohesion** - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion** - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Types of Coupling:



- **Content coupling (high)**

Content coupling (also known as pathological coupling) occurs when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module). In this situation, a change in the way the second module produces data (location, type, timing) might also require a change in the dependent module.

- **Common coupling**

Common coupling (also known as global coupling) occurs when two modules share the same global data (e.g., a global variable). Changing the shared resource might imply changing all the modules using it.

- **External coupling**

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.

- **Control coupling**

Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

- **Stamp coupling (data-structured coupling)**

Stamp coupling occurs when modules share a composite data structure and use only parts of it, possibly different parts (e.g., passing a whole record to a function that only needs one field of it).In this situation, a modification in a field that a module does not need may lead to changing the way the module reads the record.

- **Data coupling**

Data coupling occurs when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

- **Message coupling (low)**

This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing.

7. Refinement:

- Refinement is actually a process of *elaboration.* You begin with a statement of function (or) description
- Refinement helps you to reveal low-level details as design progresses

8. Refactoring

- *Refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.
- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

9.Design Classes:

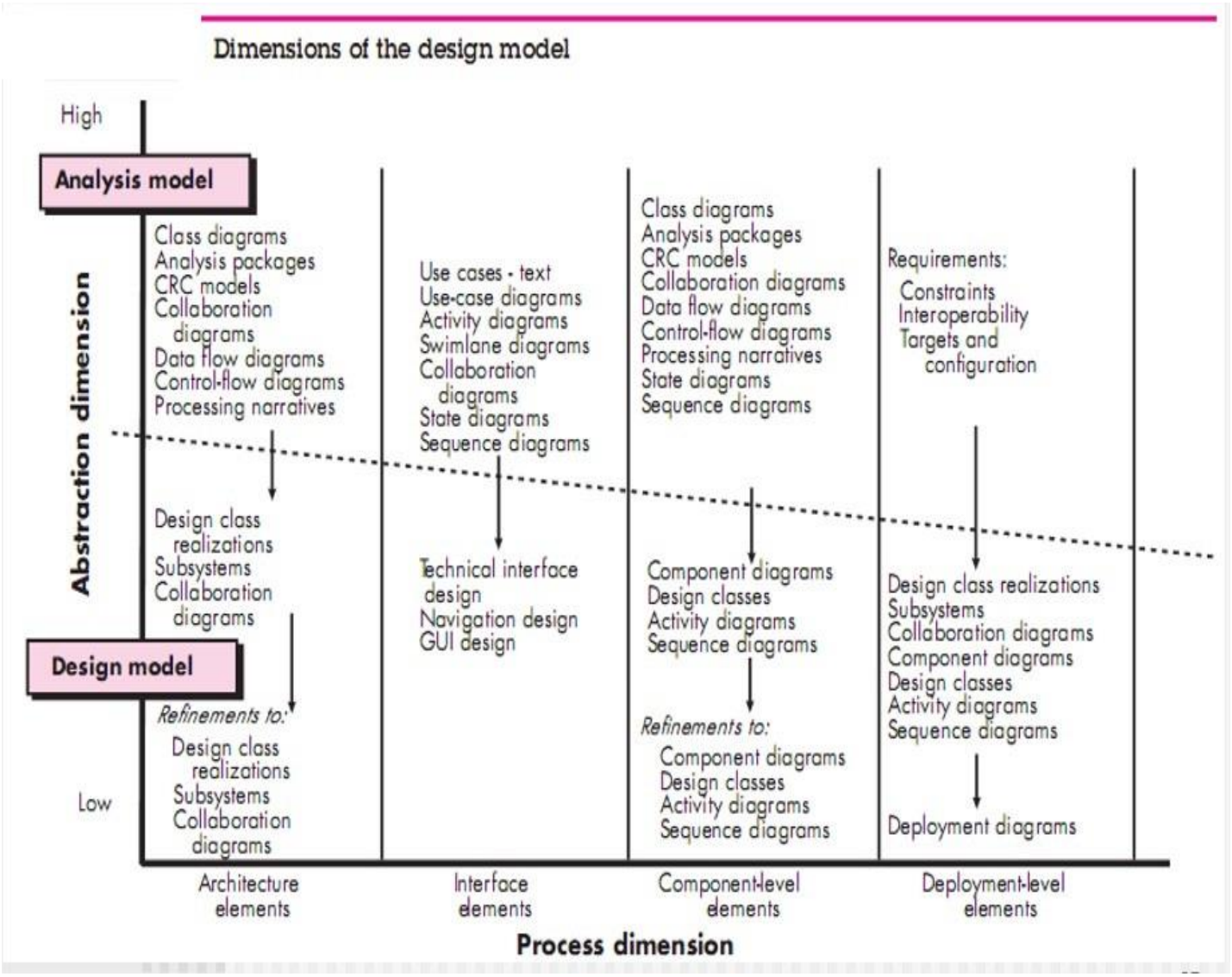
- As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- Five different types of design classes, each representing a different layer of the design architecture
  - ***User interface classes*** define all abstractions that are necessary for human computer interaction (HCI). In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.
  - ***Business domain classes*** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
  - ***Process classes*** implement lower-level business abstractions required to fully manage the business domain classes.
  - ***Persistent classes*** represent data stores (e.g., a database) that will persist beyond the execution of the software.
  - ***System classes*** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

They define four characteristics of a well-formed design class:

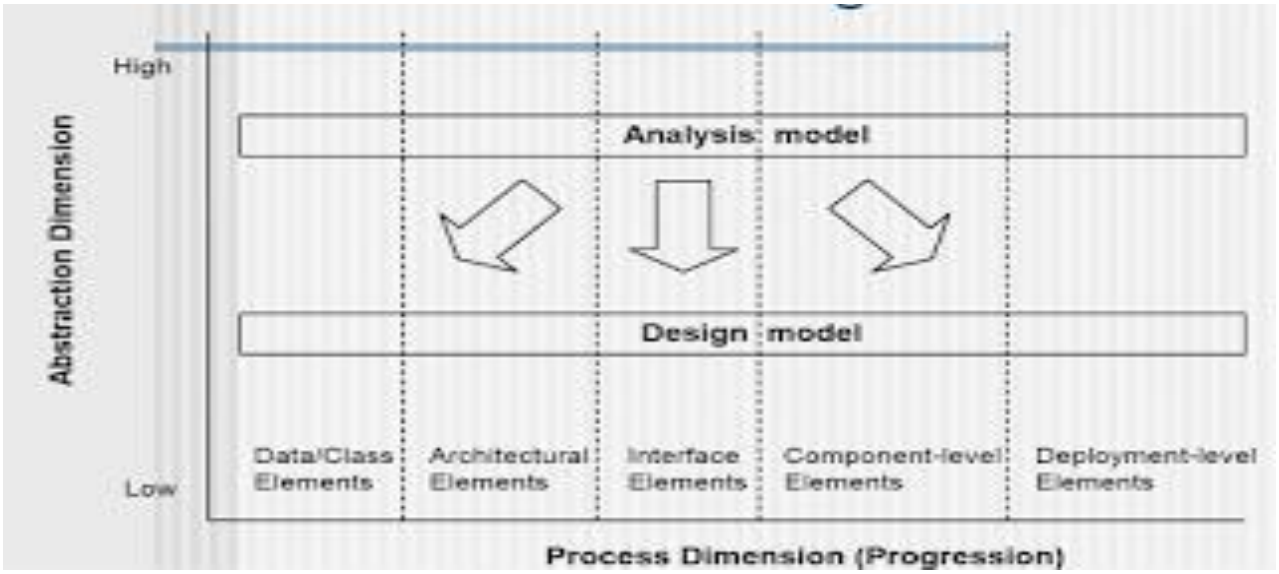
- ✓ Complete and sufficient.
- ✓ Primitiveness.
- ✓ High cohesion
- ✓ Low coupling

3.2 DESIGN MODEL

- The design model can be viewed in two different dimensions as illustrated in two different dimensions.
  - process dimension
  - abstraction dimension



- ✓ **The *process dimension*** indicates the evolution of the design model as design tasks are executed as part of the software process.
- ✓ **The *abstraction dimension*** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.



❖ **Data Design Elements**

- ✓ Customer's/ User's View:Data Architecting means it creates a model of data that is represented at a high level of abstraction i.e. Build Architecture of Data.
- ✓ Program Component Level: The design of Data structure & algorithms.
- ✓ Application Level: Translate Data Model into a database.
- ✓ Business Level: Data warehouse(Reporting & Analysis of DB) & Data mining(Analysis).
- ✓ At last it means creation of Data Dictionary.

❖ **Architectural Design Elements**

- ✓ Provides an overall view of the software product(Similar like Floor Plan of house)
- ✓ The architectural model is derived from three sources:
  - ➢ Information about the application domain for the software to be built;
  - ➢ Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand;
  - ➢ The availability of architectural styles (Chapter 9) and patterns
- ✓ Difference: An architectural style is a conceptual way of how the system will be created / will work.
- ✓ An architectural pattern describes a solution for implementing a style at the level of subsystems or modules and their relationships.

❖ **Interface Design Elements**

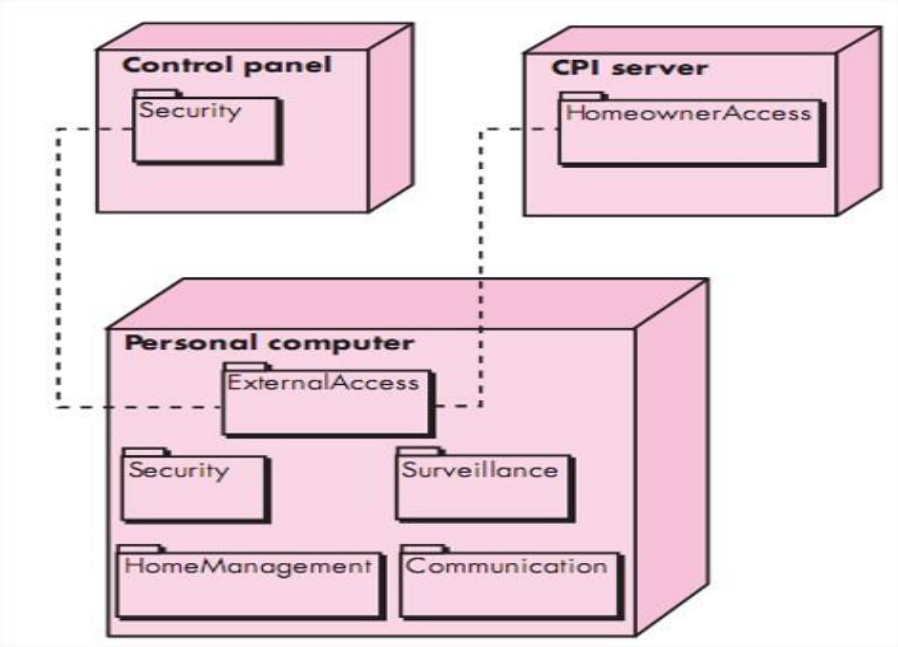
- ✓ The interface design elements for software represent information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
- ✓ There are three important elements of interface design:
  - ➢ The user interface (UI);
  - ➢ External interfaces to other systems, devices, networks, or other producers or consumers of information;
  - ➢ Internal interfaces between various design components.

❖ **Component-Level Design Elements**

- ✓ The component-level design for software fully describes the internal detail of each software component.

❖ **Deployment-Level Design Elements**

- ✓ Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.



3.3 DESIGN HEURISTICS:

1. Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.

Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence. An exploded module becomes two or more modules in the final program structure.

An imploded module is the result of combining the processing implied by two or more modules. An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

2. Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.

The structure shown inside the cloud in figure does not make effective use of factoring. All modules **are pancaked below a single control module. In general, a more reasonable distribution of control is** shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

3. Keep the scope of effect of a module within the scope of control of that module.

The scope of effect of module e is defined as all other modules that are affected by a decision made in module e. The scope of control of module e is all modules that are subordinate and ultimately subordinate to module e. Referring to figure, if module e makes a decision that affects module r, we have a violation of this heuristic, because module r lies outside the scope of control of module e.

4. Evaluate module interfaces to reduce complexity and redundancy and improve consistency. Module interface complexity is a prime cause of software errors .Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e., seemingly unrelated data passed via an argument list or other technique) is an indication of low cohesion. The module in question should be reevaluated.

5. Define modules whose function is predictable, but avoid modules that are overly restrictive.

A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details. Modules that have internal "memory" can be unpredictable unless care is taken in their use.

6.Strive for controlled entry modules by avoiding "pathological connections."

This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled. Pathological connection refers to branches or references into the middle of a module.

3.4 ARCHITECTURAL DESIGN

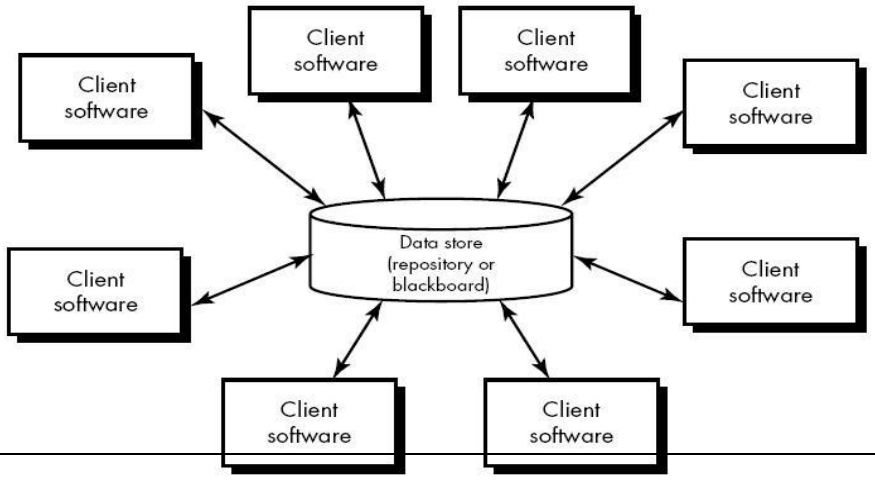
- Architectural styles
  - Five styles
- Architectural Designing
  - Representing the System in Context
  - Defining Archetypes
  - Refining the Architecture into Components
- Architectural Mapping using Data Flow
  - Transform mapping
  - Transaction mapping

3.4.1 Architectural styles

- **Data-centered architectures.**
- **Data-flow architectures.**
- **Call and return architectures**
- **Object-oriented architectures**
- **Layered architectures**

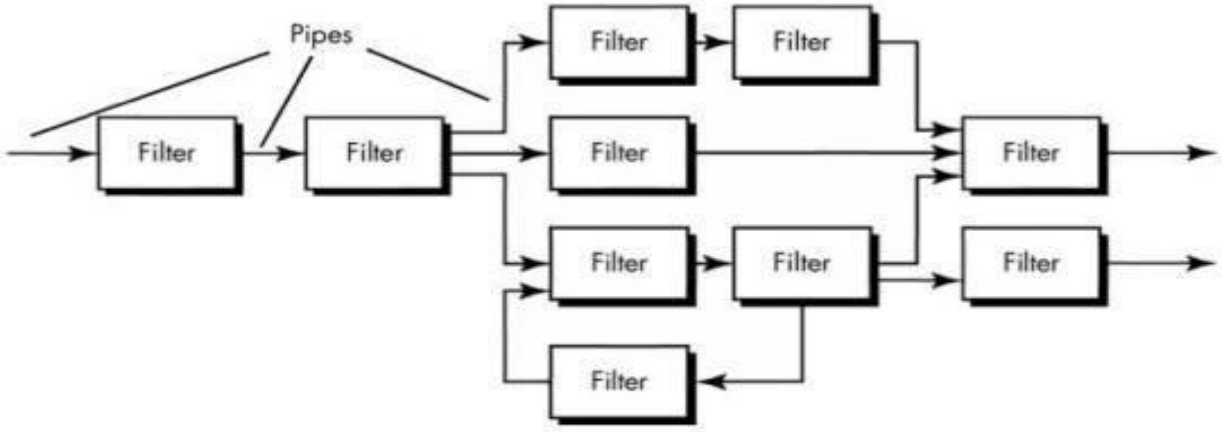
(i) Data-centered architectures:

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Client software accesses a central repository.
- In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software.
- **A variation on this approach transforms the repository into a blackboard that sends** notifications to client software when data of interest to the client change.
- Data-centered architectures promote inerrability. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently).
- Client components independently execute process.
- In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients).



(ii) Data-flow architectures:

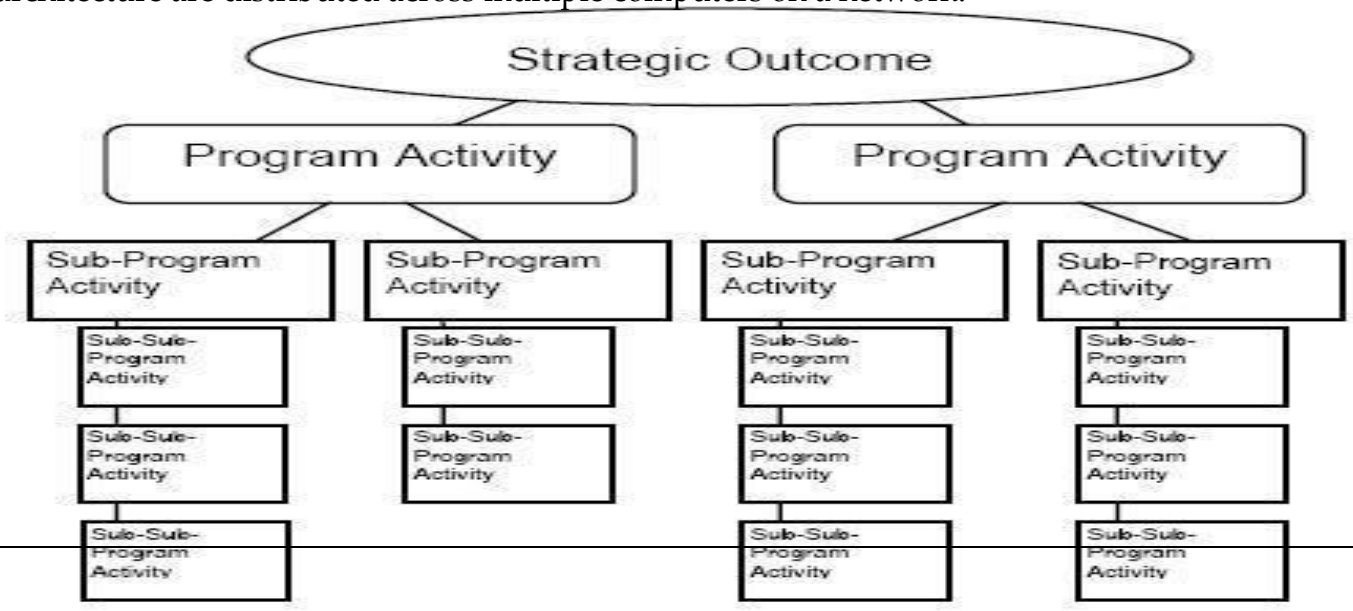
- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A pipe and filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- However, the filter does not require knowledge of the working of its neighboring filters.
- If the data flow degenerates into a single line of transforms, it is termed batch sequential. This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it.



(a) Pipes and filters

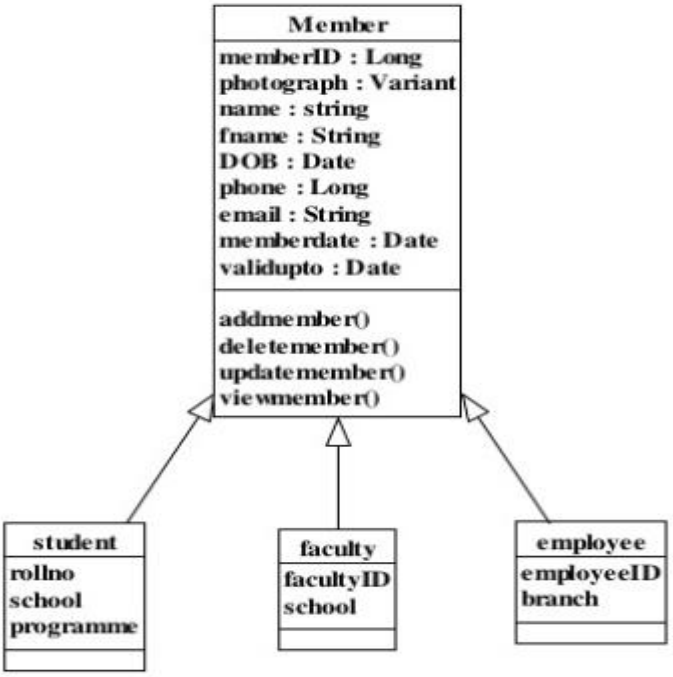
(iii) Call and return architectures:

- This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.
- A number of sub styles exist within this category:
    a. Main program/subprogram architectures
    b. Remote procedure call architectures
- Main program/subprogram architectures: This classic program structure decomposes function **into a control hierarchy where a main program invokes a number of program components,** which in turn may invoke still other components.
- Remote procedure calls architectures: The components of a main program/ subprogram architecture are distributed across multiple computers on a network.



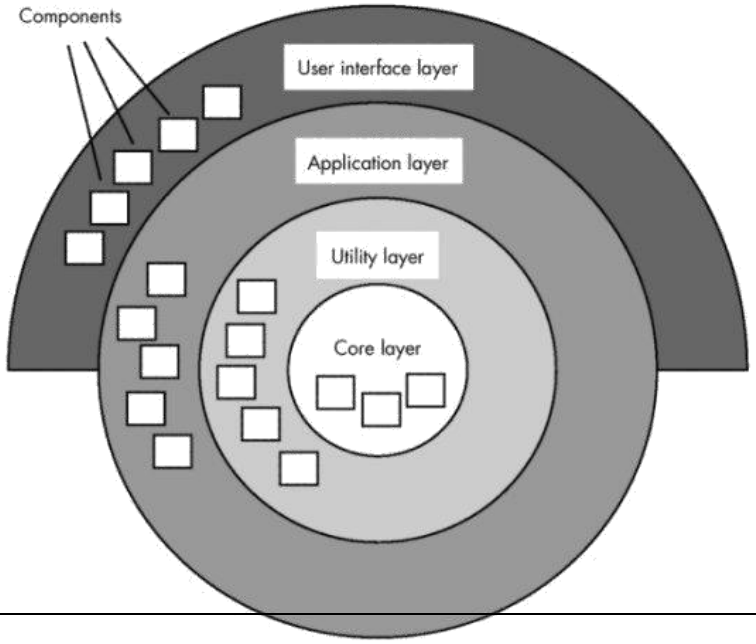
(iv) Object-oriented architectures:

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



(v) Layered architectures:

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.
- These architectural styles are only a small subset of those available to the software designer.
- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen.
- In many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

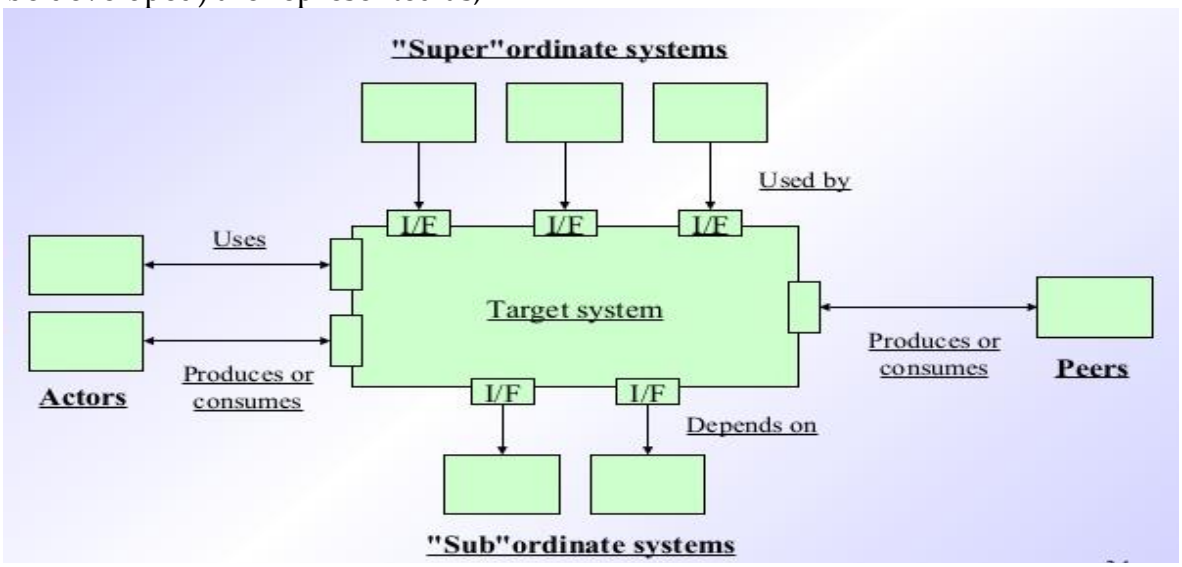


3.4.2 Architectural Designing:

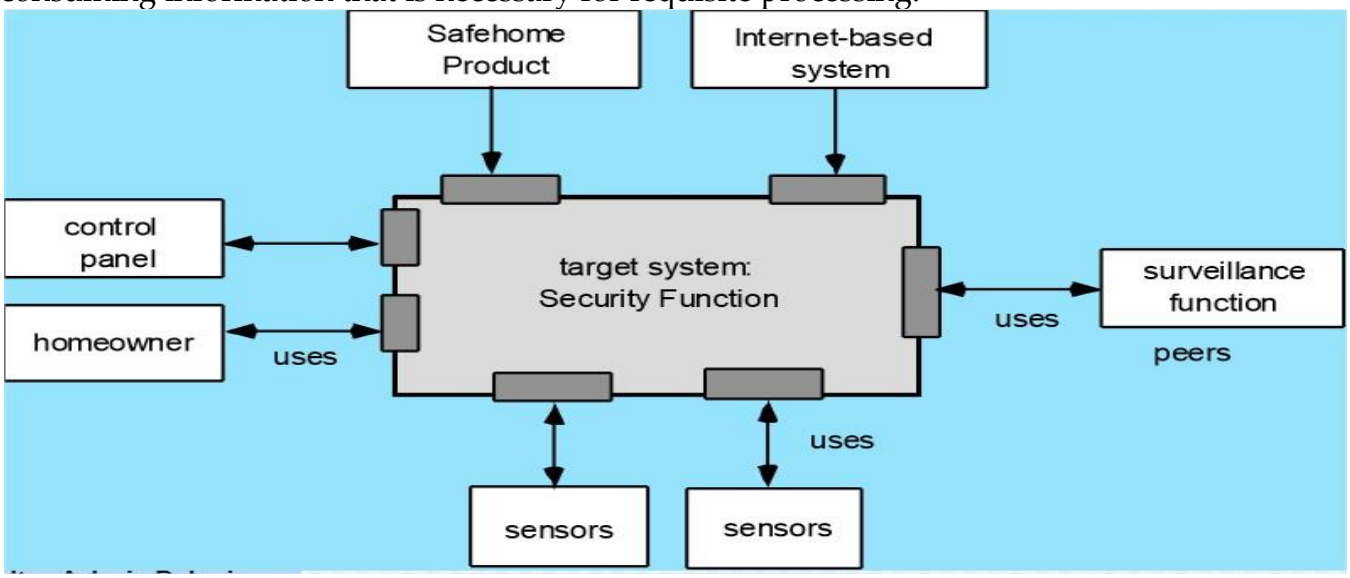
As architectural design begins, the software to be developed must be put into context that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.

Representing the System in Context

- At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries.
- Systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as,



- *Super ordinate systems* - those systems that use the target system as part of some higher-level processing scheme.
- *Subordinate systems* - those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- *Peer-level systems* - those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
- *Actors* - entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.



Defining Archetypes

- An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of architecture for the target system.
- In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.
  - **Node:** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.
  - **Detector:** An abstraction that encompasses all sensing equipment that feeds information into the target system.
  - **Indicator:** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
  - **Controller:** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

Refining the Architecture into Components

- To create full structure of the system it is required to refine the software architecture into components.
- The data flow diagram is drawn from which the specialized components can be identified. Such components are the components that process the data flow across the interfaces.
- The components can be the entities that follow following functionalities

➢ ***External communication management*** **-** coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

➢***Control panel processing*** **-** manages all control panel functionality.

➢***Detector management*** **-** coordinates access to all detectors attached to the system.

➢***Alarm processing*** **-** verifies and acts on all alarm conditions.

3.4.3 Architectural Mapping Using Data Flow

Transform flow:

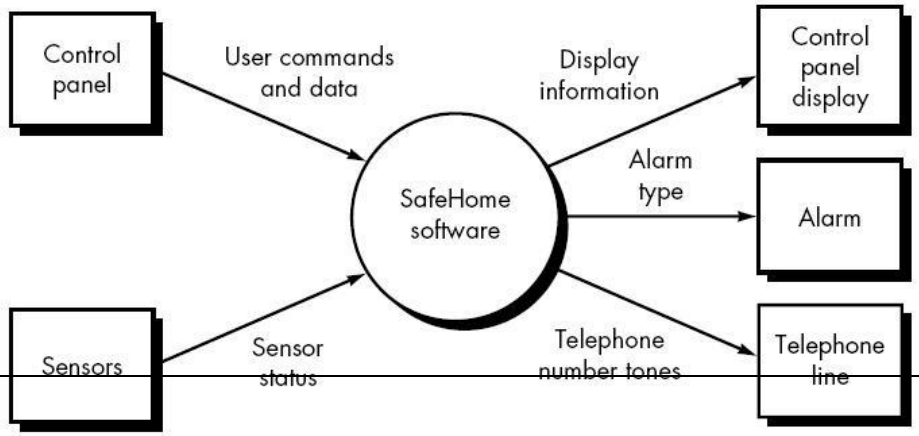
A transform flow is a sequence of paths which forms transition in which input data are transformed into output data.

Transaction flow:

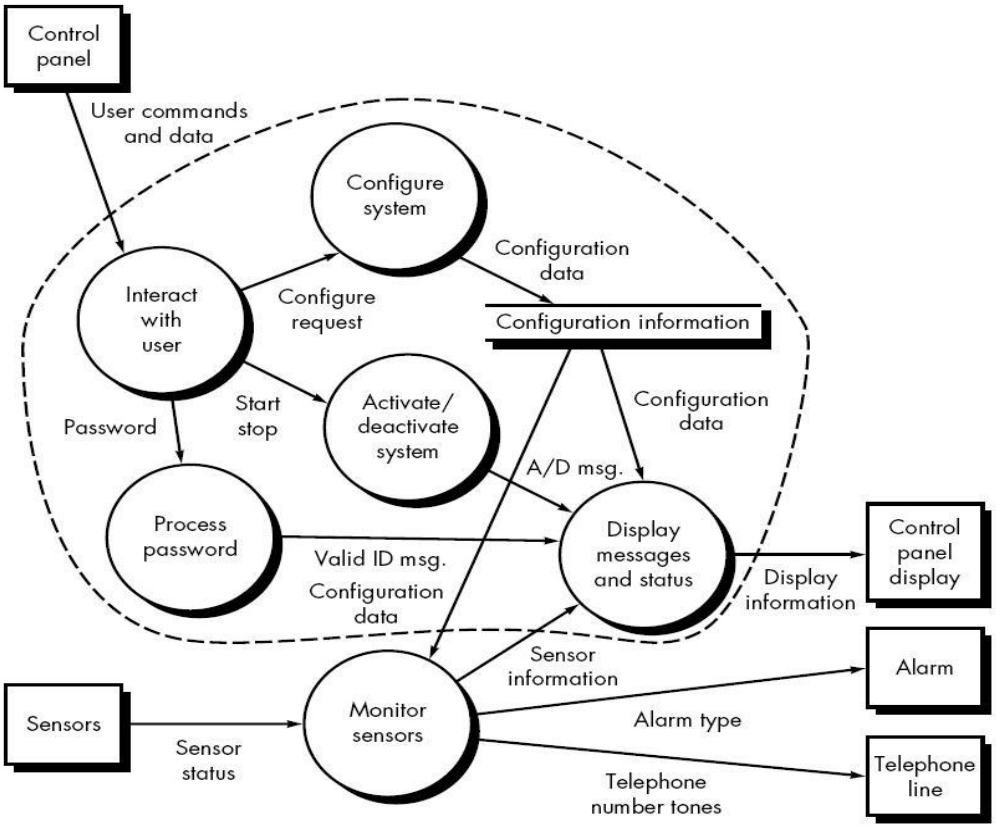
A transaction flow represents the information flow in which single data item triggers the overall information flow along the multiple paths.

(i) Transform Mapping

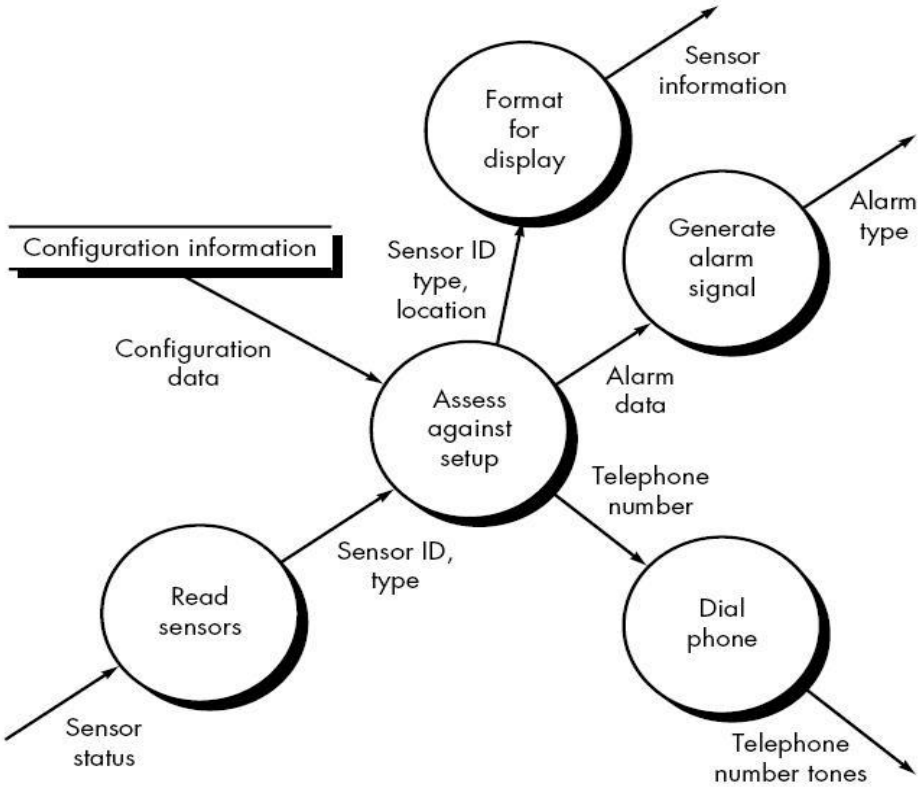
Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.



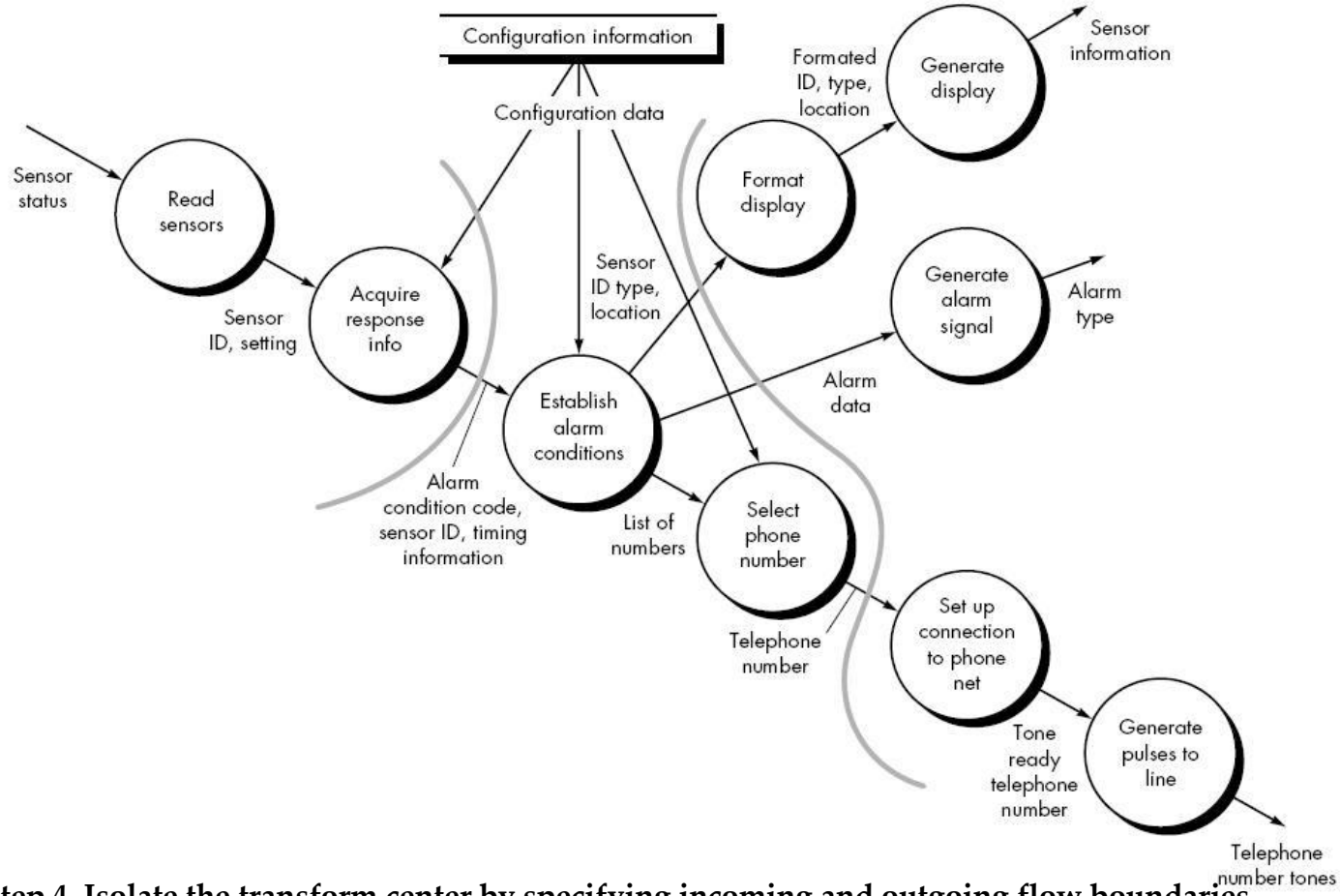
Step 1. Review the fundamental system model.



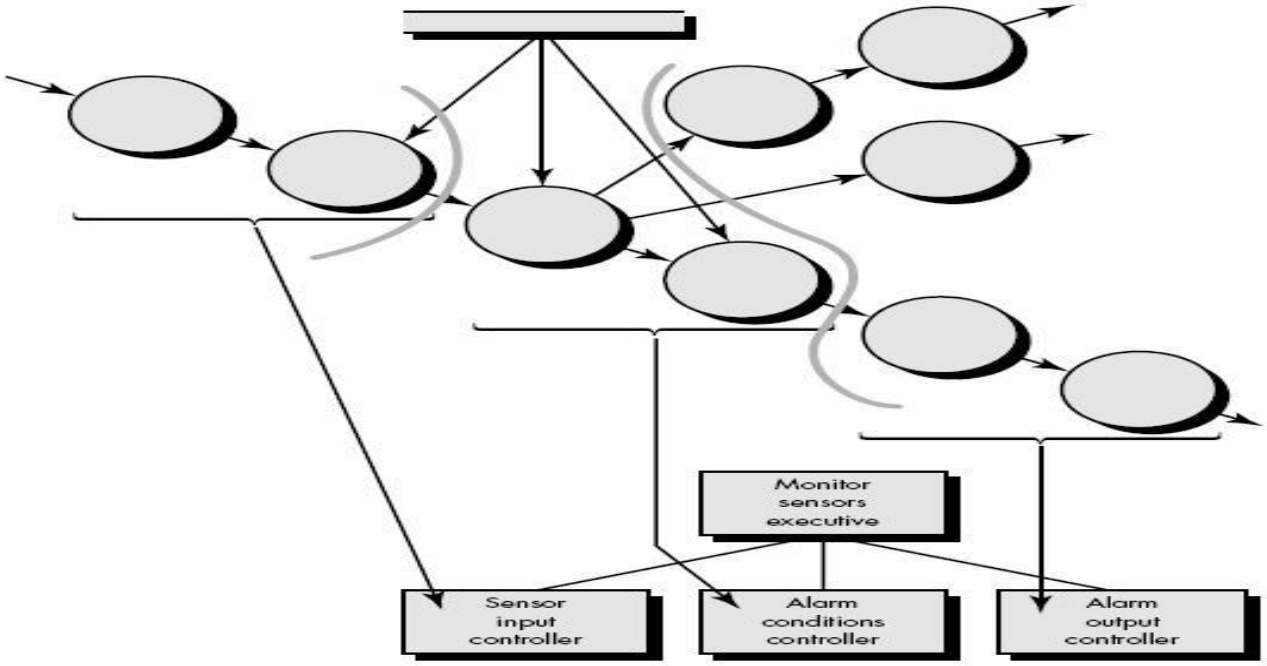
Step 2. Review and refine data flow diagrams for the software.



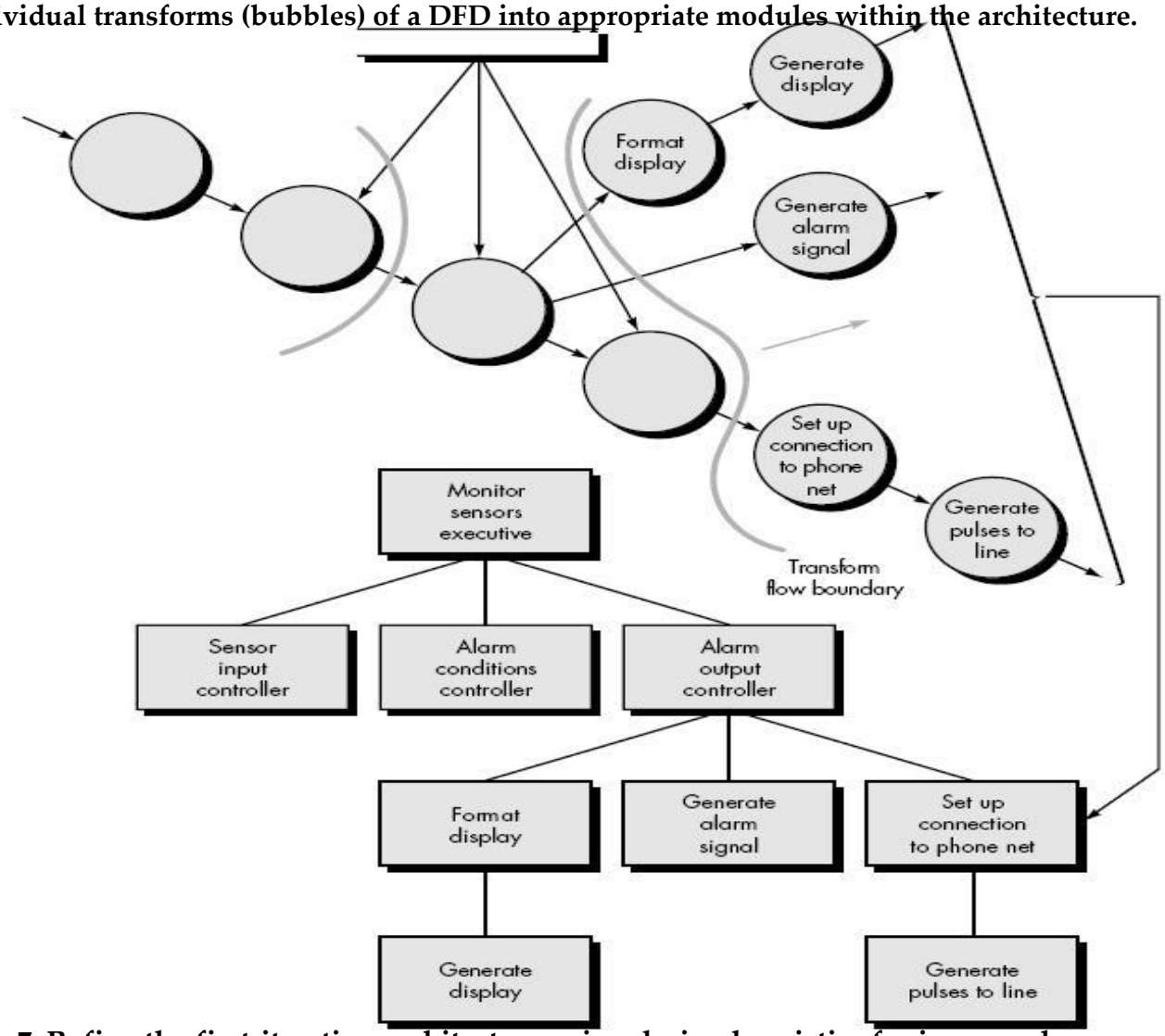
Step 3. Determine whether the DFD has transform or transaction flow characteristics.



Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.
Step 5. Perform "first-level factoring." Program structure represents a top-down distribution of control.

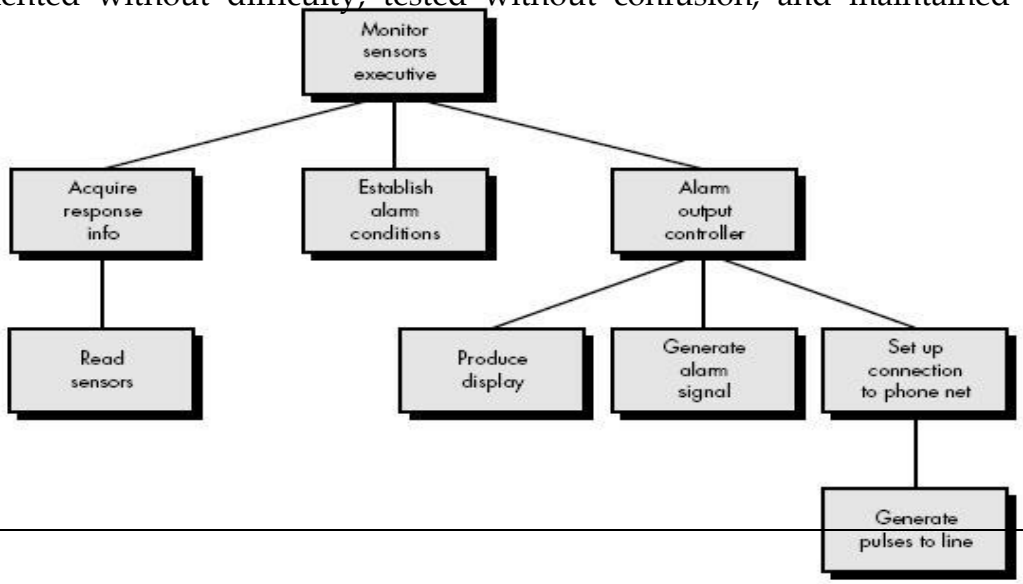


Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture.



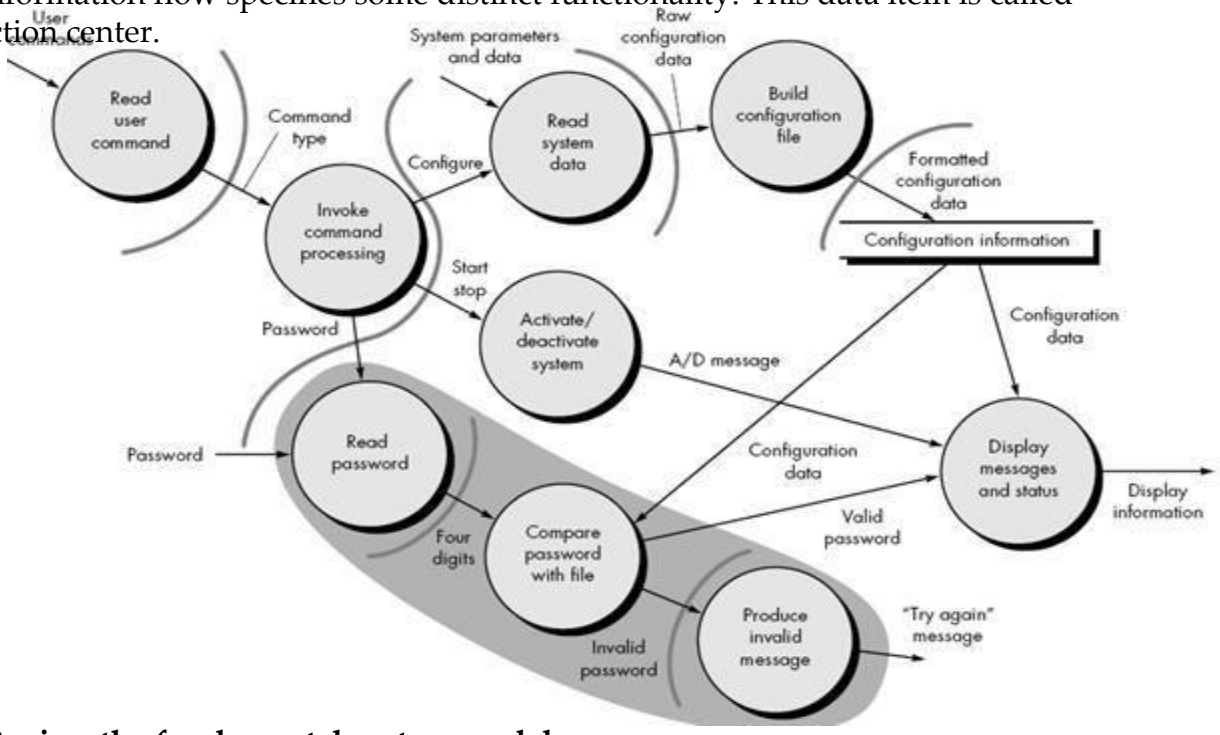
Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.

- First-iteration architecture can always be refined by applying concepts of functional independence.
- Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

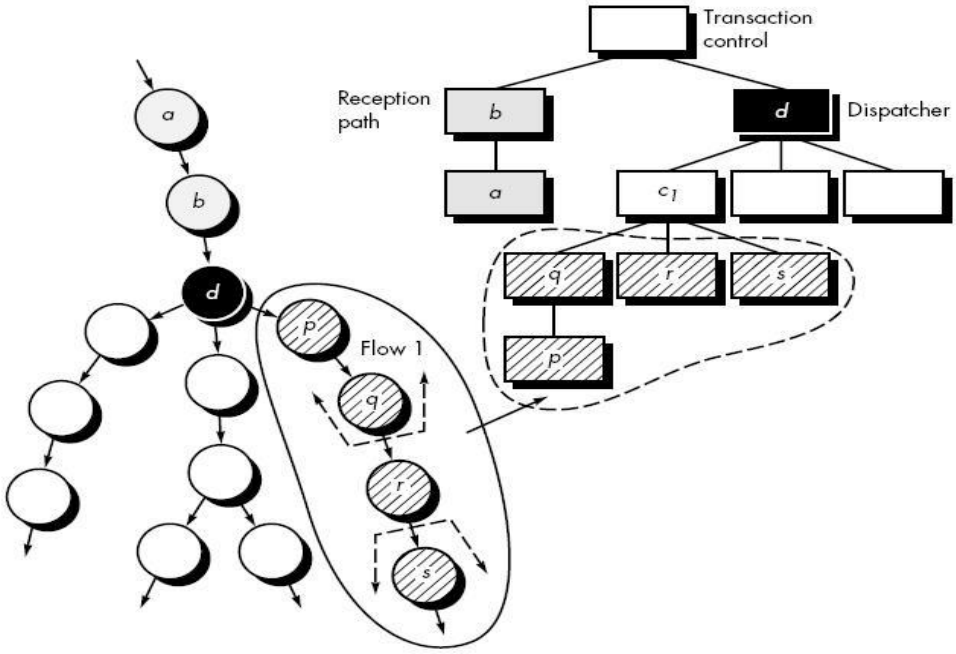


(ii)Transaction Mapping

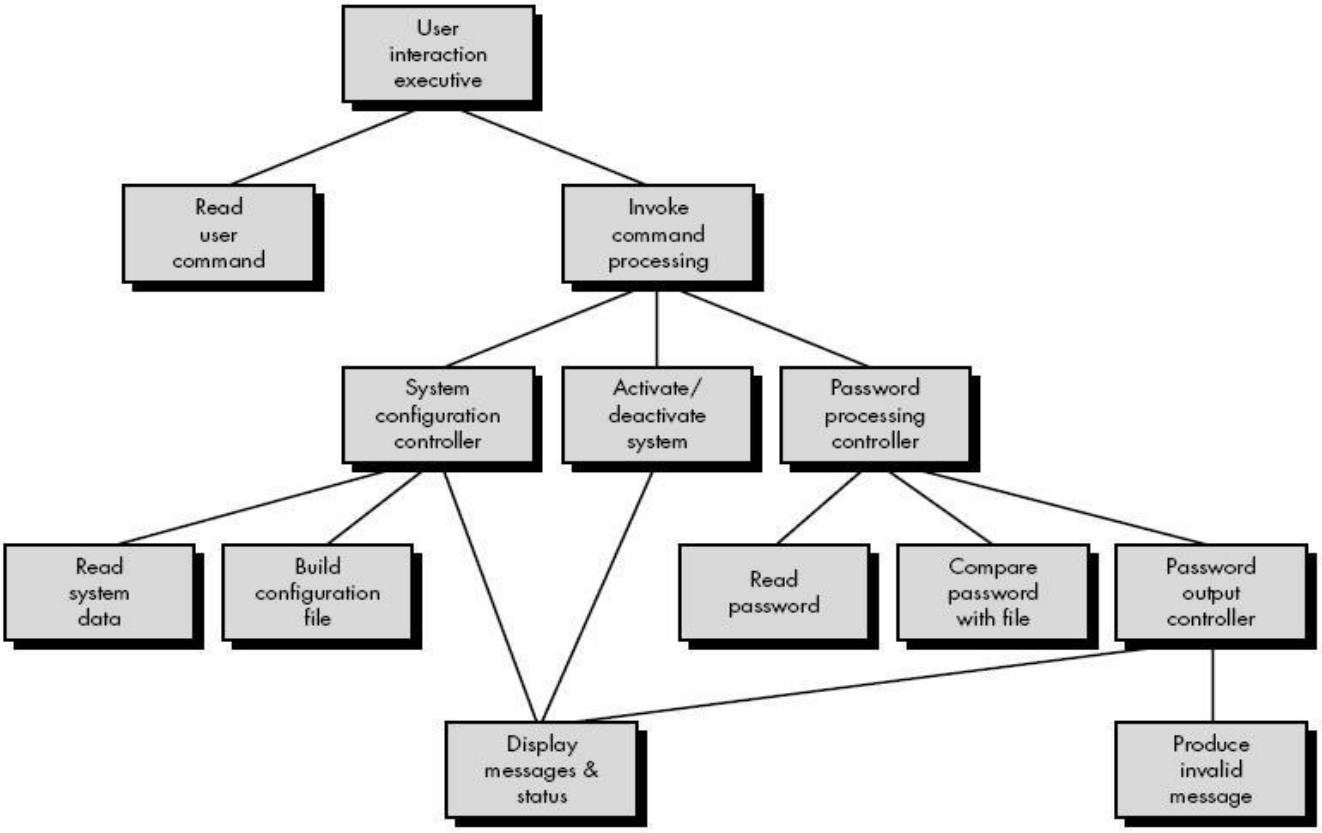
- In many software applications a single data item leads to one or more information flows.
- Each information flow specifies some distinct functionality. This data item is called transaction center.



Step 1. Review the fundamental system model.
Step 2. Review and refine data flow diagrams for the software.
Step 3. Determine whether the DFD has transform or transaction flow characteristics.
Step 4. Identify the transaction center and the flow characteristics along each of the action paths.
Step 5. Map the DFD in a program structure amenable to transaction processing.



Step 6. Factor and refine the transaction structure and the structure of each action path.



Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.

3.5 User Interface Design:

The Golden Rules
- ❖ Three Rules

User interface analysis and design
- ❖ Interface analysis and design model
- ❖ The process

Interface analysis
- ❖ User analysis

Interface design
- ❖ Design issues

User interface:

User interface is a communication between human and computer i.e. HCI (Human Computer Interaction).

User interface design:

- The overall process for designing a user interface begins with the creation of different models of system function (as perceived from the outside).
- The human- and computer-oriented tasks that are required to achieve system function are then delineated; design issues that apply to all interface designs are considered; tools are used to prototype and ultimately implement the design model; and the result is evaluated for quality.

3.5.1 The Golden Rules

- Place the user in control.
- **Reduce the user's memory load.**
- Make the interface consistent.

(i) Place the user in control:

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

(ii)Reduce demand on short-term memory.

- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

(iii)Make the Interface Consistent:

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

3.5.2 User Interface analysis and Design:

Steps:

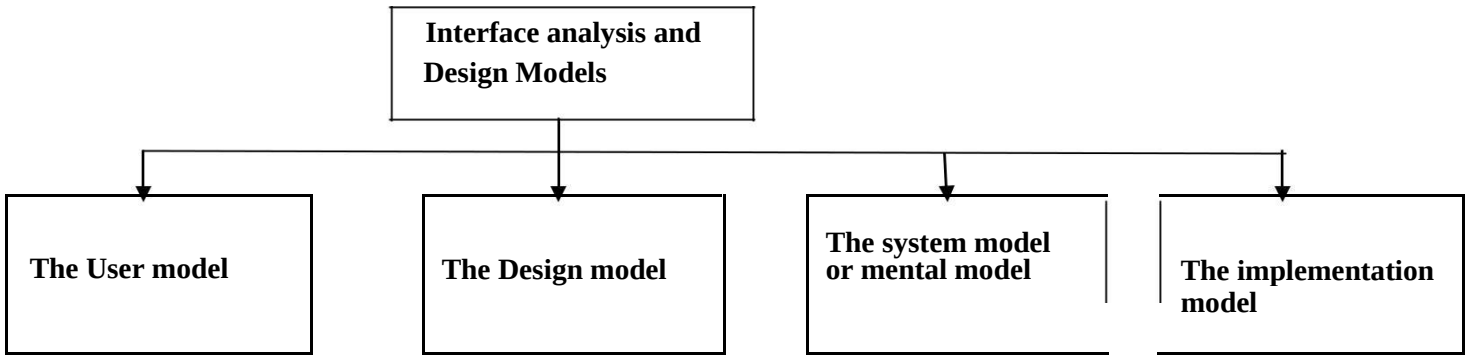
- Create different models for system functions.
- In order to perform these functions identify the human-Computer interface tasks.
- Prepare all interface design by solving various design issues.
- Apply modern tools and techniques to prototype the design.
- Implement design model
- Evaluate the design from end user to bring quality in it.

Two major things to be discussed in this topic

- Interface analysis and Design Models
- The process

(i)Interface analysis and Design Models:

The overall process for analyzing and designing a user interface begins with the creation of different models of system function.



- User profile model – Established by a software engineer
- Design model – Created by a software engineer
- Implementation model – Created by the software implementers
- User's mental model – Developed by the user when interacting with the application

Interface Design.

User models:

- The user model establishes **the profile of end users** of the system.
- **To build an effective user interface, all design should begin with an understanding** of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic **background, motivation, goals and personality .**
- **There are three types of users.**
  - ***Novices****: No syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general.*
  - ***Knowledgeable, intermittent users****: Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.*
  - ***Knowledgeable, frequent users:*** *Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.*

Design Model :

- It consists of data, architectural, interface and procedural representation of the software.
- While preparing this model, the requirement specification must be used properly to set the system constraints

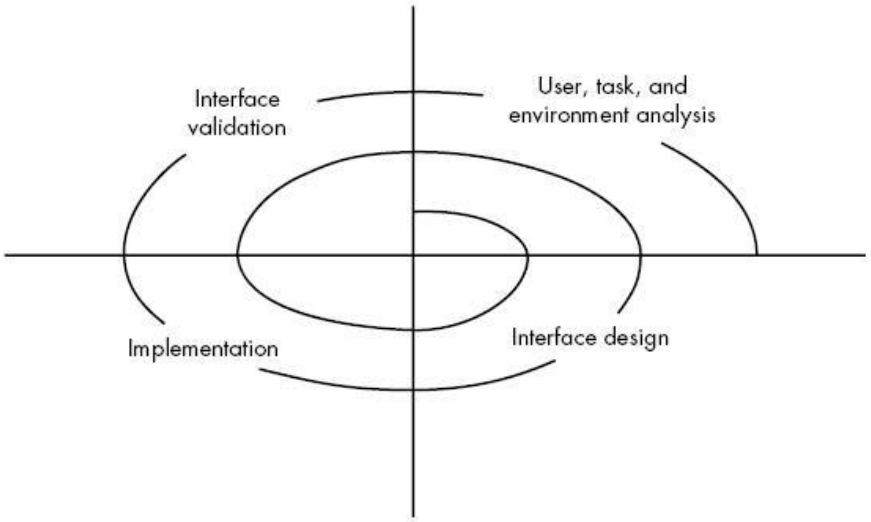
Mental Model (System Model):

- The user's mental model (system perception) is the image of the system that end users carry in their heads.
  - **For example**, if the user of a particular **word processo**r were asked to describe its operation, the system perception would guide the response. The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain.
  - A user who understands word processors fully but has worked with the specific word processor only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

Implementation model:

- Implementation model gives the look and feel of the interface, coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics.
- Matching implementation model with users mental model is necessary so that user feel comfortable with developed system.

(ii)The Process:



The design process for user interfaces is iterative and can be represented using a spiral model. The user interface design process encompasses four distinct framework activities:

- environment analysis and modeling
- Interface design
- Interface implementation
- Interface validation

Interface analysis: It focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited.

Interface design: The goal of *interface design* is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Interface validation focuses on

- ✓ The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;
- ✓ The degree to which the interface is easy to use and easy to learn, and
- ✓ **The users' acceptan**ce of the interface as a useful tool in their work.

3.5.3 Interface Analysis

A key tenet of all software engineering process models is this: *understand the problem before you attempt to design a solution.* In the case of user interface design, understanding the problem means understanding

- The people (end users) who will interact with the system through the interface,
- The tasks that end users must perform to do their work,
- The content that is presented as part of the interface,
- The environment in which these tasks will be conducted.

User Analysis

- **User Interviews.** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.
- **Sales input.** Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.
- **Marketing input.** Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.
- **Support input.** Support staff talks with users on a daily basis. They are the most likely source of **information on what works and what doesn't, what** users like and what they dislike, what features generate questions and what features are easy to use.

Task Analysis and Modeling

The goal of task analysis is to answer the following questions:

- ✓ **What tasks and subtasks will be performed as the user does the work?**
- ✓ **What specific problem domain objects will the user manipulate as work is** performed?
- ✓ **What is the sequence of work tasks** the workflow?
- ✓ **What is the hierarchy of tasks?**

These techniques are applied to the user interface to answer the above questions

- Use cases
- Task elaboration.
- Object elaboration
- Workflow analysis
- Hierarchical representation

3.5.4 Interface Design

Steps:

- Using information developed during interface analysis define interface objects and actions.
- Define events (user actions) that will cause the state of the user interface to change.
- Depict each interface state as it will actually look to the end user.
- Indicate how the user interprets the state of the system from information provided through the interface.

Design Issues

- ✓ **Response time:** System response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.
- ✓ **Help facilities.**
  - ❖ **Will help be available for all system functions and at all times during system** interaction?
  - ❖ **How will the user request help?**
  - ❖ **How will help be represented?**
  - ❖ **How will the user return to normal interaction?**
- ✓ **Error handling:** In general, every error message or warning produced by an interactive system should have the following characteristics:
  - ❖ **The message should describe the problem in jargon that the user can** understand.
  - ❖ **The message should provide construct**ive advice for recovering from the error.
- ✓ **Menu and command labeling.**
  - ❖ **Will every menu option have a corresponding command?**
  - ❖ **What form will commands take? Options include a control sequence (e.g.,** alt-P), function keys, or a
  - ❖ typed word.
  - ❖ **How difficult wi**ll it be to learn and remember the commands? What can be done if a command is
  - ❖ forgotten?
  - ❖ **Can commands be customized or abbreviated by the user?**
- ✓ **Application accessibility:** *Accessibility* for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines many designed for Web applications but often applicable to all types of software provide detailed suggestions for designing interfaces that achieve varying levels of accessibility.
- ✓ **Internationalization:** Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages.

3.6 COMPONENT LEVEL DESIGN:

Designing Class based components

- ❖ Basic design principle
- ❖ Component level design guidelines
- ❖ Cohesion
- ❖ Coupling

Traditional Components

- ❖ Structured programming

3.6.1 Designing Class based components

Component-level design, also called procedural design, occurs after data, architectural, and interface designs have been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low.

(i) Basic design principle

- ✓ ***The Open-Closed Principle (OCP):*** *Module [component] should be open for extension but closed for modification*
- ✓ ***The Liskov Substitution Principle (LSP):*** *Subclasses should be substitutable for their base classes*
- ✓ ***Dependency Inversion Principle (DIP):*** *Depend on abstractions. Do not depend on concretions*
- ✓ ***The Interface Segregation Principle (ISP***): *Many client-specific interfaces are better than one general purpose interface*
- ✓ ***The Release Reuse Equivalency Principle (REP):*** *The granule of reuse is the granule of release*
- ✓ ***The Common Closure Principle (CCP):*** *Classes that change together belong together.*
- ✓ ***The Common Reuse Principle (CRP):*** *Classes that aren't reused together should not be grouped together*

(ii)Component level design guidelines

Components:

- ✓ Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model.
- ✓ These names should be meaningful.

Interfaces:

- ✓ Interfaces provide important information about communication and collaboration.
- ✓ For consistency, interfaces should flow from the left-hand side of the component box.

Dependencies and Inheritance:

- ✓ Dependencies from left to right.
- ✓ Inheritance from bottom (derived classes) to top (base classes).
- ✓ Component interdependencies should be represented via interfaces.

(iii) Cohesion

- *Cohesion* is an indication of the relative functional strength of a module.
- Cohesion is a natural extension of the information-hiding.
- A cohesive module performs a single task, requiring little interaction with other components in other parts of a program.

Types of cohesion:

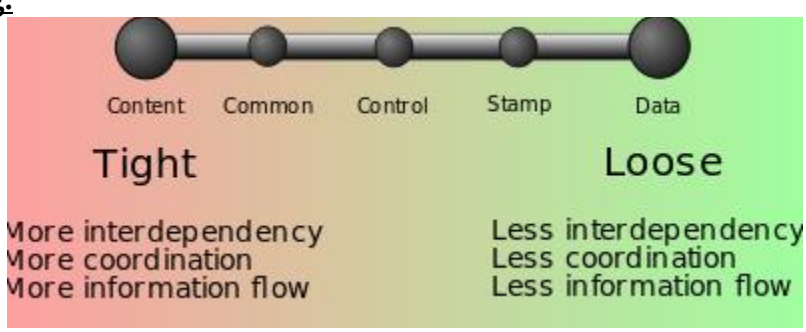
- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.
- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

- **Sequential cohesion** - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion** - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

(iv)Coupling

- Coupling is an indication of interconnection among modules in a software structure.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Types of Coupling:



- **Content coupling (high)**

Content coupling (also known as pathological coupling) occurs when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module). In this situation, a change in the way the second module produces data (location, type, timing) might also require a change in the dependent module.

- **Common coupling**

Common coupling (also known as global coupling) occurs when two modules share the same global data (e.g., a global variable). Changing the shared resource might imply changing all the modules using it.

- **External coupling**

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.

- **Control coupling**

Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

- **Stamp coupling (data-structured coupling)**

Stamp coupling occurs when modules share a composite data structure and use only parts of it, possibly different parts (e.g., passing a whole record to a function that only needs one field of it).In this situation, a modification in a field that a module does not need may lead to changing the way the module reads the record.

- **Data coupling**

Data coupling occurs when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

- **Message coupling (low)**

This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing.

3.6.2 Traditional Components

Structured programming:

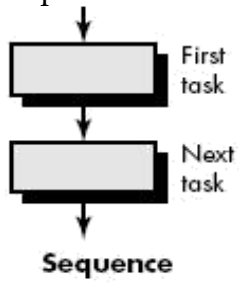
There are three constructs of structured programming

- Sequence- Sequence implements processing steps that are essential in the specification of any algorithm.
- Condition- Condition provides the facility for selected processing based on some logical occurrence.
- Repetition- Repetition allows for looping.

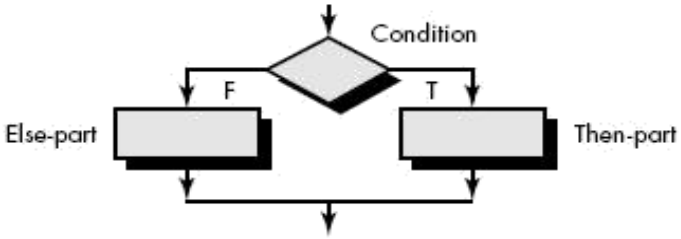
(i) Graphical Design Notation

➢ **Flowchart**

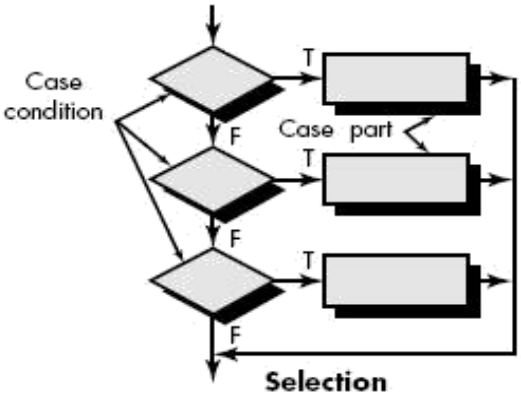
- A flowchart is quite simple pictorially. A box is used to indicate a processing step.
- A diamond represents a logical condition, and arrows show the flow of control.
- The sequence is represented as two processing boxes connected by a line (arrow) of control.
- Condition, also called if-then-else, is depicted as a decision diamond that if true, causes then-part processing to occur, and if false, invokes else-part processing.
- Repetition is represented using two slightly different forms.



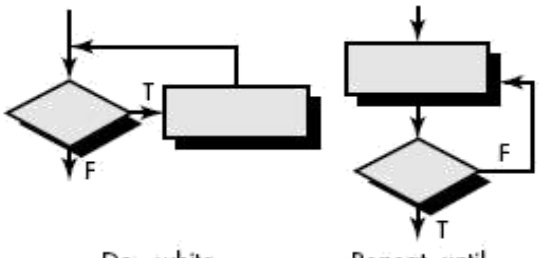
Sequence



If-then-else



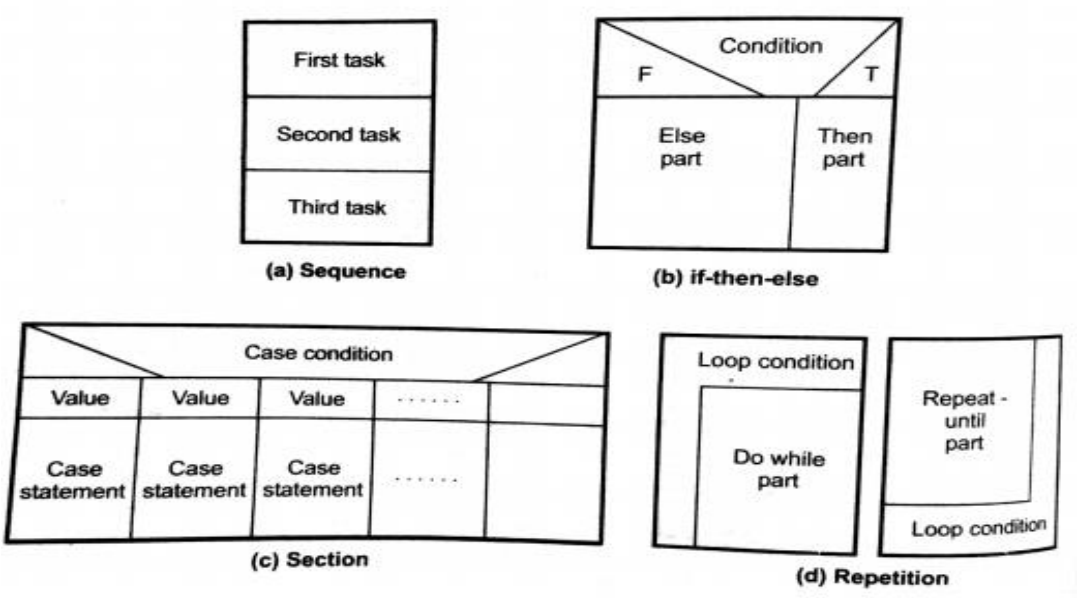
Selection



Repetition

➢ **Box Diagram**

- It is also called as NS chart.
- The scope of the programming constructs such as repetition . if-then-else is well defined.
- Arbitrary transfer of the control is not possible using the notation.
- Recursion can be represented conveniently.
- The scope of local and global data can be defined systematically.



(ii)Tabular Design Notation

- In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions.
- Decision tables provide a notation that translates actions and conditions into a tabular form.
- The table is difficult to misinterpret and may even be used as a machine-readable input to a table-driven algorithm.

	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount		✓		✓		✓

(iii)Program design language

- Program design language (PDL), also called structured English or pseudocode,
- It incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English).
- Narrative text (e.g., English) is embedded within a programming language-like syntax.
- Automated tools can be used to enhance the application of PDL.
- A basic PDL syntax should include constructs for
  - Component definition,
  - Interface description,

- Data declaration,
- Block structuring,
- Condition constructs,
- Repetition constructs,
- Input-output (I/O) constructs.

- It should be noted that PDL can be extended to include keywords.