**Unit-III  
CHAPTER-II  
FUNCTION-ORIENTED SOFTWARE DESIGN**

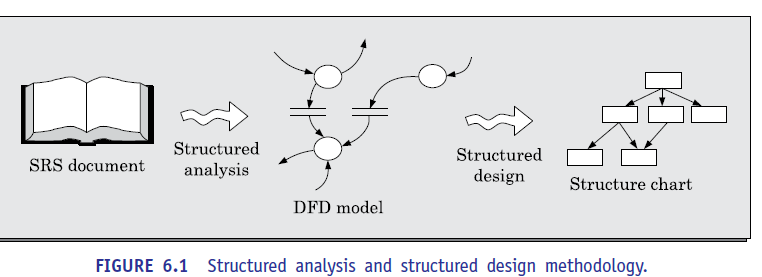
**Overview of SA/SD Methodology**

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

* Structured analysis (SA)
* Structured design (SD)

The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure 6.1. Observe the following from the figure:

* During structured analysis, the SRS document is transformed into a *data flow diagram* (DFD) model.
* During structured design, the DFD model is transformed into a structure chart.



As shown in Figure 6.1, the structured analysis activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system needs to perform is analyzed and hierarchically decomposed into more detailed functions.

On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the *high-level design* or the *software architecture* for the given problem. This is represented using a structure chart.

* The high-level design stage is normally followed by a detailed design stage. During the detailed design stage, the algorithms and data structures for the individual modules are designed. The detailed design can directly be implemented as a working system using a conventional programming language.
* The results of structured analysis can therefore, be easily understood by the user. In fact, the different functions and data in structured analysis are named using the user’s terminology. The user can therefore even review the results of the structured analysis to ensure that it captures all his requirements.
* In the following section, we first discuss how to carry out structured analysis to construct the DFD model. Subsequently, we discuss how the DFD model can be transformed into structured design.

**STRUCTURED ANALYSIS**

We have already mentioned that during structured analysis, the major processing tasks (high-level functions) of the system are analyzed, and the data flow among these processing tasks are represented graphically. Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978]. The structured analysis technique is based on the following underlying principles:

* Top-down decomposition approach.
* Application of divide and conquer principle. Through this each high-level function is independently decomposed into detailed functions.
* Graphical representation of the analysis results using *data flow diagrams* (DFDs).
* DFD representation of a problem, as we shall see shortly, is very easy to construct. Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.
* Please note that a DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed. In fact, it completely ignores aspects such as control flow, the specific algorithms used by the functions, etc. In the DFD terminology, each function is called a *process* or a *bubble*. It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.
* DFD is an elegant modelling technique that can be used not only to represent the results of structured analysis of a software problem, but also useful for several other applications such as showing the flow of documents or items in an organization. how a DFD can be used to represent the processing activities and flow of material in an automated car assembling plant. We now elaborate how a DFD model can be constructed.

**Data Flow Diagrams (DFDs)**

* The DFD (also known as the *bubble chart*) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system.
* The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism—it is simple to understand and use. A DFD model uses a very limited number of primitive symbols (shown in Figure 6.2) to represent the functions performed by a system and the data flow among these functions.
* Starting with a set of high-level functions that a system performs, a DFD model represents the subfunctions performed by the functions using a hierarchy of diagrams. We had pointed out while discussing the principle of abstraction in Section 1.3.2 that any hierarchical representation is an effective means to tackle complexity. Human mind is such that it can easily understand any hierarchical model of a system—because in a hierarchical model, starting with a very abstract model of a system, various details of the system are slowly introduced through different levels of the hierarchy. The DFD technique is also based on a very simple set of intuitive concepts and rules. We now elaborate the different concepts associated with building a DFD model of a system.

**Primitive symbols used for constructing DFDs**

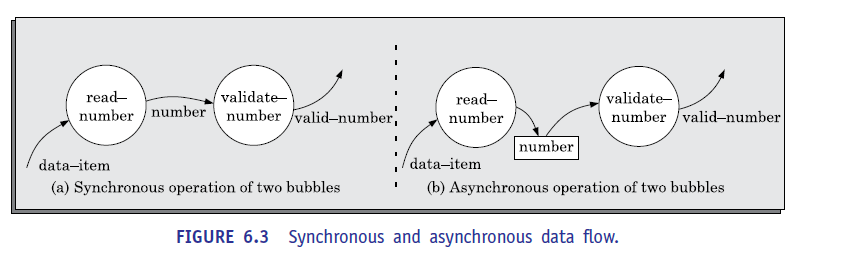
* There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure 6.2. The meaning of these symbols are explained as follows:
* **Function symbol:** A function is represented using a circle. This symbol is called a *process* or a *bubble*. Bubbles are annotated with the names of the corresponding functions (see Figure 6.3).
* **External entity symbol:** An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software that would interact with the software being modelled.
* **Data flow symbol:** A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names. For example, the DFD in Figure 6.3(a) shows three data flows—the data item number flowing from the process read number to validate-number, data-item flowing into read-number, and valid-numberflowing out of validate-number.
* **Data store symbol:** A data store is represented using two parallel lines. It represents ab logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire with the name of the corresponding data items. As an example of a data store, number is a data store
* **Output symbol:** The output symbol is as shown in Figure 6.2. The output symbol is used when a hard copy is produced. The notations that we are following in this text are closer to the Yourdon’s notations than to the other notations. You may sometimes find notations in other books that are slightly different than those discussed here. For example, the data store may look like a box with one end open. That is because, they may be following notations such as those of Gane and Sarson [1979].

**Important concepts associated with constructing DFD models**

Before we discuss how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs:

**Synchronous and asynchronous operations**

* If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed. An example of such an arrangement is shown in Figure 6.3(a). Here, the validate-number bubble can start processing only after the read-number bubble has supplied data to it; and the read-number bubble has to wait until the validate-number bubble has consumed its data.
* However, if two bubbles are connected through a data store, as in Figure 6.3(b) then the speed of operation of the bubbles are independent. This statement can be explained using the following reasoning. The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the consumer bubble consumes any of them.



**Data dictionary**

* Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model. Please remember that the DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1
* DFD, level 2 DFDs, etc., as shown in Figure 6.4 discussed in new subsection. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.
* For example, a data dictionary entry may represent that the data *grossPay* consists of the components *regularPay* and *overtimePay*. *grossPay* = *regularPay* + *overtimePay* For the smallest units of data items, the data dictionary simply lists their name and their type. Composite data items are expressed in terms of the component data items using certain operators. The operators using which a composite data item can be expressed in terms of its component data items are discussed subsequently.
* The dictionary plays a very important role in any software development process, especially for the following reasons:
* A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. A consistent vocabulary for data items is very important, since in large projects different developers of the project have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.
* The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
* The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and *vice versa*. Such impact analysis is especially useful when one wants to check the impact of changing an input value type, or a bug in some functionality, etc.

**Data definition**

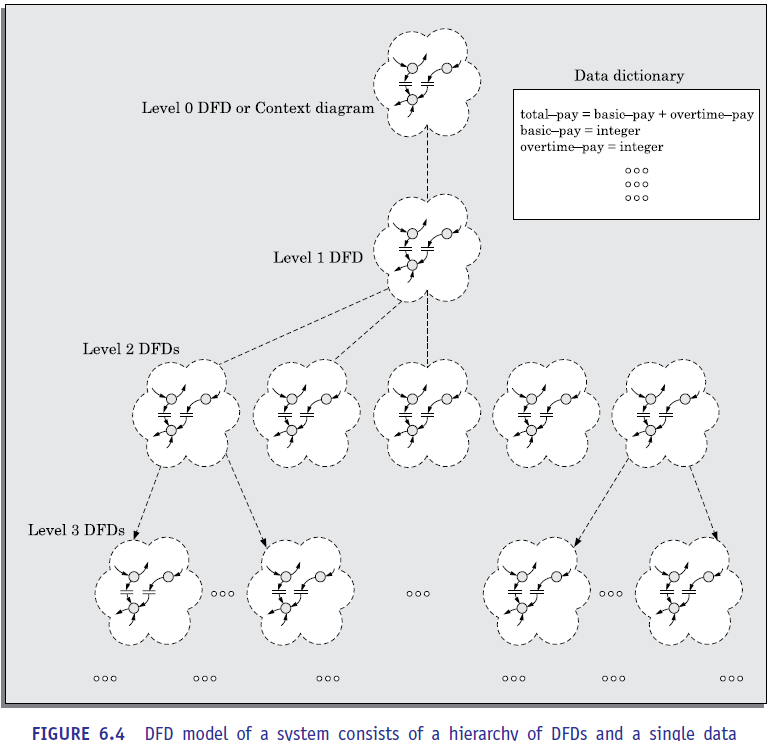
* Composite data items can be defined in terms of primitive data items using the following data definition operators.
* + : denotes composition of two data items, e.g. a+b represents data a and b.
* [,,] : represents selection, i.e. any one of the data items listed inside the square bracket can occur. For example, [a,b] represents either a occurs or b occurs.
* () : the contents inside the brac a+(b) represents either a or a+ b occurs.
* {} : represents iterative data definition, e.g. {name}5 represents five name data. {name}\* represents zero or more instances of name data.
* = : represents equivalence, e.g. a=b+c means that a is a composite data item comprising of both b and c.
* /\*\*/ : Anything appearing within /\* and \*/ is considered as comment. Ket represents optional data which may or may not appear.

**DEVELOPING THE DFD MODEL OF A SYSTEM**

* A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.
* The DFD model of a system is constructed by using a hierarchy of DFDs (see Figure 6.4). The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand.
* At each successive lower level DFDs, more and more details are gradually introduced. To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.
* To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on. However, there is only a single data dictionary for the entire DFD model.
* All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

**Context Diagram**

* The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble. The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is the only bubble in a DFD model, where a noun is used for naming the bubble.
* The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality.
* As an example of a context diagram, consider the context diagram a software developed to automate the book keeping activities of a supermarket (see Figure 6.10). The context diagram has been labelled as ‘Supermarket software’.
* The name context diagram of the level 0 DFD is justified because it represents the context in which the system would exist; that is, the external entities who would interact with the system and the specific data items that they would be supplying the system and the data items they would be receiving from the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data names.
* To develop the context diagram of the system, we have to analyze the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term users of the system also include any external systems which supply data to or receive data from the system.



**Level 1 DFD**

* The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions. To develop the level 1 DFD, examine the high-level functional requirements in the SRS document. If there are three to seven high level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD. Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

**Decomposition**

* Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as *factoring* or *exploding* a bubble. Each bubble at any level of DFD is usually decomposed to anything three to seven bubbles. A few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes trivial and redundant. On the other hand, too many bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm. We can now describe how to go about developing the DFD model of a system more systematically.

1. **Construction of context diagram:** Examine the SRS document to determine:

* Different high-level functions that the system needs to perform.
* Data input to every high-level function.
* Data output from every high-level function.
* Interactions (data flow) among the identified high-level functions.
* Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level *data flow diagram* (DFD), usually called the DFD 0.

2. **Construction of level 1 diagram:** Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.

3. **Construction of lower-level diagrams:** Decompose each high-level function into its constituent subfunctions through the following set of activities:

* Identify the different subfunctions of the high-level function.
* Identify the data input to each of these subfunctions.
* identify the data output from each of these subfunctions.
* Identify the interactions (data flow) among these subfunctions.
* Represent these aspects in a diagrammatic form using a DFD.
* Recursively repeat Step 3 for each subfunction until a subfunction can be represented by using a simple algorithm.

**Numbering of bubbles**

It is necessary to number the different bubbles occurring in the DFD. These numbers helping uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubble sat level 1 are numbered, 0.1, 0.2, 0.3, etc. When a bubble numbered x is decomposed, its children bubble is numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and itssuccessors.

**Balancing DFDs**

* The DFD model of a system usually consists of many DFDs that are organized in a hierarchy. In this context, a DFD is required to be balanced with respect to the corresponding bubble of the parent DFD. We illustrate the concept of balancing a DFD in Figure 6.5. In the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1 (shown by the dotted circle). In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1, 0.1.2, 0.1.3). The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in. Please note that dangling arrows (d1, d2, d3) represent the data flows into or out of a diagram.

**How far to decompose?**

* A bubble should not be decomposed any further once a bubble is found to represent a simple set of instructions. For simple problems, decomposition up to level 1 should suﬃce. However, large industry standard problems may need decomposition up to level 3 or level 4. Rarely, if ever, decomposition beyond level 4 is needed.

**Commonly made errors while constructing a DFD model**

* Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is a powerful thing, it is an expensive pedagogical technique in the business world. It is therefore useful to understand the different types of mistakes that beginners usually make while constructing the DFD model of systems, so that you can consciously try to avoid them. The errors are as follows:
* Many beginners commit the mistake of drawing more than one bubble in the context diagram. Context diagram should depict the system as a single bubble.
* Many beginners create DFD models in which external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level.
* It is a common oversight to have either too few or too many bubbles in a DFD. Only three to seven bubbles per diagram should be allowed. This also means that each bubble in a DFD should be decomposed three to seven bubbles in the next level.
* Many beginners leave the DFDs at the different levels of a DFD model unbalanced.
* A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD.

**Shortcomings of the DFD model**

* DFD models suffer from several shortcomings. The important shortcomings of DFD models are the following:
* Imprecise DFDs leave ample scope to be imprecise. In the DFD model, we judge the function performed by a bubble from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find book- position has only intuitive meaning and does not specify several things, e.g. what happens when some input information is missing or is incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.
* Not-well defined control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified.

A DFD model does not specify the order in which the different bubbles are executed Representation of such aspects is very important for modelling real-time systems.

* Decomposition: The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one.
* Improper data flow diagram: The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its subfunctions and we have to use subjective judgment to carry out decomposition

**STRUCTURED DESIGN**

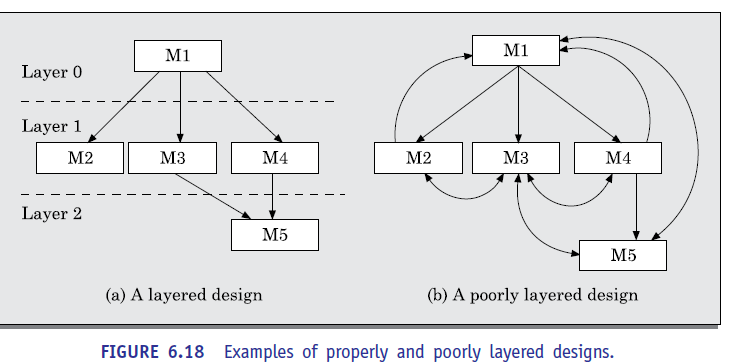
The aim of structured design is to transform the results of the structured analysis (that is, the DFD model) into a structure chart. A structure chart represents the software architecture. The various modules making up the system, the module dependency (i.e., which module calls which other modules), and the parameters that are passed among the different modules. The structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects (e.g., how a particular functionality is achieved) are not represented.

The basic building blocks using which structure charts are designed are as following:

* **Rectangular boxes:** A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.
* **Module invocation arrows:** An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow. However, just by looking at the structure chart, we cannot say whether a module calls another module just once or many times. Also, just by looking at the structure chart, we cannot tell the order in which the different modules are invoked.
* **Data flow arrows:** These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. Data flow arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.
* **Library modules:** A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called *modules*. Usually, when a module is invoked by many other modules, it is made into a library module.
* **Selection:** The diamond symbol represents the fact that one module of several modules connected with the diamond symbol is invoked depending on the outcome of the condition attached with the diamond symbol.
* **Repetition:** A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.
* In any structure chart, there should be one and only one module at the top, called the *root*. There should be at most one control relationship between any two modules in the structure chart. This means that if module A invokes module B, module B cannot invoke module A. The main reason behind this restriction is that we can consider the different modules of a structure chart to be arranged in layers or levels. The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules However, it is possible for two higher-level modules to invoke the same lower-level module. An example of a properly layered design and another of a poorly layered design

**Flow chart *versus* structure chart**

* We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:
* It is usually diﬃcult to identify the different modules of a program from its flow chart representation.
* Data interchange among different modules is not represented in a flow chart.
* Sequential ordering of tasks that is inherent to a flow chart is suppressed in a structure chart.



**Transformation of a DFD Model into Structure Chart**

* Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by as a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart:
* Transform analysis
* Transaction analysis
* At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

**Whether to apply transform or transaction processing?**

* Given a specific DFD of a model, how does one decide whether to apply transform analysis or transaction analysis? For this, one would have to examine the data input to the diagram. The data input to the diagram can be easily spotted because they are represented by dangling arrows. If all the data flow into the diagram are processed in similar ways (i.e., if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is applicable.
* Otherwise, transaction analysis is applicable. Normally, transform analysis is applicable only to very simple processing. Please recollect that the bubbles are decomposed until it represents a very simple processing that can be implemented using only a few lines of code. Therefore, transform analysis is normally applicable at the lower levels of a DFD model. Each different way in which data is processed corresponds to a separate transaction. Each transaction corresponds to a functionality that lets a user perform a meaningful piece of work using the software.

**Transform analysis**

* Transform analysis identifies the primary functional components (modules) and the input and output data for these components. The first step in transform analysis is to divide the DFD into three types of parts:
* Input.
* Processing.
* Output.

The input portion in the DFD includes processes that transform input data from physical (e.g., character from terminal) to logical form (e.g., internal tables, lists, etc.). Each input portion is called an *afferent branch.*

The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an *efferent branch.* The remaining portion of a DFD is called *central transform*.

The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an *efferent branch.* The remaining portion of a DFD is called *central transform*.

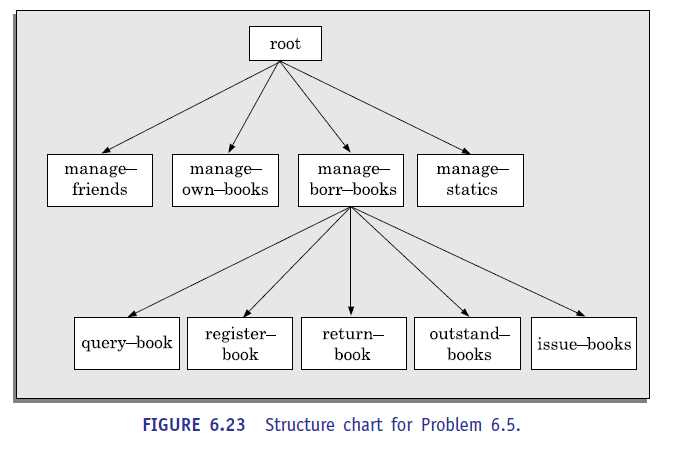
* In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches. These are drawn below a root module, which would invoke these modules. Identifying the input and output parts requires experience and skill.
* One possible approach is to trace the input data until a bubble is found whose output data cannot be deduced from its inputs alone. Processes which validate input are not central transforms.
* Processes which sort input or filter data from it are central transforms. The first level of structure chart is produced by representing each input and output unit as a box and each central transform as a single box.
* In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called *factoring*. Factoring includes adding read and write modules, error handling modules, initialization and termination process, identifying consumer modules etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

**Transaction analysis**

* Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs. A transaction allows the user to perform some specific type of work by using the software. For example, ‘issue book’, ‘return book’, ‘query book’, etc., are transactions.
* As in transform analysis, first all data entering into the DFD need to be identified. In a transaction-driven system, different data items may pass through different computation paths through the DFD. This is in contrast to a transform centered system where each data item entering the DFD goes through the same processing steps.
* Each different way in which input data is processed is a transaction. A simple way to identify a transaction is the following. Check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transactions may not require any input data. These transactions can be identified based on the experience gained from solving a large number of examples.
* For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction as a module. Every transaction carries a tag identifying its type. Transaction analysis uses this tag to divide the system into transaction modulesand a transaction-centre module.

**DETAILED DESIGN**

* During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart. These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English. The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower-level modules.
* The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out. To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.



**DESIGN REVIEW**

* After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing, and maintenance perspectives, who may or may not be the members of the development team. Normally, members of the team who would code the design, and test the code, the analysts, and the maintainers attend the review meeting. The review team checks the design documents especially for the following aspects:
* **Traceability:** Whether each bubble of the DFD can be traced to some module in the structure chart and *vice versa*. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model and *vice versa*.
* **Correctness:** Whether all the algorithms and data structures of the detailed design are correct.
* **Maintainability:** Whether the design can be easily maintained in future.
* **Implementation:** Whether the design can be easily and eﬃciently be implemented. After the points raised by the reviewers is addressed by the designers, the design document becomes ready for implementation.