CHAPTER 7

DESIGN CONCEPTS AND PRINCIPLES

- **DESIGN :**

  Software Design: An iterative process transforming requirements into a “blueprint” for constructing the software.
  It can be also defined as the process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realization.
  SW Design is the first of three technical activities: design, coding, testing.
  During designing, each activity transforms information in a manner that ultimately results in validated computer software.
  Each of the elements of the analysis model provides information that is necessary to create the four design models required for a complete specification of design.
  The flow of information during software design is illustrated in Figure1. There are 4 types of Software Design:
  - Data design
  - Architectural design
  - Interface design
  - Procedural design

1. **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.
   Data design is often included within Architectural and Component level design.

2. **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.

3. **INTERFACE DESIGN:**
   The 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.

4. **COMPONENT LEVEL DESIGN:**
   The 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 (State Transition Diagram) serves as the basis for component design.

   During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained.

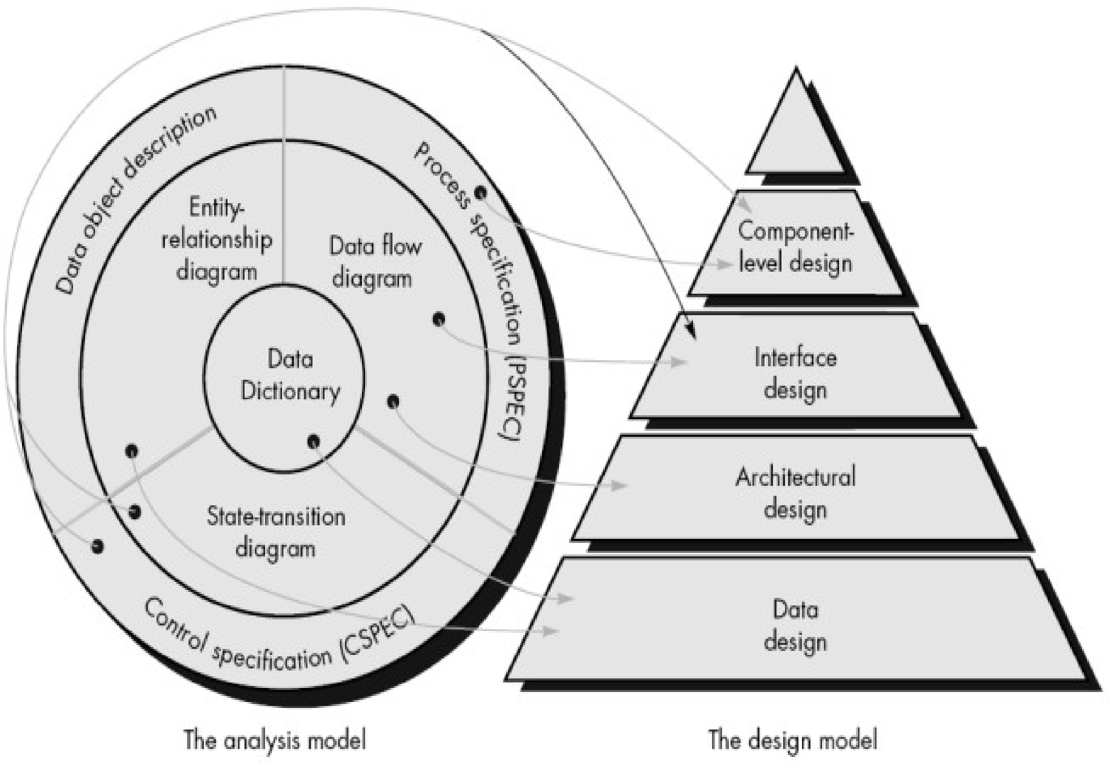


Figure 1: Translating the analysis model into a software design

❖ IMPORTANCE OF SOFTWARE DESIGN:

1. Maintains quality of the s/w.
2. Representations of software.
3. Translate Customer requirements.
4. Design provides us with representations of software that can be assessed for quality.
5. Without design we risk building an unstable system- one that will fail when small changes are made, one that may be difficult to test.

❖ THE DEDSIGN PROCESS:

Software design is an iterative process through which requirements are translated into a "Blueprint" for constructing the software.

Firstly high level design or abstraction level design is drafted. With each iteration, the details in the designing are integrated. Hence low level designs are achieved.

High level design is often termed conceptual design and low level design is called detailed design.

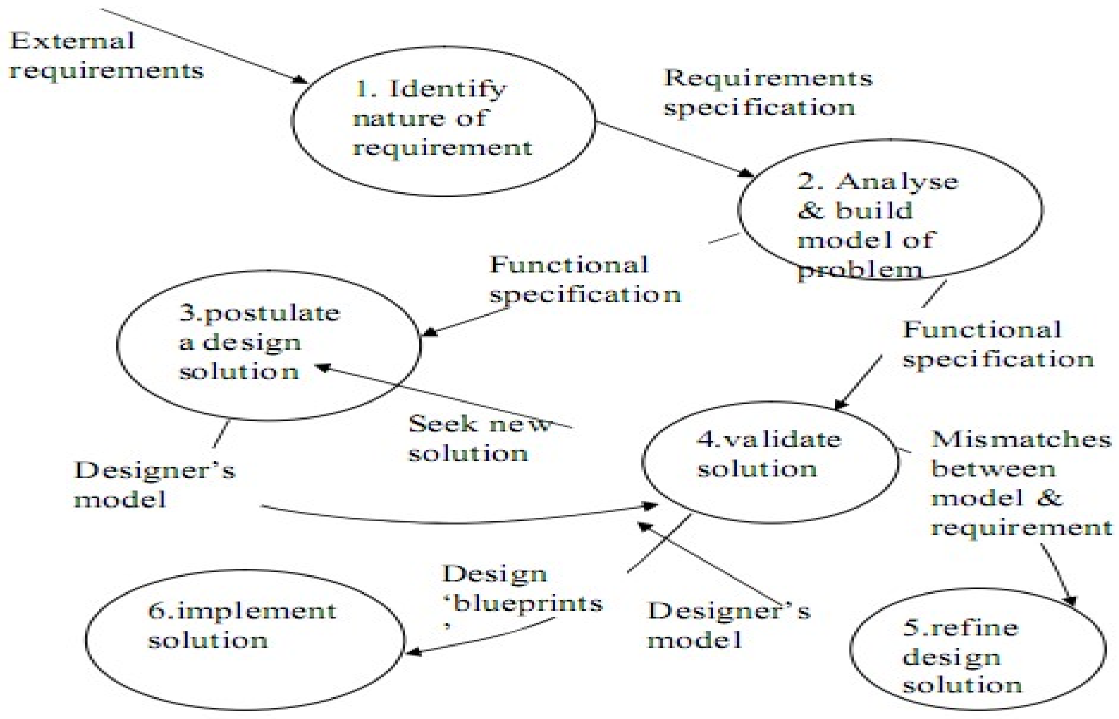


Fig: Model of the design process

- **How should a good design be?**

A good design:

1. Must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
2. Must be readable, understandable guide for coders, testers and maintainers.
3. Should provide a complete picture of the SW, addressing the data, functional and behavioral domains from an implementation perspective.
4. Should exhibit a hierarchical
5. Should contain both data and procedural abstractions.
6. Modules should exhibit independent functional characteristics.
7. Interfaces should reduce complexity.
8. Should be obtained from a repeatable method, driven by analysis.

❖ THE DESIGN PRINCIPLES:

The basic design principles are:

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" between the SW and the problem as it exists in the real world.
5. The design should exhibit uniformity & 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 is being created.
10. The design should be reviewed to minimize conceptual (semantic) errors.

❖ DESIGN CONCEPTS

The design concept provides the development with foundation from which more sophisticated design methods can be applied. The design concepts help in determining the criteria to partition the software into different components. Then we can determine how data structure or function are assigned to different components as responsibilities and finally maintain the quality. Few of the factors, while designing, to be focused are:

1. ABSTRACTION:

In software engineering, abstraction is a technique for arranging complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level. It concentrate on the essential features and ignore details that are not relevant.

- Data Abstraction
  - A named collection of data that describes a data object
- Procedural Abstraction
  - A named sequence of instructions that has a specific & limited function
- Control Abstraction
  - Implies a program control mechanisms without specify internal details

2. **REFINEMENT**:

- Refinement is the process of elaboration. It is similar to the process of refinement & partitioning in requirement analysis.
- It is a top-down strategy in which development is done by refining levels of procedural details.
- Refinement is a process where one or several instructions of the program are decomposed into more detailed instructions.
- We begin with the description at the higher level; ultimately we refine the statement by describing till we reach the lowest level of abstraction.

3. **MODULARITY**:

In this concept, software is divided into separately named and addressable components called modules. It follows "divide and conquer" concept, a complex problem is broken down into several manageable pieces.
Modular Design (Easier to build, easier to change, easier to fix).
But if we divide software indefinitely, the effort to develop individual software does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size.
As the number of modules, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in figure below:

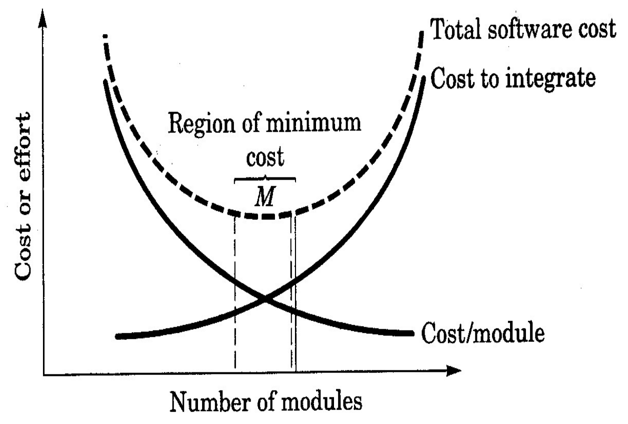


FIGURE 13.2.
Modularity and software cost

4. **SOFTWARE ARCHITECTURE**

   It is the overall structure of the software and the ways in which the structure provides conceptual integrity for a system. In simple words, architecture is the hierarchical structure of program components, the manner in which they interact and the data structure that these components uses.

   S/w Architecture must focus on these points as:

   -Structural properties

   -Extra functional properties

   -Families of related systems

Five different types of models are used to represent the architectural design:

– Structural models
  - represent architecture as an organized collection of program components
– Framework models
  - Identify repeatable architectural design framework that similar to the types of applications
– Dynamic models
  - Behavioral aspects of the program architecture
– Process models
  - design of the business or technical process of a system
– Functional models
  - Functional hierarchy of a system

5. **CONTROL HIERARCY**

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

   It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

6. **DATA STRUCTURE**

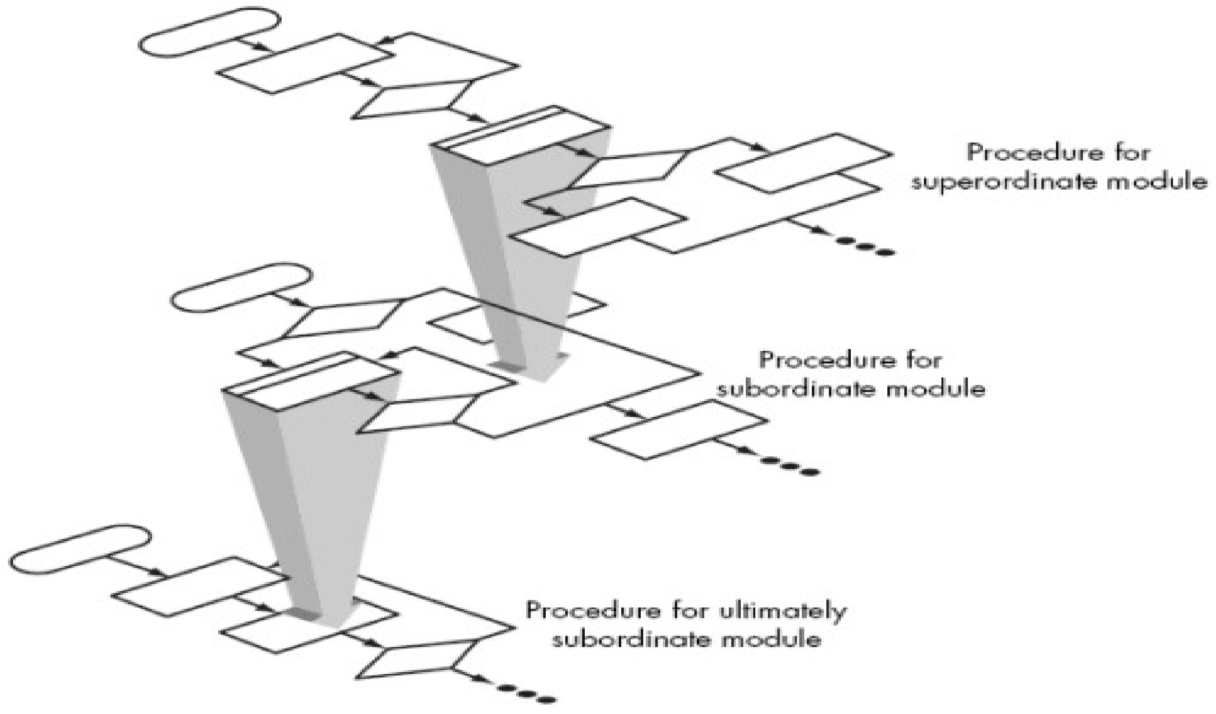
   Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture.

   Data structure dictates the organization, methods of access, degree of associatively, and processing alternatives for information.

7. **SOFTWARE PROCEDURE:**

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually.

Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.


Procedure for superordinate module
Procedure for subordinate module
Procedure for ultimately subordinate module

8. **INFORMATION HIDING**

The concept of modularity leads every software designer to a fundamental question: "How do we decompose a software solution to obtain the best set of modules?“

The principle of information hiding suggests that modules be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

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

Errors introduced during modification are less likely to propagate to other locations within the software.

❖ WHAT IS COMPLEX LEVEL DESIGN?

A complete set of software components is defined during architectural design.
But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code.
Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component.
A component is a set of collaborating classes in Object Oriented view.

OBJECTIVES:

1. Identify design classes in problem domain.
2. Identify infrastructure design classes.
3. Elaborate design classes.
4. Describe persistent data sources.
5. Elaborate behavioral representations.
6. Elaborate deployment diagrams.
7. Refine design and consider alternatives.

❖ SOFTWARE ARCHITECTURE

The structure of the system which compromises software components, externally visible properties of those components and relationship between them is termed as s/w architecture.
It is the representation that helps the s/w engineer to analyze the effectiveness of the design in meeting requirements, consider alternatives at a stage when making design changes is still relatively easy and reducing the risk associated with the construction of the s/w.
Importance of s/w architecture:

1. Facilitate communication between all parties interested in development of the computer based system.
2. The architecture highlights early designs decisions that have profound impact on all the succeeding work ultimately effecting success of the system as an operational entity.
3. Architecture constitutes a relatively small intellectual model of how the system is structured and how its components work.

❖ DATA DESIGN

The data design creates a model of data or information that is represented at high level of abstraction. This data model is then iteratively refined to reach the lowest level of abstraction that can be processed by the computer based system.
Data design activity translates the elements of requirements model into data structure at the software components. With time and bigger and multiple databases serving a common application, the concept of data mining and data warehousing has evolved.

The data design activity translates these elements of the requirements model into data structure at the software component level and when necessary a database architecture at the application level.
With time and bigger and multiple databases serving a common application, the concept of data mining and data warehousing has evolved.
Data mining often known as knowledge discovering in databases (KDD), helps in navigating through existing databases in an attempt to extract appropriate business level information.
Specially for multiple databases, data mining in complex and difficult and data warehousing in suitable for such.
Data warehousing adds an additional layer to the data architecture. It is a separate data environment that is not directly integrated with day to day application but encompasses all data used by a business. It is a large independent database that encompasses some but not all of the data that are stored in databases that serves the set of application required by business. Few characteristics of data warehousing are:

- Subject orientation
- Time variance
- Integration
- Non volatility

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

❖ ARCHITECTURAL STYLE

Architectural style describes a system category that encompasses:

- Set of components that perform a function required by a system.
- Set of connectors that enable "communication, coordination and cooperation" among components.
- Constraints that define how components can be integrated to form the system and
- Semantic models that enable a designer to understand the overall properties of a system by analysis of the known properties of its constituent's parts.

Generally used architectural styles are:

1. **Data centered architecture:**
   Data store resides at the center of this architecture and is accessed frequently by other components that update, add and delete data within the store.
   Existing components can be changed and new client components can be added to the architecture without concern about another client.

Data centered architectures promote integration. That is existing components can be changed and new client components can be added to the architecture without concern about other clients.

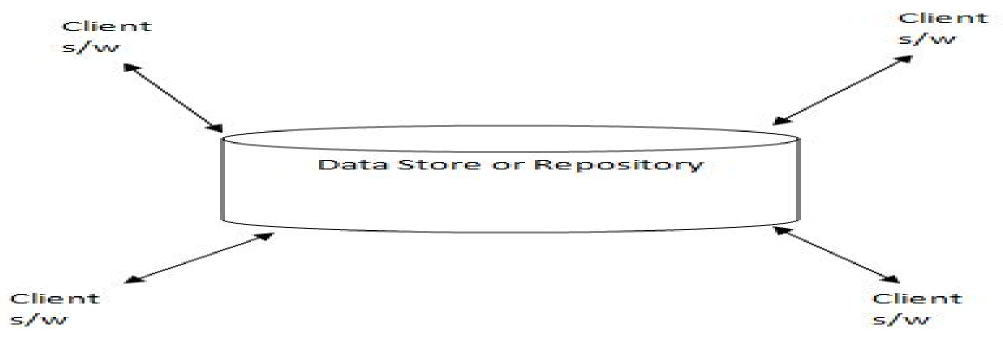
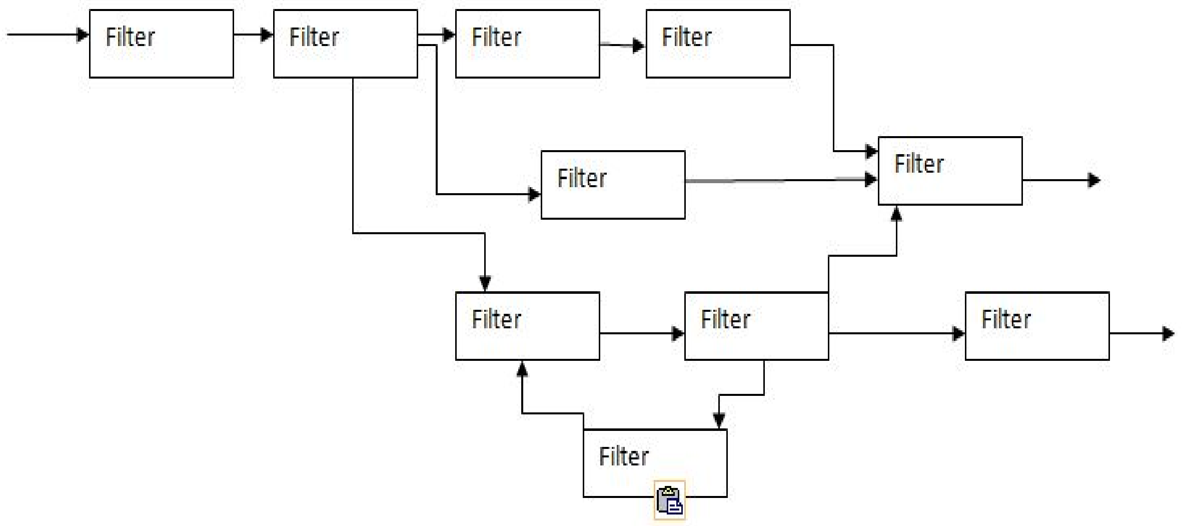


Fig: Data centered architecture

2. **Data flow architecture:**
This architecture is applied when input data are transformed through 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 transmits data from one components to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form and produce data output of a specified form. The filter doesn't 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 that applies a series of sequential components to transform it.



3. **Call and return architecture:**
   This architecture enables a s/w designer to achieve a program structure that is relatively easy to modify and scale.
   Types:
   a. Main program/ sub program architecture:
   This structure decomposes functions into number of program components which in turn may invoke other components.
   b. Remote procedure calls architecture:
   The components of a main program are distributed across multiple n/w.

4. **Object oriented Architecture:**
   This architecture encapsulates data and the operations that must be applied to manipulate the data. Communication and coordination between components is achieved via message passing.

5. **Layered Architecture:**
   In this approach different layered are defined each accomplishing some operations that progressively become closer to machine instruction set.

❖ MAPPING REQUIREMENTS INTO SOFTWARE ARCHITECTURE

We have already study that requirements are to be mapped into design and the architecture styles are different and there is no exact technique of mapping. Basically, structured design focus on data flow because it provides a convenient transition of software architecture from the data flow data flow diagram.
The transition of the program structure from information is achieved in six steps:

1. Establish the type of information flow
2. Indicate flow boundaries
3. Mapping DFD into program structure
4. Define control hierarchy
5. Refine resultant structure using design measure
6. Refine and elaborate architecture description

There are two types of flow of information that drive mapping requirements:

a. Transform Flow:

The transform mapping is a set of design steps applied on the DFD in order to map the transformed flow characteristics (i/p – process- o/p) into specific architectural style. In this section transform mapping is described by applying design steps to an example system—a portion of the SafeHome security software.

For Eg: SafeHome security system (level 0)

- **Design Steps:**

1. **Review the fundamental system model:** The fundamental system model encompasses the level 0 DFD and supporting information. In actuality, the design step begins with an evaluation of both the System Specification and the Software Requirements Specification. Both documents describe information flow and structure at the software interface.
2. **Review and refine data flow diagrams for the software:** Information obtained from analysis models contained in the Software Requirements Specification is refined to produce greater detail. For example, the level 2 DFD for monitor sensors is examined, and a level 3 data flow diagram is derived . At level 3, each transform in the data flow diagram exhibits relatively high cohesion.
3. **Determine whether the DFD has transform or transaction flow characteristics:** In general, information flow within a system can always be represented as transform. However, when an obvious transaction characteristic is encountered, a different design mapping is recommended. In this step, the designer selects global (software wide) flow characteristics based on the prevailing nature of the DFD.
4. **Isolate the transform center by specifying incoming and outgoing flow boundaries:** In the preceding section incoming flow was described as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations.
5. **Perform "first-level factoring." Program structure represents a top-down distribution of control:** Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work.
6. **Refine the first-iteration architecture using design heuristics for improved software quality:** First-iteration architecture can always be refined by applying concepts of module independence. Modules are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

b. Transaction mapping:

In many software applications, a single data item triggers one or a number of information flows that affect a function implied by the triggering data item. The data item, called a transaction, and its corresponding flow characteristics.

- **Design steps:**

1. Review the fundamental system model

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

3. Determine whether the DFD has transform or transaction flow characteristics: Steps 1, 2, and 3 are identical to corresponding steps in transform mapping. The DFD shown in above figure has a classic transaction flow characteristic. However, flow along two of the action paths emanating from the invoke command processing bubble appears to have transform flow characteristics. Therefore, flow boundaries must be established for both flow types.

4. **Identify the transaction center and the flow characteristics along each of the action paths:** The location of the transaction center can be immediately discerned from the DFD. The transaction center lies at the origin of a number of actions paths that flow radically from it.

5. Factor and refine the transaction structure and the structure of each action path: Each action path of the data flow diagram has its own information flow characteristics. We have already noted that transform or transaction flow may be encountered. The action path-related "substructure" is developed using the design steps discussed in this and the preceding section.

6. **Refine the first-iteration architecture using design heuristics for improved software quality:** This step for transaction mapping is identical to the corresponding step for transform mapping. In both design approaches, criteria such as module independence, practicality (efficacy of implementation and test), and maintainability must be carefully considered as structural modifications are proposed.

❖ STRUCTURED PROGRAMMING:

- Structured programming (sometimes known as *modular programming*) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as Ada, Pascal, and dBASE are designed with features that encourage or enforce a logical program structure.

- Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or sub module, which means that code, can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

❖ COMPARISION OF DESIGN NOTATION

- A natural question that arises in any discussion of design notation is: "What notation is really the best, given the attributes noted?"
- The following attributes of design notation have been established in the context of the general characteristics:

1. Modularity
2. Overall simplicity
3. Ease of editing
4. Machine readability
5. Maintainability
6. Automatic processing
7. Data representation
8. Logic verification
9. "Code-to" ability