3

Requirement Analysis Modeling

CHAPTER COVERAGE

3.1 ANALYSIS MODELING APPROACHES

As discussed in Chapter 2, the analysis stage acts as the bridge between design and requirement and it happens after the requirement phase. The analyst critically explores the requirements and ensures that the system's operational characteristics are understood clearly, interfacing requirements with other systems are brought out and the constraints that the proposed system should meet are established in detail. Figure 3.1 helps to understand the requirements in a better way.

During the Requirement Analysis stage, the analyst aims to fulfill the following objectives (Figure 3.2).

1. Clearly illustrate the user scenarios
2. Explain the functional activities in detail

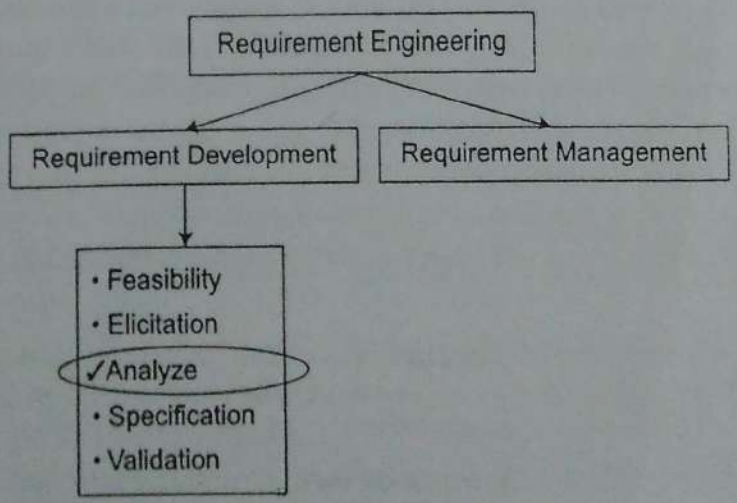


Figure 3.1 Requirement Analysis

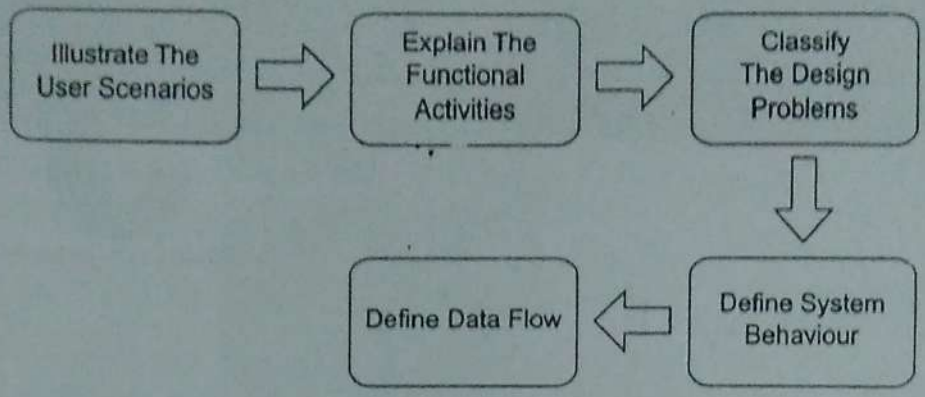


Figure 3.2 Requirement Analysis Objectives

3. Classify the design problems and their relationships
4. Define system behavior and class structure
5. Define data flow as it is transformed

Discussion Questions

Discuss the importance of the above objects in detail. List down at least 4 important points.

1. _____
2. _____
3. _____
4. _____

Providing a graphical representation of the system behavior (Figure 3.3), which is depicted as a combination of functional flow and data flow to create the input for the design phase is called the analysis modeling.

The below-mentioned guidelines (Figure 3.4) are followed while creating a model.

During the early days of software development in 1960s–1970s, most of the systems were written in languages available in those days (for example, COBOL and FORTRAN). There were not much focus on documenting the requirements and transforming the user needs into a system design. When the systems gradually started becoming bigger and complex, the structured analysis became popular. The two most popular approaches to software analysis, which have developed and got fine-tuned over years, are *structured analysis* and *object-oriented analysis* (Figure 3.5).

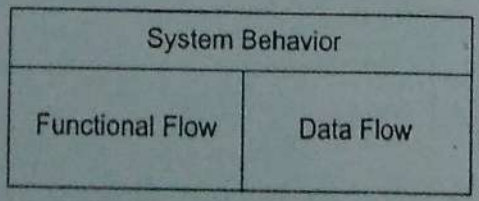


Figure 3.3 System Behavior

- Are the elements of the model improving the understanding of the requirements and addressing each of the area - domain, functional and behavioral requirements?
- Is the model focusing on the requirements that are captures during the requirements gathering phase?
- Is the model keeping analysis at relatively high level so that the architectural decisions can be made during the design?
- Is the graphical representation providing good insight of the system to all stakeholders?
- Is the model kept simple and as modularized as possible to avoid interdependencies?
- Can all the inputs required for starting the design be derived from this model?

Figure 3.4 Requirement Analysis Guidelines

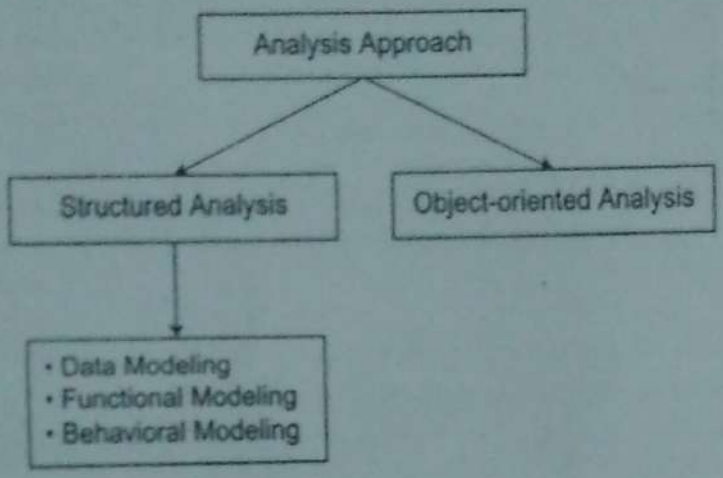


Figure 3.5 Analysis Approach

3.2 STRUCTURED ANALYSIS

The term structured analysis was first used by Douglas Ross and then popularized by DeMarco. The graphical notations and models suggested by DeMarco created the basis for structured analysis, which then got enriched by the works of Page–Jones and Gane and Sarson. In mid-1980s, the work of Ward and Mellor followed by Hatley and Pirbhai added notations to capture the control and behavioral aspects of the real-time engineering problems along with the information systems applications also called as function-oriented analysis.

The domain of structured analysis comprises three modeling approaches, which collectively describe the requirement and intended behavior of the proposed system

- Data modeling
- Functional or flow-oriented modeling and
- Behavioral modeling

As we go through elements of each of the modeling approaches it becomes clear that these techniques need to be used in conjunction with each other to get the full view of the system. By breaking the system into these three models, the analyst tries to make the designer's job easy who can elaborate each part of data, flow and the system's reaction to the data flow separately to create a modular design.

POINTS TO PONDER

The domain of structured analysis comprises of three modeling approaches

- Data modeling
- Functional or flow oriented modeling and
- Behavioral modeling

3.2.1 Data Modeling

Data modeling is an analysis technique (Figure 3.6) that deals with the data processing part of an application. It deals with identifying the data elements in a system, the structure and composition of data, relationships and different processes that causes transformation to these data.

The objective of data modeling is not to capture the details of the processes that cause data transformation or to reveal how data is transformed. Rather, this modeling technique focuses on independently analyzing the data elements and their relationship in the overall system. This visual representation, which is called the Entity Relationship Diagram (Table 3.1), helps the engineers understand and design data elements and finally the data dictionary of the system.

Some of the notations that help to draw the diagram are explained in the following section.

3.2.1.1 Data Objects, Attributes and Relationships

Data object or an entity (Figure 3.7) in a system is the identity which has multiple attributes and is related to other entities in the system. Identifying all entities in the system is a crucial task as these form the basis of all the relationships and flows inside the system that the engineers create.

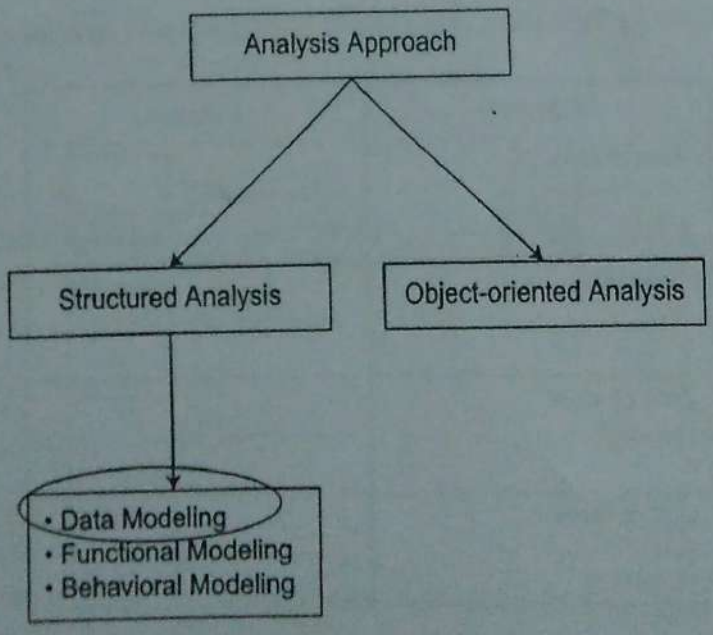


Figure 3.6 Data Modeling

Table 3.1 Structured Analysis – ER Diagram

Structured Analysis	
Type	**Corresponding Diagram**
Data Modeling	ER Diagram
Functional Modeling	Data Flow Diagram
	Control Flow Diagram
Behavioral Modeling	State Transition Diagram

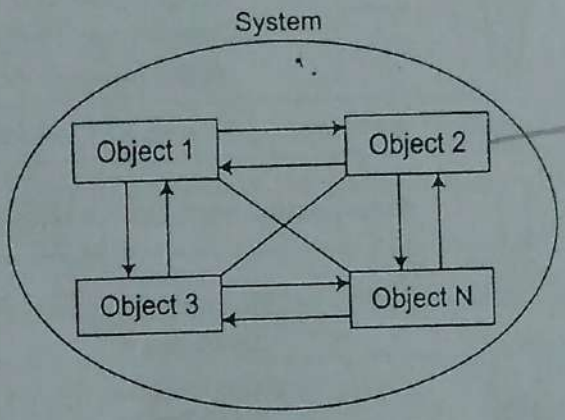


Figure 3.7 System versus Data Objects

POINTS TO PONDER

A System consists of multiple objects. The data flow occurs between objects which are inside the same or with the objects insider other system.

For example, in a health insurance system the entities will be

1. The insurer
2. The insurance service provider
3. The policy
4. The claims, etc.

Each of these entities possesses a set of attributes and also holds relationship with other entities. An insurer has attributes such as ID number, name, address, age, whereas a policy comprises attributes such as policy number, policy tenure, nominee, policy value, etc. (Figure 3.8).

The properties that describe the entity and are common across all the instances of the entity are called the *attributes*.

Entities are related to each other. For example, an insurer can hold one or multiple policies whereas an insurance provider may have several insurers attached to it. This leads to the concept of

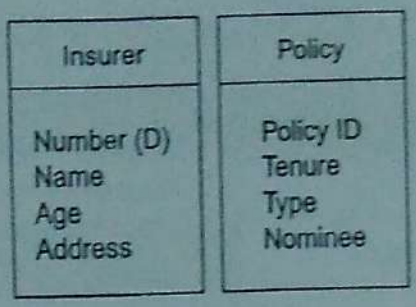


Figure 3.8 Insurer and Policy Objects

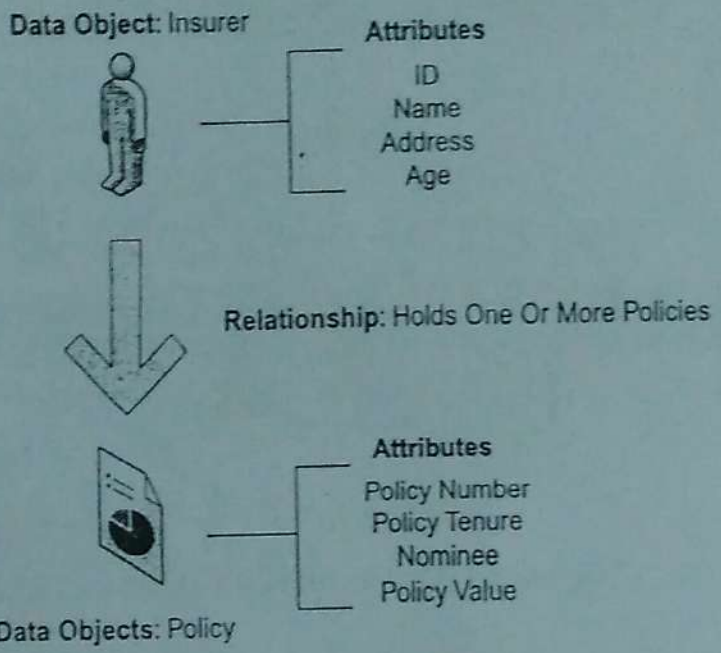


Figure 3.9 Data Object, Attributes, Relationship

relationship where the entities can be linked to each other in a system. Figure 3.9 shows the entity, attributes and relationships in a health insurance system.

Data objects hold only the data components and there should be no intent to hold the processes that deal with the data inside data objects. If the process, which a claim approval of a policy undergoes, needs to be captured then the data flow diagram needs to be initiated and it should not be captured inside the ERD. Here, the structure of the database or the system that holds the data for the system logically comes out from this analysis.

3.2.1.2 Cardinality and Modality

Along with the relationship information of different objects, it is also important to capture how many types of each object can be attached with other objects, which is called *cardinality* and whether it is mandatory to attach an object with the another object type which is called *modality*.

In the example of the health insurance system, an insurer can have one or many insurance policies which will give rise to one-to-many relationship, whereas a policy can be attached to one insurer only giving rise to one-to-one relationship. Thus, the cardinality provides the information on the maximum number of relationships an object can hold with another object and can be classified as:

1. One-to-one relationship (1:1)
2. One-to-many relationship (1:M)
3. Many-to-many relationship (M:N)

However, the modality holds the information on whether two objects must enter into a relationship or not. In case it is mandatory to enter into a relationship, then the modality is 1. But in case the relationship is optional and the objects may not carry any relationship, then the modality is 0.

POINTS TO PONDER

Along with the relationship information of diffrent objects it is also important to capture how many of each object types can be attached with other objects, which is called Cardinality and wether it is mandatory to attach an object with the another object type which is called Modality.

There are several types of notations used for creating an ERD. The "information engineering" notations for different cardinality and modality representation are shown in Figure 3.10.

Refer to Figure 3.10 for examples of the modality depiction.

The cardinality and modality information are combined (Figure 3.11) to represent the relationship among the objects completely.

According to Figure 3.11, the mandatory (Modality) options are as follows.

It is clear 0 means there is no relationship and 1 means there is a relationship.

Insurance provider should have at least one insurance policy to be called as an insurance provider. An insurance policy may not be provided by all the insurance providers. It is optional for the insurance provider to provide a particular policy.

Now cardinality determines whether the relationship is

1. One-to-one relationship (1:1)
2. One-to-many relationship (1:M)
3. Many-to-many relationship (M:N)

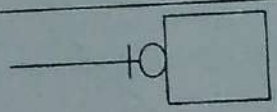
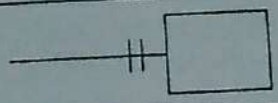
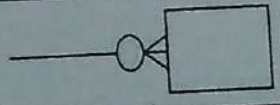
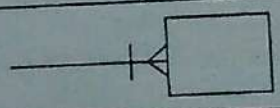
Multiplicity	Notation
Zero or One	
One Only	
Zero or More	
One or More	

Figure 3.10 Relationship Notations

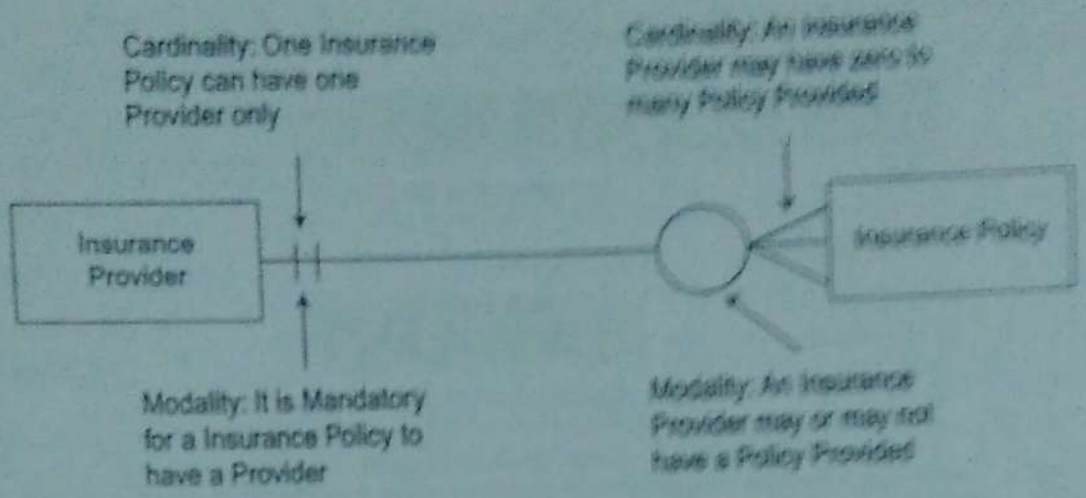


Figure 3.11 Cardinality and Modality Combined

3.2.1.3 Entity Relationship Diagrams

The relationships of the objects present in a system are represented graphically through the ERDs. The *entities* are represented as a rectangle. The relationships are represented with lines with the ends representing the cardinality and modality as shown in Figure 3.11. Thus the various components of an ERD are

- Data objects or entities
- Attributes
- Relationships and
- Various indicators describing the relationships

The ERD for a simplified insurance policy management system is depicted in Figure 3.12.

Here insurer, insurance provider, policy and claim are the objects.

An insurer should have at least one policy. An insurer can have more than one policy.

An insurance provider should issue at least one policy. A particular policy may be with zero or multiple insurance provider.

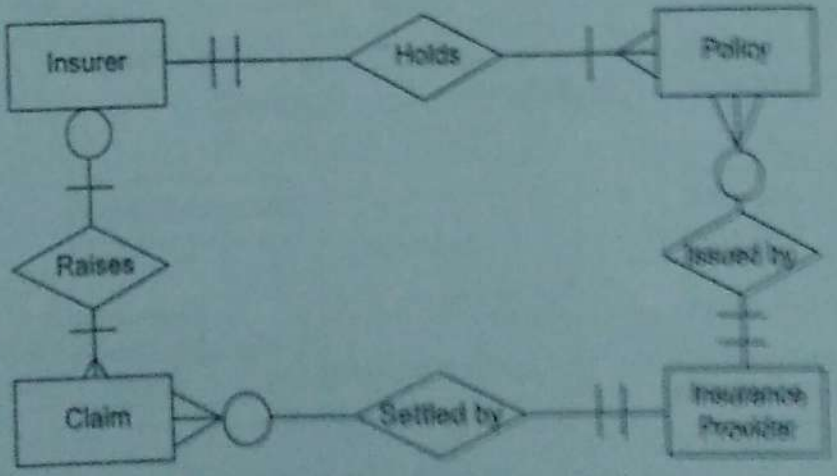


Figure 3.12 Requirement Management

3.2.1.4 Data Dictionary

Data dictionary is the place where the description and information about all data objects produced and consumed by the software system is maintained. Some of the attributes that normally gets stored inside a data dictionary are such as

Name	Each data, data store and external entity is identified with this primary name.
Alias	Each data object can have alternate names.
Where-used/how-used	Processes that use the data/control item and how it is used are represented. The use may be • as a store • input to process • output from process • as an external entity
Content description	This notation is used for representing content.
Supplementary information	Other information such as the preset values, restrictions, limitations, etc. is captured.

Few common notations used for describing data objects in a data dictionary are

=	is composed of
+	and
()	optional
{}	iteration
[|]	either - or
..text...	comment
{ }n	n repetitions of
(...)	delimits a comment

An example of data dictionary maintained for the customer of a banking system is as follows:

Customer Name = Title + First Name + (Middle Name) + Last Name
Title = [Mr.| Ms.| Mrs.]
Address = {Address Text}^{2} *... Home and Office Address..*

Along with the basic information that normally becomes part of the data dictionary, other information may optionally be stored inside it.

Below is a list of properties that can be stored inside a data dictionary item if these help in clarifying the data structure in the system.

- Name
- Aliases or synonyms
- Default label

- Description
- Source (s)
- Date of origin
- Users
- Programs in which used
- Change authorizations
- Access authorization
- Data type
- Length
- Units (cm., °C, etc.)
- Range of values
- Frequency of use
- Input/output/local
- Conditional values
- Parent structure
- Subsidiary structures
- Repetitive structures

3.2.2 Functional Modeling or Flow-oriented Modeling

Functional modeling or flow-oriented modeling (Figure 3.13) is the second model of structural analysis. In data modeling, ER diagram was discussed, and in functional modeling, data flow diagram (DFD) is discussed.

Functionality flow is also known as the data flow.

3.2.2.1 Data Flow Diagrams

DFD (Table 3.2) is a diagram of functional modeling. Entire system is shown as a process and data are interlinked to each other. Data flows between one object to the other.

These data get transformed (takes different form) as they flow through the software (Processes) (refer Figure 3.14).

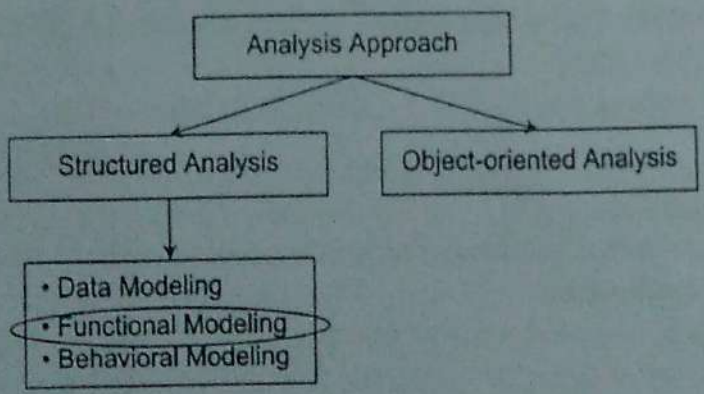


Figure 3.13 Structured Analysis, Functional Modeling

Table 3.2 Structured Analysis, Data Flow Diagram

Type	Corresponding Diagram
Data Modeling	ER Diagram
Functional Modeling	Data Flow Diagram
	Control Flow Diagram
Behavioral Modeling	State Transition Diagram

Any portion of the system can be depicted as a sub process that processes the input data, make some change to it or takes some decision based on it and produces the output data.

To clarify the concept of the data flow, think about a software program which calculates the insurance premium based on the information of the insurer. Figure 3.15 shows how the input data is processed by the software system to produce the desired output – the premium amount (called as premium information) and it is also stored (recorded) in "insurance policy record" for further analysis.

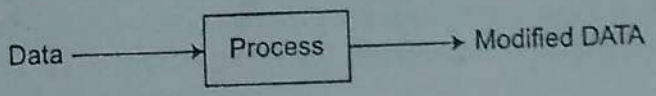


Figure 3.14 Data versus Process

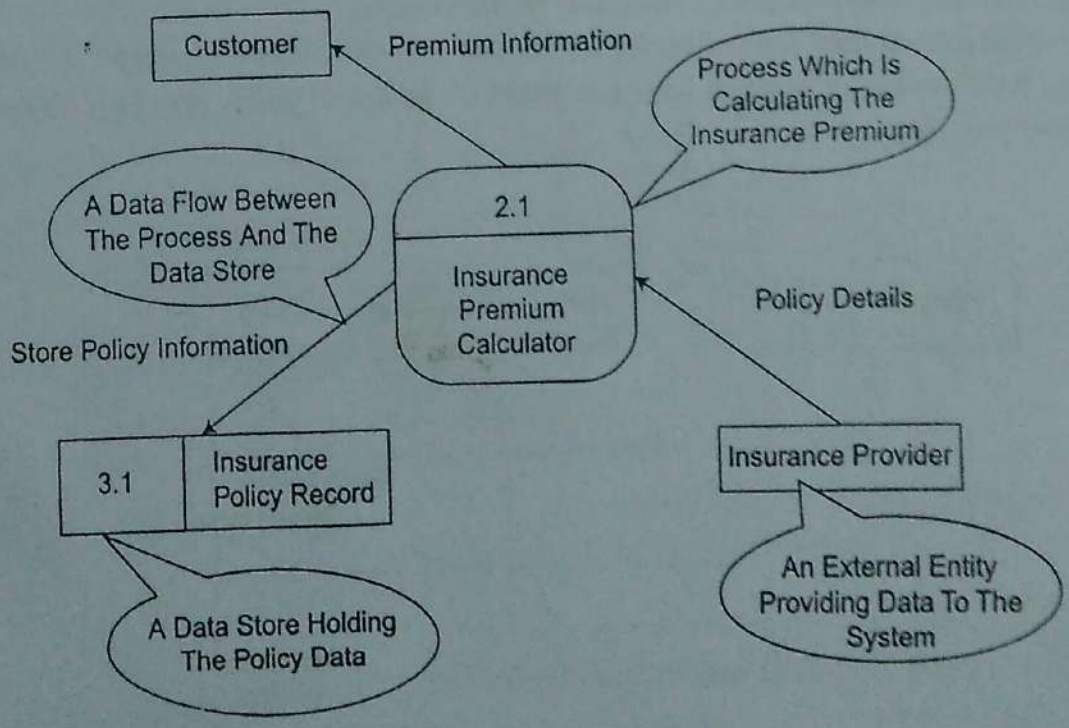


Figure 3.15 Level 1 DFD

DFDs demonstrate the relationships between the various components in a program or system. This modeling technique reveals the high-level details of a system by showing how input data is transformed to output results after processing through the functional modules.

The four main components (Figure 3.16) of a DFD modeling are

- Entities
- Processes
- Data stores and
- Data flows

Entities

The external systems which either provide data to the system or receive from it are called entities. An entity which provides data to the system is called a source and an entity which receives data from the system is called a sink (Figure 3.17).

Entities are represented as rectangles (Figure 3.18) with a name that clearly represents its kind. The entities are closely related to the identification of the system boundary.

Process

A process is the manipulation or work that transforms data while it flows through the system. A system can have one or multiple processes (Figure 3.19) that are interlinked and cause the data transformation through computations, logical decisions, or workflow branching based on business rules. A process receives some input and generates some output.

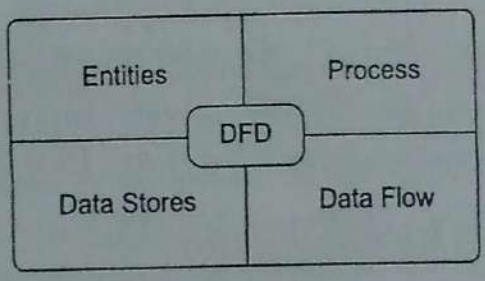


Figure 3.16 Components of DFD

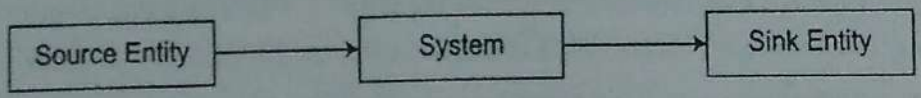


Figure 3.17 Source and Sink Entity

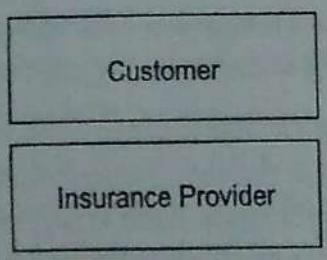


Figure 3.18 Entity Notations – Example

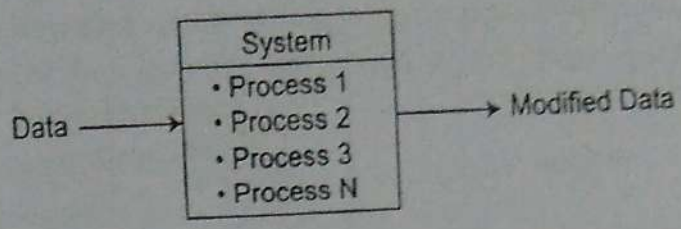


Figure 3.19 System versus Process

There are two kinds of notations being used in DFD (Figure 3.20).

1. DeMarco–Yourdon symbols
2. Gane–Sarson symbols

Their notations of process, data store, source/sink, dataflow are represented in Figure 3.20.

Processes (Figure 3.21) include a process name and process number. Mentioning the process names in a descriptive fashion, such as "calculate premium," "store in database," help conveying the purpose of the process to the DFD users.

Data Store

At the time of data processing the processes may store the data intermittently for future use. These stored data can be retrieved and used by the same process at later point of time or by a separate process. Files and tables are examples of types of data stores (Figure 3.22). Data stores are usually represented as shown in Figure 3.22.

Discussion Questions

1. Discuss and list down various data sources used in the college to run the college.

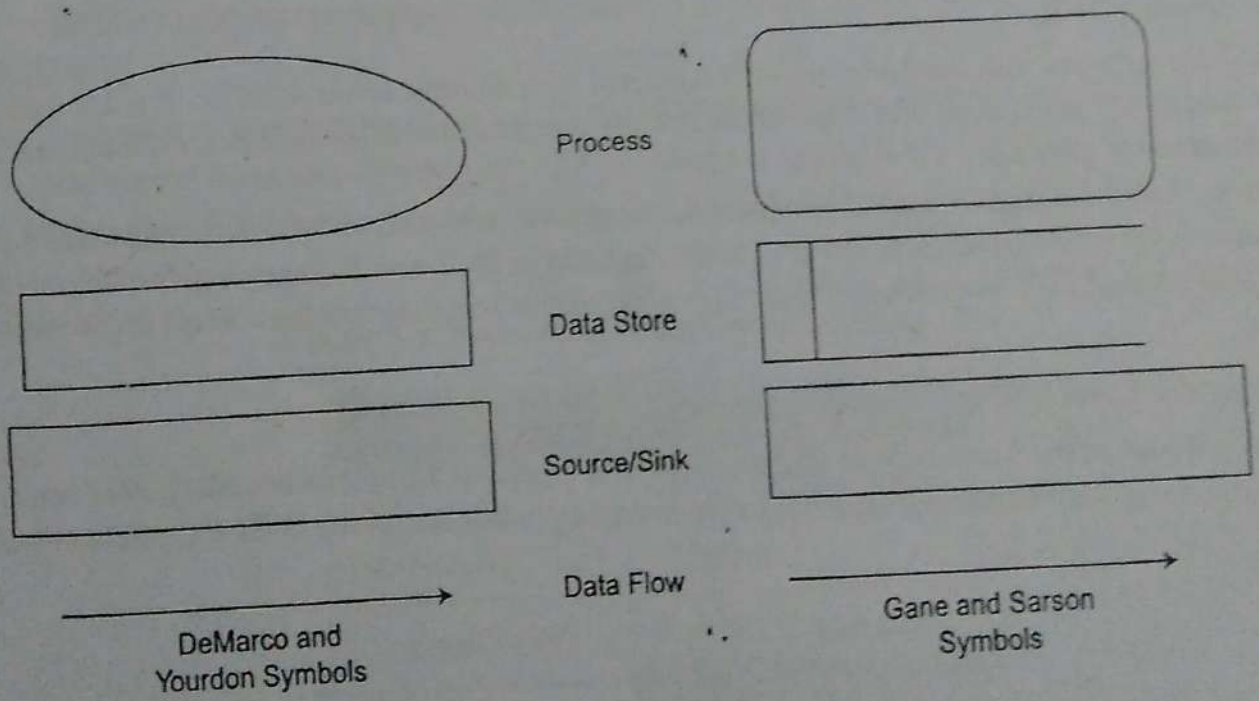


Figure 3.20 Data Flow Notations

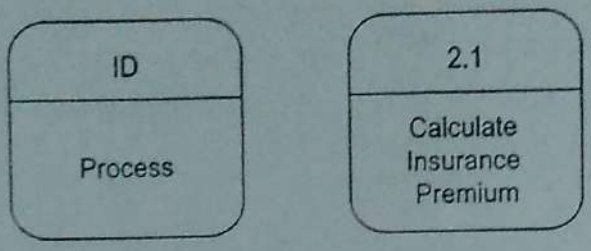


Figure 3.21 Process Notations

Figure 3.22 Data Source Notation

Data Flow

All the above three parts are connected to a DFD using data flow. Data flow (Figure 3.23) demonstrates the movement of data between the entity, the process and the data store revealing how the components of the DFD interface with each other. Arrow is used to represent data flow and the data name is written on top of it and should be unique within a specific DFD.

Difference between Flow Diagram and DFD

The arrows in DFD show that the data is being flown from one component to the other, and it does not specify whether the data is being pushed by the component. Components in DFD do not describe whether data will be executed by the components. Data in DFD can go through multiple components via multiple connections.

How to create a DFD

The DFD for a system may get progressively elaborated with multiple levels of DFDs. It starts with a context diagram (Figure 3.24) where the entire system is represented as a single process and associated with external entities.

Then the Level 0 diagram is created which shows general processes and the data stores. Then in subsequent level of DFDs each of the processes depicted in the Level 0 diagram gets elaborated. The highest level of DFD is generally called as context diagram as it depicts the context of the entire system.

Developing the Level 0 DFD

The main purpose of the Level 0 DFD (Figure 3.25) is to represent the system boundary. All components inside the system boundary are included in a single process box in the DFD. External entities

Figure 3.23 Data Flow Notations

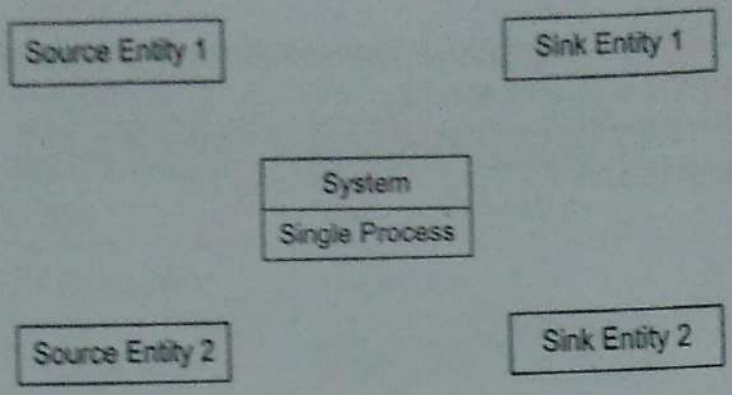


Figure 3.24 Context Diagram

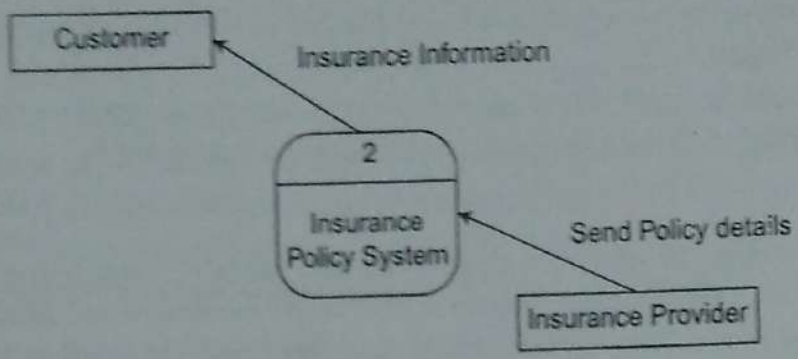


Figure 3.25 Level 0 DFD

are shown and the arrows to and from the external entities show the system's relationship with its environment. It is not required to show the data flow happening between the external entities, but if depicting such a data flow helps in adding clarity to the overall DFD then such interactions can be shown using hashed lines.

Level 0 DFS shows the data stores of the system clearly.

Child (Level 1+) Diagrams

Each of the sub-processes is then elaborated in subsequent Child DFDs. Refer Figure 3.26 where the insurance premium calculator is elaborated as one of the Child DFD. The symbol and notations are same as Level 0 diagram, but more information is added about the process in this level.

Some Good Practices While Creating the DFD

DFDs act as the principle source of information for the design team and thus, the volume of information and correctness are very importantly to be ensured. There is no right or wrong way to create a DFD, but definitely there are efficient and inefficient DFDs. Some guidelines for creating efficient DFDs and some caveats to avoid the inefficient ones are as follows:

1. Discover all the data items: This is the first and the most important step for creating a DFD. These data items may come out from the requirement document or get unearthed during the later stages while analyzing the requirement. Each and every data item has some origin (from where they come) and a target (where they go). Create a table to document this information in the below-mentioned format.

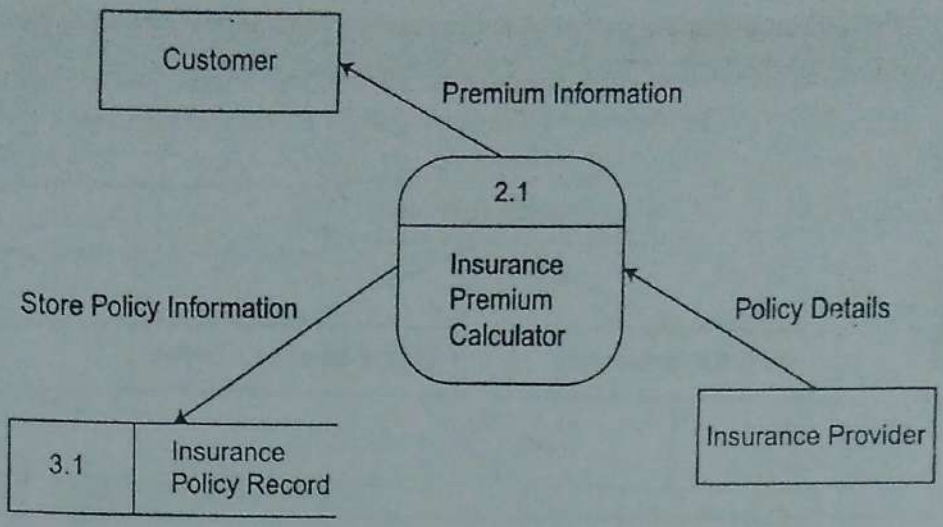


Figure 3.26 Level 1 DFD

Data Item	Origin	Target

2. Determine the system boundaries: Analyze the table created in the above step for each of the data items and find out whether it belongs to internal system or outside the system. This defines the system boundary.
3. Develop Level 0 DFD: At this stage, all the data items inside the system boundary should be contained in one DFD and connected with others. The interaction of data items external to the system with those inside the system needs to be shown as well this defines how the system interacts with its environment. All the interactions crossing the system boundaries should be captured in Level 0 DFD before moving to the lower level blow ups. There should not be any new data flow crossing the system boundary in child level figures.
4. Develop child (Level 1+) level DFD: The aim of the first level DFD is to provide a high-level view of the system which has all the major processes and data stores identified. List down all the incoming and outgoing data flows so that the corresponding sources and destinations can be found. There may be some data stores shared by multiple processes. Understand that creating of the DFD is an iterative process – each level of DFD can be refined till all the desired information is part of it. Hence, more granular level DFDs may be elaborated based on the complexity of the system.

Some of the best practices that can be taken into consideration while creating the DFDs are mentioned below. This also provides guidelines on what a valid and non-valid DFD is based on.

1. There should not be any data flow directly from one data source to another. This is because the data stores are simply the storage area and cannot initiate a communication. In between there should mandatorily be one or more processes which initiate the process flow.
2. Entities are people or systems outside the system boundary. The data from the entities may not be matching the syntax of the data stored in the data stores inside the system boundaries. Thus, it is important to avoid the direct data flow between the entity and the data store.
3. If the DFD contains the data flow between the entities then it will be difficult for the system to validate the same as the entities are outside the system boundary. Model only the flows which the system is expected to now or react to.

4. If there are two processes in the system which are not running simultaneously, then it is not valid to show a data flow between them. This is because the processes have no memory and cannot store any intermediate results.
5. There are 3 types of valid data flows:
   (a) Between a process and an entity
   (b) Between a process and a data store
   (c) Between two processes that run simultaneously
6. If there is a process which has only input data flow then it is an invalid process and is called a Black Hole
7. Similarly a process which has only output flow is not valid and is called a Miracle
8. Validate whether there is sufficient input data flow to produce the intended output data flow. A process with insufficient data input for the predicted output is called a Grey Hole.

3.2.2.2 Control Flow Model

Data flow diagram which is used to show the data flow in a system is already discussed. There are systems where the flow is controlled by events rather than by data. A control flow diagram (Table 3.3) is used to represent such systems.

These diagrams are generally time dependent. The flow of controls activates some processes. Here the model contains same processes as the DFD, but instead of showing the data flow here, the control flow is shown.

The concept of Control Specification (CSPEC) is introduced which is like a window that controls the processes represented in the DFD based on the event that is passed through the window. Below is an example where the control flow and the CSPEC are shown (Figure 3.27).

The inflow of water is stopped once the water level in the tank reaches the safety mark. Note that the control flow is represented with dotted lines. And the CSPEC is represented with a solid bar (Figure 3.28).

A control model and a process model are connected through data condition. Data condition occurs when data input to a process causes a control output. The simple guideline for creating a CFD is discussed below.

1. Draw the data flow diagram and then get rid of all the data flow arrows.
2. Add control items and windows into CSPEC wherever applicable.
3. Create the CFD from the Level 0 and elaborate into more granular child CFDs.

Table 3.3 Structured Analysis, Control Flow Diagram

Structured Analysis	
Type	**Corresponding Diagram**
Data Modeling	ER Diagram
Functional Modeling	Data Flow Diagram
	Control Flow Diagram
Behavioral Modeling	State Transition Diagram

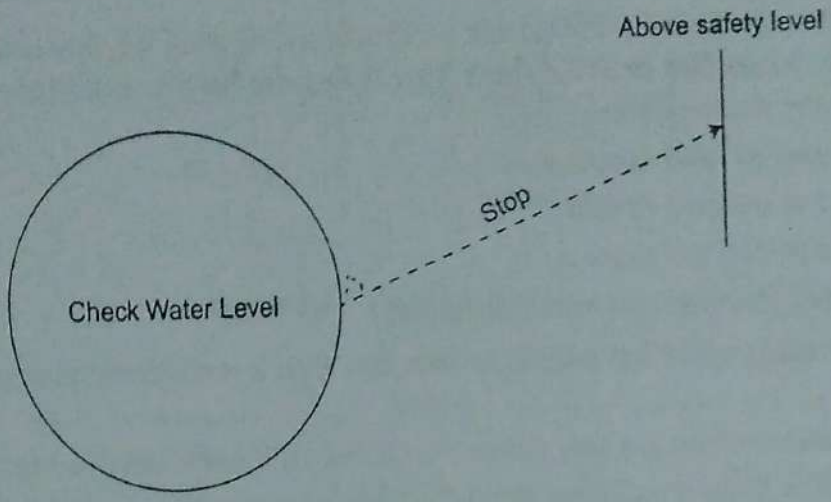


Figure 3.27 Control Flow Example

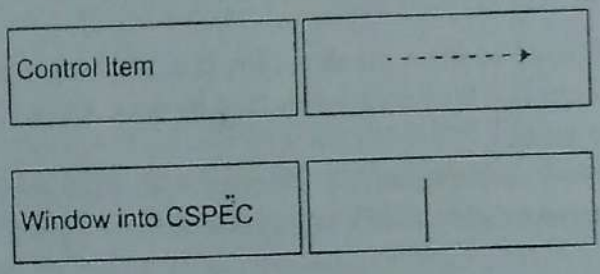


Figure 3.28 Control Flow Notations

Discussion Questions

1. Discuss the control flow of a road traffic signal system.

3.2.3 Behavioral Modeling

Behavioral modeling (Figure 3.29) is the third type of structured analysis approach.

The type of analysis models which try to capture the change in behavior of the system as an effect of some event or trigger are grouped as the behavioral models. State transition diagram (STD) is one example of behavioral modeling.

3.2.3.1 State Transition Diagram (STD)

STDs (Table 3.4) represent the system states and events that trigger state transitions or state change.

The actions (e.g., activation of some process) as a result of triggers in the system are captured in this model. A state can be an observable mode of behavior. STDs are created for objects which have significant dynamic behavior. For example, the customer entry form of the banking system may add a new customer where the customer's status is "open". If the customer fails to maintain the minimum

balance for a considerable time then the account status may change to "suspended." See Figure 3.30 where the state transition for the banking system is elaborated.

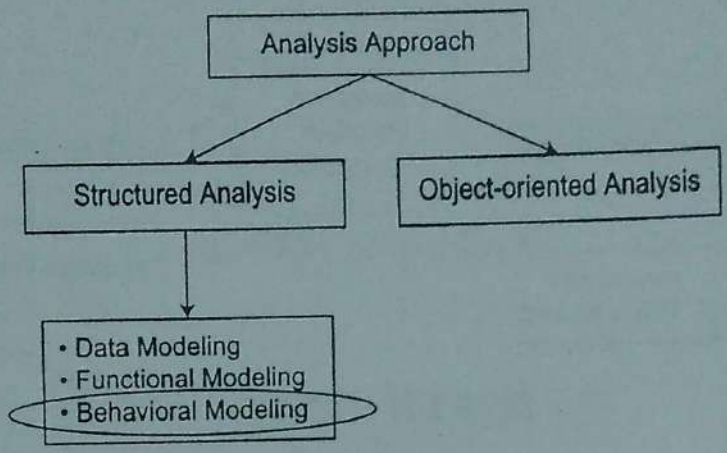


Figure 3.29 Structured Analysis, Behavioral Modeling

Table 3.4 Structured Analysis, State Transition Diagram

Structured Analysis	
Type	**Corresponding Diagram**
Data Modeling	ER Diagram
Functional Modeling	Data Flow Diagram
	Control Flow Diagram
Behavioral Modeling	State Transition Diagram

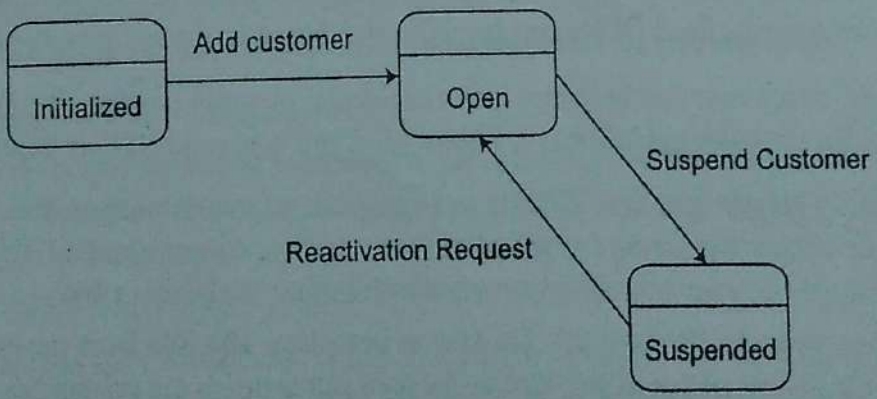


Figure 3.30 State Transition Diagram

Discussion Question

List down various state transition status in a college student life from morning till evening.

1. _____
2. _____
3. _____
4. _____

3.3 OBJECT-ORIENTED ANALYSIS

In an object-oriented analysis the system is depicted as a group of interacting objects. The attributes of the objects are represented by its class, the state and the behavior. With modern tools and techniques which support creation of object-oriented analysis, these analysis techniques have become very popular. The details of the Object-oriented analysis and design models are discussed in detail in the subsequent chapters.

Summary

The domain of structured analysis comprises three modeling approaches:

- Data modeling
- Functional or flow-oriented modeling and
- Behavioral modeling

Structured Analysis	
Type	Corresponding Diagram
Data Modeling	ER Diagram
Functional Modeling	Data Flow Diagram
	Control Flow Diagram
Behavioral Modeling	State Transition Diagram

- Entity relation diagram is a form of data modeling technique. It diagrammatically shows the relationship between different objects inside a system.
- Data flow diagram is a highly effective model which is used for showing the flow of information through a system. DFDs can be used in the early stages of systems analysis to help understand the current system and to represent a required system. The DFDs represent external entities sending and receiving information (called entities), the processes that change information (called processes), the information flows (called data flows), and the place where the information is stored (called data stores). Level 0 or the context diagram is the top level diagram which can be decomposed into lower level diagrams (Level 1, Level 2...Level N) to represent different areas of the system.
- STD is an example of the behavioral modeling technique. It shows the system states and events that trigger state transitions or state change.

Model Questions

Part A (Objective type)

1. Providing a graphical representation of the system behavior which is depicted as a combination of functional flow and data flow to create the input for the design phase is called

   A. Analysis modeling
   B. Data modeling
   C. Functional modeling
   D. Behavioral modeling

Answer: A

2. The two most popular approaches to software analysis, which have developed and got fine-tuned over the years, are structured analysis and object-oriented analysis.

   A. True
   B. False

Answer: A

3. What are the notations to capture the control and behavioral aspects of the real time engineering problems along with the information systems applications?

   A. Analysis modeling
   B. Data modeling
   C. Functional-oriented analysis
   D. Behavioral modeling

Answer: C

4. All of the following are examples of structured analysis except?

   A. Data modeling
   B. Functional or flow-oriented modeling
   C. Behavioral modeling
   D. Design analysis

Answer: D

5. Which is an analysis technique that deals with the data processing part of an application?

   A. Analysis modeling
   B. Data modeling
   C. Functional modeling
   D. Behavioral modeling

Answer: B

6. This is the identity which has multiple attributes and is related to other entities in the system.

   A. Flow object
   B. Attributes object
   C. Data object
   D. Information object

Answer: C

7. Along with the relationship information of different objects, it is also important to capture how many types of each object can be attached with other objects. This is called

   A. Modality
   B. Cardinality
   C. Modularity
   D. Attach ability

Answer: B

8. Along with the relationship information of different objects, it is also important to know whether it is mandatory to attach an object with another object. This is called

   A. Modality
   B. Cardinality
   C. Modularity
   D. Attachability

Answer: A

Section 3
Design and Architectural Engineering

4

Design and Architectural Engineering

CHAPTER COVERAGE

4.1 DESIGN PROCESS AND CONCEPTS

During the design process, the software requirements model (discussed in the previous chapters), which was prepared for the requirements is converted into suitable and appropriate design models (Figure 4.1) that describe the architecture and other design components.

Each design product is verified and validated before moving to the next phase of software development.

In the requirements analysis stage, the decision about implementation is consciously avoided. In the design stage, the decision about implementation is consciously made and a step-by-step procedure is followed.

The design process encompasses a sequence of activities that slowly reduces the level of abstraction with which the software is represented.

Design activities are subdivided into high-level design activities and low-level (or detailed) design activities. Architecture is the outcome of high-level design activities, whereas the data structure and the corresponding algorithms are the outcomes of low-level design activities.

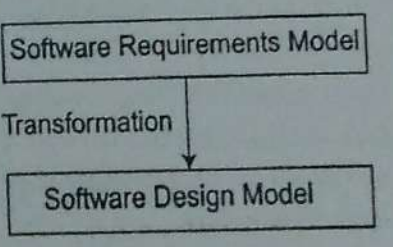


Figure 4.1 Software Design Model

4.2 BASIC ISSUES IN SOFTWARE DESIGN

The following are the basic issues in software design:

1. If the requirements model is wrongly represented then the design model also will go wrong.
2. Mapping each functional requirement into design representation is difficult.
3. Mapping all the requirements into a single design model is a tedious job.
4. Modeling a design that enables easy coding as well as catering to all the requirements is difficult.
5. Requirements keep changing and evolve over a period of time, but changing the design accordingly is difficult.
6. Balancing simplicity and efficiency is difficult.
7. Balancing clarity and code safety is difficult.
8. Balancing the robust design and the system response time is difficult.
9. Following standards in design and simultaneously catering to and matching the requirements is difficult.
10. Error function handling is also a difficult part of design.

4.3 CHARACTERISTICS OF A GOOD DESIGN

1. Should be able to convert all the requirements into design
2. Should be easily understandable and maintainable
3. Should be easy to change the design
4. Should be easily scalable

4.4 SOFTWARE DESIGN AND SOFTWARE ENGINEERING

A software design is an appropriate engineering representation of a software product under study and is the process of problem solving by converting the requirements of the product into a software solution (Figure 4.1). Traceability between the requirements and design is important so that it is possible to independently assess the design for requirements mapping. Software design should also be assessed for its own quality parameters such as robustness, flexibility, security, and design clarity. Software design can be considered as a plan for coding.

Software engineering comprises strict disciplines that need to be followed to develop a programmable solution to solve the problems of customers. Strategies are defined to obtain a runable, efficient, and maintainable software solution that takes care of the costs, quality, resources, and other expenditure. These strategies have been proposed by various engineers after implementing and analyzing the merits and demerits. It includes disciplines related to requirements collection, requirements analysis, designing, coding (development), testing the developed code, operation, and maintenance of the complete software.

4.5 FUNCTION-ORIENTED SYSTEM VS OBJECT-ORIENTED SYSTEM

The design of function-oriented system is called function-oriented design and the design of object-oriented system is called object-oriented design (Figure 4.2).

Since the function-oriented system is made up of functions, the function oriented design and the corresponding design structure is also based on functions. Data are stored outside the system in a

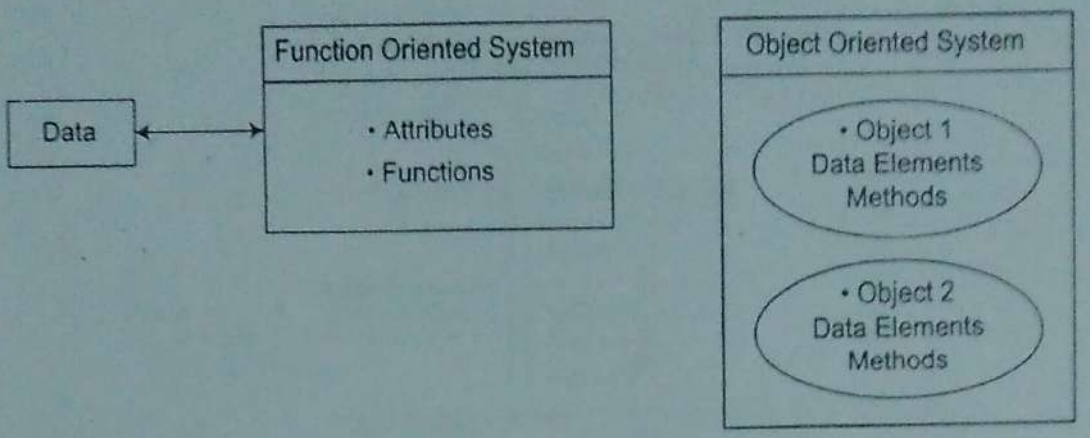


Figure 4.2 Function-Oriented System vs Object-Oriented System

function-oriented system. Functions are used to access, process, modify the data and store it back into the data storage system. The system state is maintained in the data. Functions are easily mapped to modules (subsystems).

Object-oriented design is the design concept of an object-oriented system, which is completely made up of objects. Objects have data and functions inside them. In this system also, functions (inside objects) are used to access the data (inside objects). We will be discussing the object-oriented concepts in detail in Chapter 5.

4.6 MODULARITY, COHESION, COUPLING, LAYERING

The possibility of dividing a single project into smaller units called modules is termed modularity (Figure 4.3) of the project. Modularity helps in designing the system in a better way and ensures that each module communicates and does the specific task assigned to it. It also helps in reusability and maintainability.

The two main characteristics that need to be considered while designing the modularity of projects are cohesion and coupling.

Cohesion refers to "how strongly" one module is related to the other, which helps to group similar items together. Cohesion measures the semantic strength (high and low cohesion) of relationships between modules within a functional unit. Internal cohesion of a module is the strength of the elements within the modules, whereas external cohesion of a module is the strength of relationships between modules (Figure 4.4).

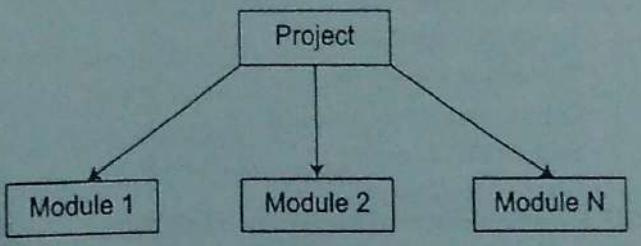


Figure 4.3 Modularity

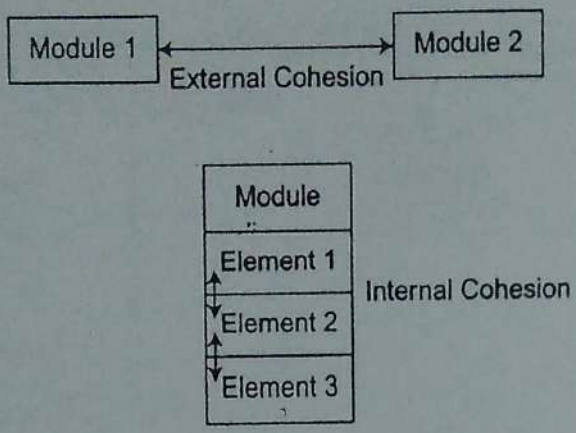


Figure 4.4 External vs Internal Cohesion

POINTS TO PONDER

Cohesion refers to "how strongly" one module is related to the other, which helps to group similar items together. Cohesion measures the semantic strength (high and low cohesion) of relationships between modules within a functional unit.

In a highly cohesive system code readability and reusability increases, and complexity is kept at a manageable level.

There are several types of cohesion, which are discussed below (Figure 4.5):

- Coincidental cohesion
- Logical cohesion
- Temporal cohesion
- Procedural cohesion
- Communicational cohesion
- Sequential cohesion
- Functional cohesion

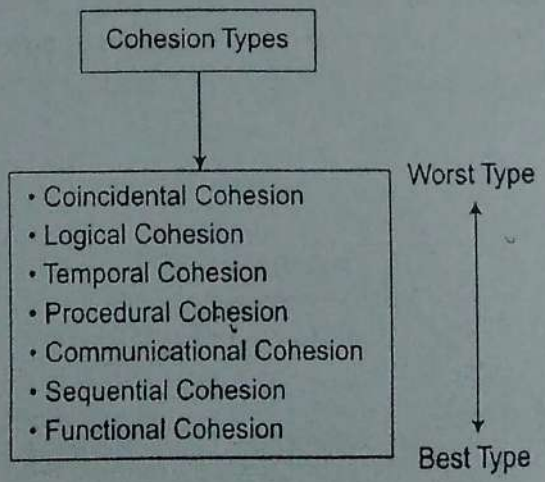


Figure 4.5 Cohesion Types

Coincidental cohesion: As the name indicates, in this type the cohesion (grouping) occurs coincidently. Elements or modules are grouped together for different purposes and there is no common reason for grouping. The only common thing is that they are grouped together.

Logical cohesion: In this type, as the name indicates, cohesion (grouping) occurs logically based on similarity and not based on functionality. Elements or modules are grouped together based on common logic. Logical grouping may be based on the functionality, nature, and behavior of the elements/modules.

Temporal cohesion: In this type, cohesion (grouping) occurs during run time at a particular time of program execution. Modules that are being called during the exception handling procedure can be grouped together. Modules that are being executed at time T, T + X, or T + 2X can be grouped together.

Procedural cohesion: In this type, cohesion (grouping) is used to execute a certain procedure. Modules that are used for a procedure can be grouped together. Note that the same module can be grouped together with another module for executing another procedure.

Communicational cohesion: In this type, cohesion (grouping) occurs because part of the module shares same data (acts on same data) for communication purposes. For example, functions A, B, and C access common data (same database). Modules A, B, and C access a common database.

Sequential cohesion: In this type, cohesion (grouping) occurs as the output of one module is an input of another module. This will look like an assembly line sequence.

Functional cohesion: In this type, cohesion (grouping) occurs as all the functions contribute to a single task of the module. In order to execute a single task, all modules contribute. Within a function, each part is executed in order.

Coupling is a measure of "how tightly" two entities or modules are related to each other. Coupling measures the strength of the relationships between entities or modules. It is very easy to measure coupling both quantitatively and qualitatively.

Figure 4.6 shows the difference between loosely coupled and highly coupled modules.

POINTS TO PONDER

Coupling is a measure of "how tightly" two entities or modules are related to each other. Coupling measures the strength of the relationships between entities or modules. Coupling is very easy to measure quantitatively and qualitatively.

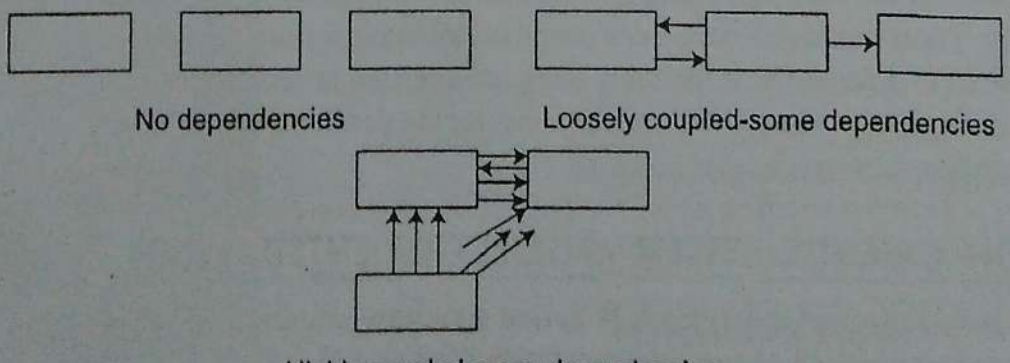


Figure 4.6 Coupling

Minimize coupling and maximize cohesion is the principle of design. This means that a module should be as independent as possible.

There are different types of coupling, which are discussed below (Figure 4.7):

- Content coupling
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling
- Uncoupled

In content coupling, one component refers the content of another component and here the coupling is considered to be higher. For example, Module 1 accesses the data of Module 2 and changes it (Figure 4.8).

In common coupling, one or more modules share the same data which is common. This is an example of bad design and should be avoided (Figure 4.9).

In control coupling, a module passes its control component to another module. Depending on the control being passed, it is considered as good or bad. Module 1 calls Module 2 and Module 2 passes the flag that indicates a status (Figure 4.10).

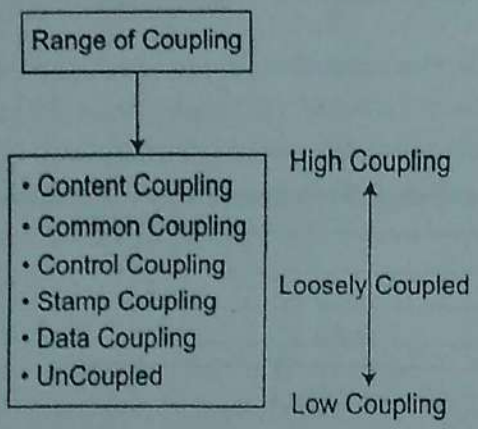


Figure 4.7 Types of Coupling

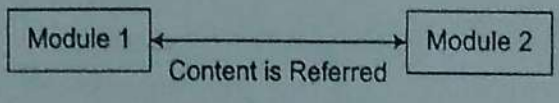


Figure 4.8 Content Coupling

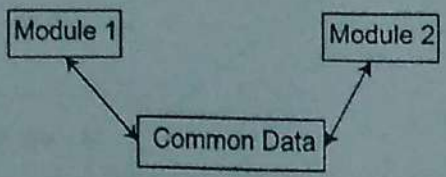


Figure 4.9 Common Coupling

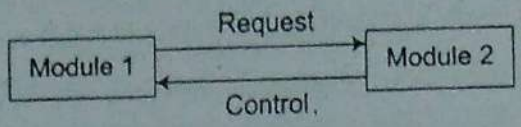


Figure 4.10 Control Coupling

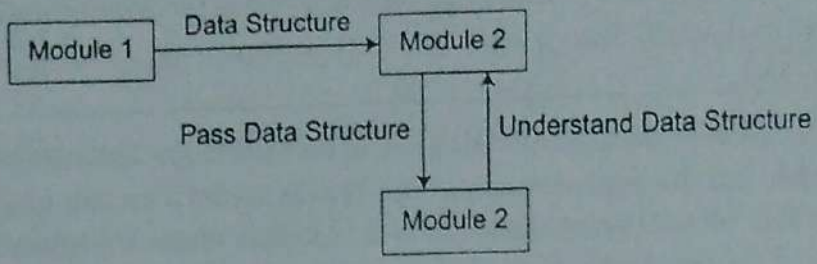


Figure 4.11 Stamp Coupling

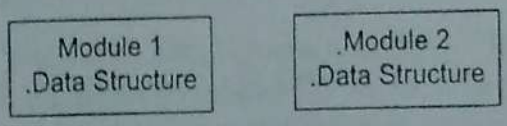


Figure 4.12 Data Coupling

In stamp coupling, a module passes a data structure to a module, which does not have access to the entire data structure. The module sends the data structure to another module to understand it. The module does not have any power and just acts as a stamp and is therefore called stamp coupling (Figure 4.11).

Two modules are data coupled, if there are homogeneous data items or structures (Figure 4.12).

POINTS TO PONDER

Scope of Effect and Scope of Control: The scope of effect of a module refers to the other modules that are affected by a decision made by a particular module. Note: The module can be at any level, that is, it can be at the same level, higher level, or lower level.

The scope of control of a module refers to that module and all of its subordinate modules belonging to that particular module.

4.7 REAL-TIME SOFTWARE DESIGN (RTS DESIGN)

Real time refers to designing the software systems whose behavior is subject to timing constraints and is embedded in a large hardware system. This kind of software monitors and controls the system and its environment by gathering different sets of data using sensors. Actuators act in the opposite way and they change the environment of the system.

A microwave oven has a functionality to stop the cooking after a specified time period. In this case, the sensor continuously monitors the time and the actuator stops the control of the microwave oven after a specified time period. Burglar alarm is another classic example of real-time software.

POINTS TO PONDER

Real time refers to designing the software systems whose behavior is subject to timing constraints and is embedded in a large hardware system. This kind of software monitors and controls the system and its environment by gathering different sets of data using sensors. Actuators acts in the opposite way and they change the environment of the system.

4.8 DESIGN MODELS

The software engineer creates a design model, the end user develops a mental image that is often called the user's model, and the implementers of the system create a system image. These models differ drastically. The role of an interface designer is to reconcile these differences and derive a consistent representation of the interface.

Figure 4.13 shows the design models of an entire system.

4.8.1 Data Design

In data design, a high-level model is depicted (drawn) based on the user's view of the data cr information. Data design is actually derived from the ER diagram (refer to Structural Analysis – Data Modeling in Chapter 3). Designing to correctly fit the data structure is essential for creating a high-quality application. Transforming this data model into a database structure is critical for implementing the business requirements. The data warehouse structure, if any, is also designed in this step.

Database is the center of this design. Analysis that is applicable to a function, is also to be applied on the data. Data dictionary (information about data) is also established here.

4.8.2 Architectural Design

The objective of software design is to derive an architectural design (diagrammatic representation) of a system and this representation provides a framework from which more detailed design activities are conducted. Architectural design is derived from requirements analysis and is also based on already available architectural patterns and styles. It defines the physical and logical relationships between the major structural elements of a system and also between the corresponding components at a high level.

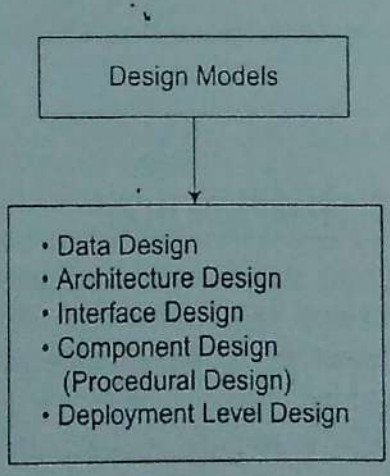


Figure 4.13 Design Models

4.8.2.1 What Is Software Architecture?

Software architecture defines the logical relationships between the major structural elements of a system and also the relationship between the corresponding components at a high level to execute the functionalities of the overall system.

4.8.2.2 Software Architecture vs System Architecture

The representation of software architecture in physical components of machines is called system architecture.

4.8.2.3 Architectural Notations

To represent the architectural design, different notations are used to represent the flow of control hierarchy. Depth indicates the number of levels of control and width represents the overall span of control (Figure 4.14).

Fan-in indicates the number of modules that directly control a given module. In Figure 4.15, Module 1 is controlled by three other modules and therefore fan-in of Module 1 is 3.

Fan-out is a measure of the number of modules that are directly controlled by another module. In Figure 4.16, Module 1 controls three different modules and therefore fan-out of Module 1 is 3.

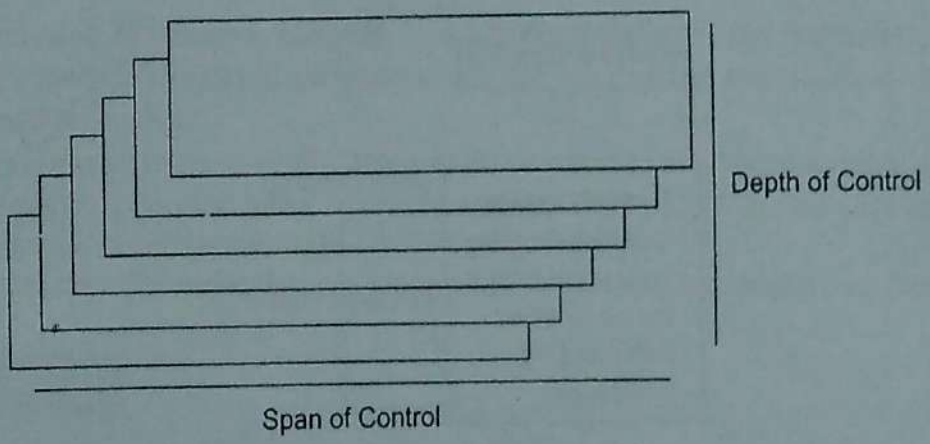


Figure 4.14 Level and Span of Control

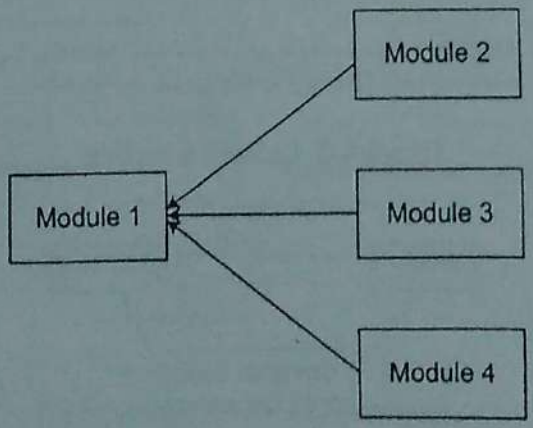


Figure 4.15 Fan-in Notation

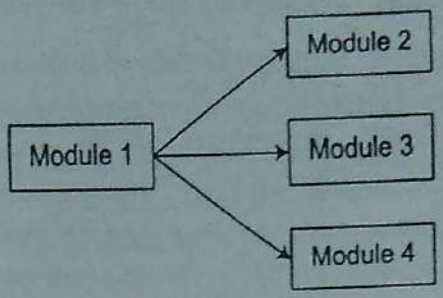


Figure 4.16 Fan-out Notation

A pipe-and-filter pattern (Figure 4.17) has a set of components called filters and they are connected by pipes, which transmit data from one component to the next.

A data store (Figure 4.18) resides at the center of this architecture and is accessed frequently by other components.

POINTS TO PONDER

Fan-out is a measure of the number of modules that are directly controlled by another module.

4.8.2.4 Mapping Requirements into Architecture

Requirements (data flow and control flow) are transformed into design using the architectural diagram. This diagram establishes the information flow between various components. There are two types of information flow (or data flow), namely, transform flow and transaction flow. In the transform flow, the data flow occurs sequentially and in straight line paths, whereas one data triggers the flow of data in one or various paths in the transaction flow.

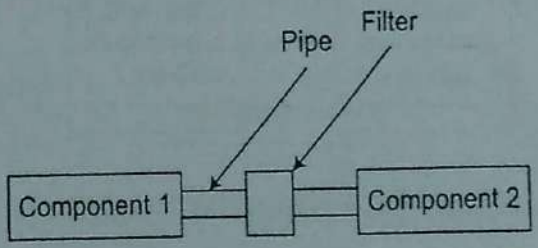


Figure 4.17 Pipe and Filter

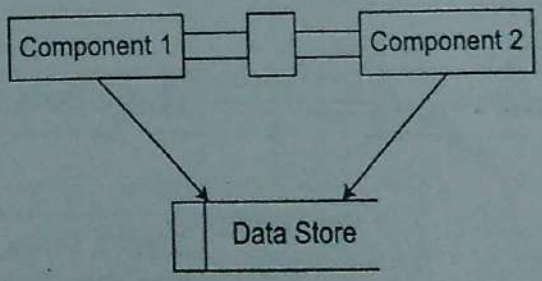


Figure 4.18 Data Store Representation

Design Heuristics Design heuristics are used in both transform and transaction mapping. First, let us look into design heuristics before getting into the details of transform and transaction mapping.

Heuristic is a Greek word that means "to find or to discover," indicating that it is an experience-based technique and is simply a thumb rule or an intuitive judgment. Design heuristic is a heuristic for the design – which is purely based on previous experience.

The following are the simple and basic design heuristics:

- Evaluate the design first using the principle "minimize coupling and maximize cohesion"
- Minimize a high fan-out structure (refer to Fan-in and Fan-out structures given in this chapter)
- Move to a fan-in structure as depth increases (refer to Fan-in and Fan-out structures given in this chapter)
- Scope of effect of a module should be within the scope of control of that particular module (refer to Scope of Effect and Scope of Control given in this chapter)

Transform Mapping Transform mapping has several steps, which are as follows (Figure 4.19):

Step 1: Analyze and refine the existing DFD (DFD is a functional model structure which was discussed in Chapter 3 in detail)

Step 2: Find out the transform and transaction characteristics of DFD

Step 3: Try to isolate the transform center by providing input, output, and boundary conditions

Step 4: After factoring (minimum of two levels), use design heuristics to improve the mapping

Transaction Mapping The steps in transaction mapping are similar to transform mapping, except that in Step 3, we try to isolate the transaction center instead of the transform center which are as follows (Figure 4.20):

Step 1: Analyze and refine the existing DFD

Step 2: Find out the transform and transaction characteristics of the DFD

Step 3: Try to isolate the transaction center

Step 4: After factoring (minimum of two levels), use design heuristics to improve the mapping

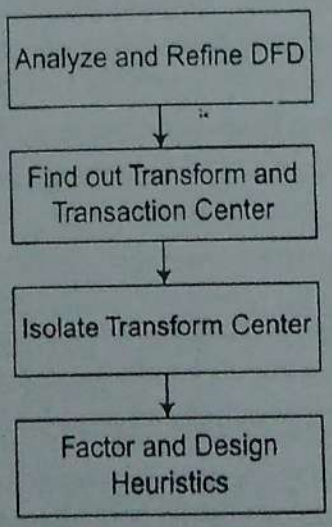


Figure 4.19 Transform Mapping

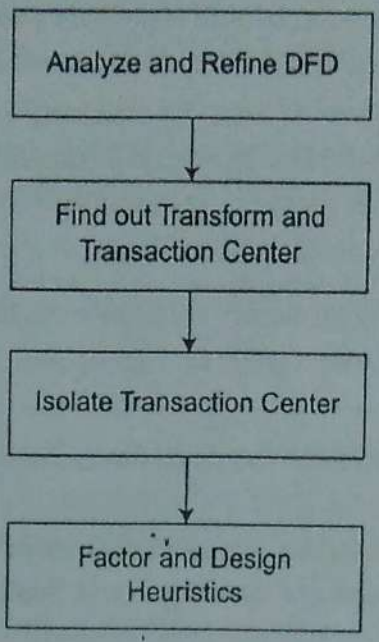


Figure 4.20 Transaction Mapping

Refining Architectural Design After carrying out transform and transaction mapping, the architecture can be further refined using the following steps (not inclusive):

- Define a processing account for every component of the architecture
- Develop an interface description for every component of the architecture
- Define design limitations and boundaries clearly
- Conduct a set of design reviews
- Make further refinement if needed

Refinement of software architecture during the early stages of design is encouraged. Alternative architectural styles may be derived, refined, and evaluated to obtain the "best" approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture. It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of modules that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

Assessing Alternative Architectural Design Each decision has two outputs. While representing the project in architectural design at one position, the design changes and an alternative design is needed.

4.8.2.5 Distributed System Architecture

Distributed System A distributed system has three main components, namely, data, process, and interface. If these components are distributed across locations, it is called a distributed system. A computer network connects the components which are at different locations.

Centralized Structure In a centralized structure, the server (data and process components) resides in a centralized place. The interface component (called client) is distributed across locations. When the number of clients increases, the load of the server also increases (Figure 4.21).

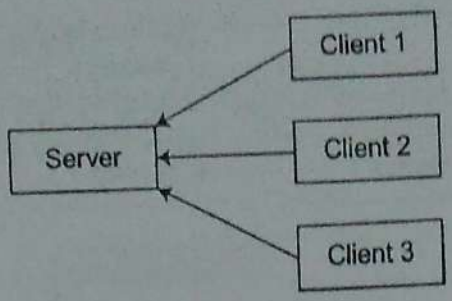


Figure 4.21 Centralized Structure

Decentralized Structure In a decentralized structure, both the client and the server reside in the same machine and they also communicate with another machine to share the resources (server and client components), which is also called peer-to-peer communication. Both the computers (system) perform the same functionality, which is called symmetric functionality (Figure 4.22). The connection between one computer (system) and another computer (system) is unstructured.

Hybrid Structure A combination of the centralized structure and the decentralized structure is called a hybrid structure (Figure 4.23).

POINTS TO PONDER

A distributed system has three main components, namely, data, process, and interface. If these components are distributed across locations, it is called a distributed system. The three types of systems are centralized, decentralized, and hybrid.

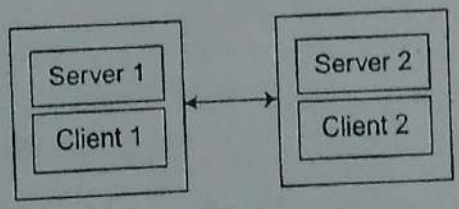


Figure 4.22 Decentralized Structure

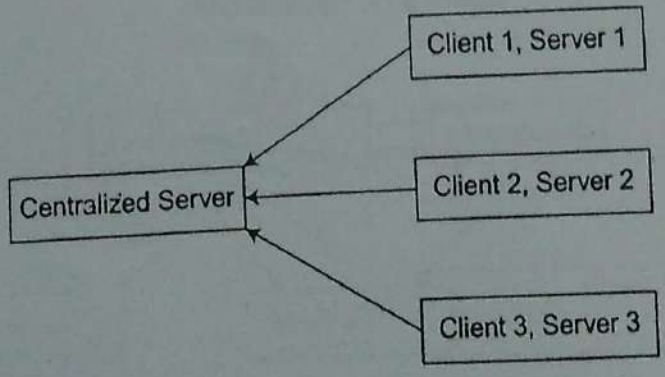


Figure 4.23 Hybrid Structure

4.8.2.6 Top-Down Approach and Bottom-Up Approach of Design

The top-down approach is represented by the hierarchical structure. It starts from the highest level to the lowest level in the hierarchy structure. This approach identifies the major components of the system and divides the components into lower level components. The root module is called the main module and the lower level components are called the subordinate modules. The top-down approach describes the organization and interaction of software components (Figure 4.24).

The bottom-up approach is just the opposite of the top-down approach and it starts with the basic lower level components of the system and proceeds to the higher level components (Figure 4.25).

4.8.2.7 Modular Design Approach

In this approach, the overall system is divided into modules (subsystems), which can be tested separately and can be combined (integrated) to form the overall system at a later point in time (Figure 4.26). We discussed this approach while discussing coupling and cohesion.

A modular system has the following characteristics:

- Each module will act as a separate system
- Each module is scalable and reusable
- Modules (subsystems) can be integrated easily to form the overall system
- Industry standards will be used in all modules
- Interface is consistent across the modules

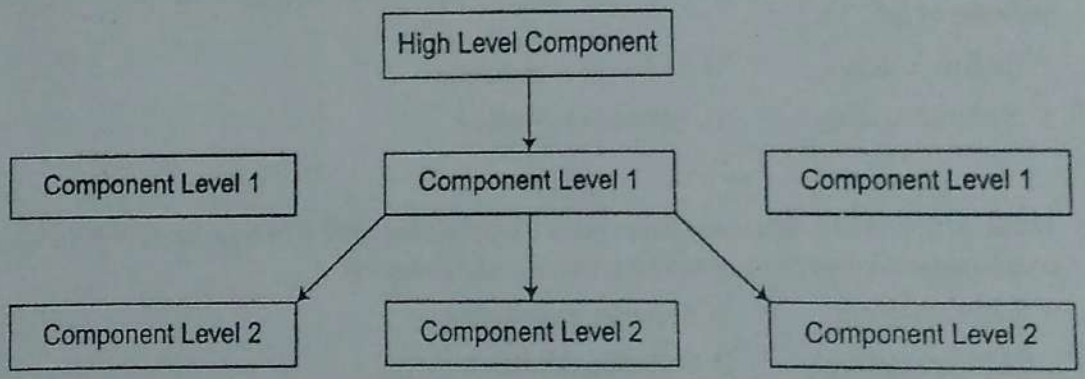


Figure 4.24 Top-Down Approach

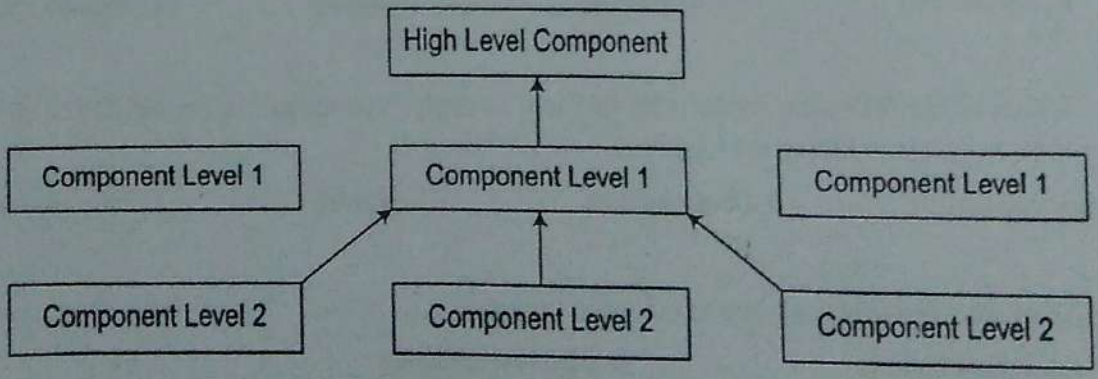


Figure 4.25 Bottom-Up Approach

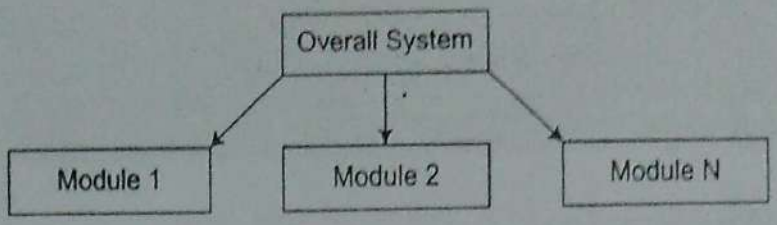


Figure 4.26 Modular Approach

4.8.3 User Interface Design

The data flow diagram (requirements analysis model) helps both architectural design and user interface design.

The user interface design creates an effective communication medium between a human (user of the system) and a computer. It proposes a set of interface design principles such that it identifies the objects and methods involved and then creates a screen layout that forms the basis for a user interface.

The user interface design focuses on three basic areas:

1. Interface between components of the system
2. Interface between the system and humans (users)
3. Interface between the system and other systems (nonhumans)

According to Theo Mandel, the three "golden rules" of interface design are:

1. Place the user in control
2. Reduce the user's memory load
3. Make the interface consistent

These three golden rules are discussed in detail in Chapter 7.

4.8.4 Procedural Design

Procedural design is also called component design. It is completely based on process specification (PSPEC) and control specification (CSPEC). The "state transition diagram" of the requirements analysis model is also used in component design.

Component design is usually done after user interface design. During user interface design, we discuss the components and details of their interface. At the component design stage, these interfaces get fine tuned. Component-level architecture and data designs are fine tuned and lower level algorithms are defined in detail for each component.

Component-level design depicts the software at a level of abstraction that is very close to the code. At the component level, the software engineer must represent data structures, interfaces, and algorithms in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of a number of design notations that represent component-level detail (text, graphical, or tabular formats).

Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, code, test, and maintain.

4.8.5 Deployment-Level Design

Deployment-level design takes care of the procedure of implementing the actual systems and their components within a physical environment. The Unified Modelling Language (UML) deployment diagram is used for this purpose and we will be discussing UML in detail in Chapter 6.

4.9 DESIGN DOCUMENTATION

There are two types of design documents, namely, high-level design document and low-level design document. A high-level design document contains architectural details.

System architecture document (SAD) is the design document where all the design and architectural elements are captured. An SAD along with a low-level design document is used as an input to the coders (developers). A low-level design document contains the data structure and its associated components along with interaction details.

Summary

A software design is an appropriate engineering representation of a software product under study and is the process of problem solving by converting the requirements of the product into a software solution. Software Engineering comprises strict disciplines that need to be followed to develop a programmable solution to solve the problems of customers. Strategies are defined to obtain a runable, efficient, and maintainable software solution that takes care of the costs, quality, resources, and other expenditure.

- The design of function-oriented system is called function-oriented design and the design of object-oriented system is called object-oriented design
- The possibility of dividing a single project into smaller units called modules is termed **modularity**.
- **Cohesion** refers to "how strong" one module is related to the other, which helps to group similar items together. Cohesion measures the semantic strength (high and low cohesion) of relationships between modules within a functional unit.

  The various types of cohesion are:
  - Coincidental cohesion
  - Logical cohesion
  - Temporal cohesion
  - Procedural cohesion
  - Communicational cohesion
  - Sequential cohesion
  - Functional cohesion
- **Coupling** is a measure of "how tightly" two entities or modules are related to each other. Coupling measures the strength of the relationships between entities or modules.

  The different types of coupling are:
  - Content coupling
  - Common coupling
  - Control coupling
  - Stamp coupling
  - Data coupling
  - Uncoupled

- The **scope of effect** of a module refers to the modules that are affected by a decision made by a particular module.
- The **scope of control** of a module refers to that module and all of its subordinate modules belonging to that particular module.
- A distributed system has three main components, namely, data, process, and interface. If these components are distributed across locations, it is called a distributed system. A computer network connects the components which are at different locations. In a centralized structure, the server (data and process components) resides in a centralized place. The interface component (called client) is distributed across locations. In a decentralized structure, both the client and the server reside in the same machine and they also communicate with another machine to share the resources (server and client components), which is also called peer-to-peer communication.
- **Procedural/component design** is completely based on process specification (PSPEC) and control specification (CSPEC). The "state transition diagram" of the requirements analysis model is also used in component design.
- **Deployment-level design** takes care of the procedure of implementing the actual systems and their components within a physical environment. The UML deployment diagram is used for this purpose.

Model Questions

PART A (Objective type)

1. What is an appropriate engineering representation of a software product under study and is the process of problem solving by converting the requirements of the product into a software solution referred to as?

   A. Software design B. Software engineering

   C. Software coding D. Software validation

Answer: A

2. What is the study that comprises strict disciplines that need to be followed to develop a programmable solution to solve the problems of customers?

   A. Software design B. Software engineering

   C. Software coding D. Software validation

Answer: B

3. What is the possiioility of dividing a single project into smaller units called modules termed as?

   A. Modularity B. Cohesion C. Coupling D. Encapsulation

Answer: A

4. Which of the following terms refer to "how strongly" one module is related to the other, which helps to group similar items together?

   A. Modularity B. Cohesion C. Coupling D. Encapsulation

Answer: B

5. All of the following are types of cohesion except:

   A. Logical cohesion B. Physical cohesion

   C. Functional cohesion D. Communicational cohesion

Answer: B