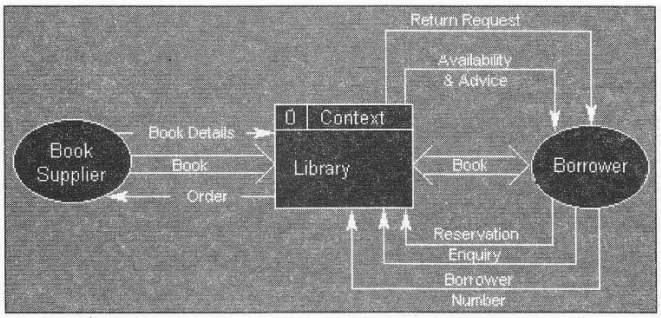


The components of a context diagram are clearly shown on this screen. The system under investigation is represented as a single process, connected to external entities by data flows and resource flows.

The context diagram clearly shows the interfaces between the system under investigation and the external entities with which it communicates. Therefore, whilst it is often conceptually trivial, a context diagram serves to focus attention on the system boundary and can help in clarifying the precise scope of the analysis.

The context diagram shown on this screen represents a book lending library. The library receives details of books, and orders books from one or more book suppliers.

Books may be reserved and borrowed by members of the public, who are required to give a borrower number. The library will notify borrowers when a reserved book becomes available or when a borrowed book becomes overdue.

In addition to supplying books, a book supplier will furnish details of specific books in response to library enquiries.

Note, that communications involving external entities are only included where they involve the 'system' process. Whilst a book supplier would communicate with various agencies, for example, publishers and other suppliers - these data flow are remote from the 'system' process and so this is not included on the context diagram.

Data Flow Diagrams – Context Diagram Guidelines

Firstly, draw and name a single process box that represents the entire system.

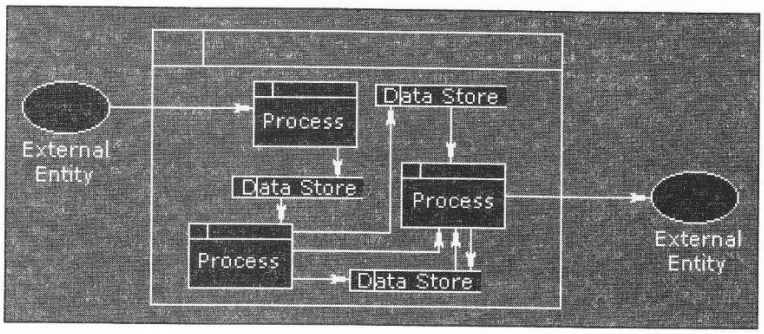
Next, identify and add the external entities that communicate directly with the process box. Do this by considering origin and destination of the resource flows and data flows.

Finally, add the resource flows and data flows to the diagram.

In drawing the context diagram you should only be concerned with the most important information flows. These will be concerned with issues such as: how orders are received and checked, with providing good customer service and with the paying of invoices. Remember that no business process diagram is the definitive solution - there is no absolute right or wrong.

Data Flow Diagrams – Level 1 Diagrams

The level 1 diagram shows the main functional areas of the system under investigation. As with the context diagram, any system under investigation should be represented by only one level 1 diagram.



There is no formula that can be applied in deciding what is, and what is not, a level 1 process. Level 1 processes should describe only the main functional areas of the system, and you should avoid the temptation of including lower level processes on this diagram. As a general rule no business process diagram should contain more than 12 process boxes.

The level 1 diagram is surrounded by the outline of a process box that represents the boundaries of the system. Because the level 1 diagram depicts the whole of the system under investigation, it can be difficult to know where to start.

There are three different methods, which provide a practical way to start the analysis. These are explained in the following section and any one of them, or a combination, may prove to be the most helpful in any given investigation.

There are three different methods, which provide a practical way to start the analysis. These are introduced below and any one of them, or a combination, may prove to be the most helpful in any given investigation:

Data Flow Diagrams - Resource Flow Analysis

Resource flow analysis may be a useful method for starting the analysis if the current system consists largely of the flow of goods, as this approach concentrates on following the flow of physical objects.

Resource flow analysis may be a useful method for developing diagrams if the current system consists largely of the flow of goods. Physical resources are traced from when they arrive within the boundaries of the system, through the points at which some action occurs, to their exit from the system. The rationale behind this method is that information will normally flow around the same paths as the physical objects.

Data Flow Diagrams - Organizational Structure Analysis

The organizational structure approach starts from an analysis of the main roles that exist within the organization, rather than the goods or information that is flowing around the system.

Identification of the key processes results from looking at the organizational structure and deciding which functional areas are relevant to the current investigation. By looking at these areas in more detail, and analyzing what staff actually do, discrete processes can be identified.

Starting with these processes, the information flows between them and between these processes and external entities are then identified and added to the diagram.

Data Flow Diagrams - Document Flow Analysis

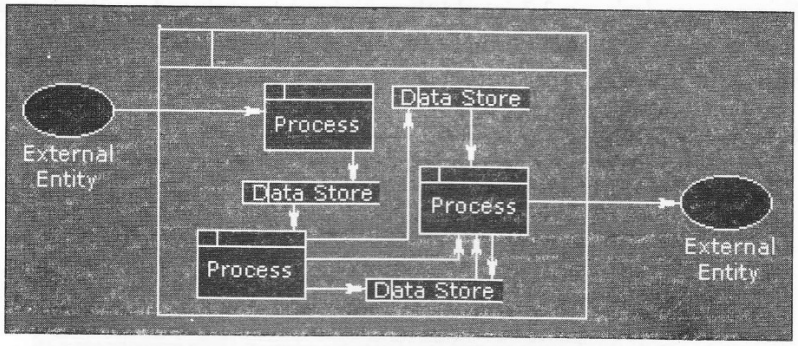
The document flow analysis approach is appropriate if the part of the business under investigation consists principally of flows of information in the form of documents or computer input and output.

Document flow analysis is particularly useful where information flows are of special interest. The first step is to list the major documents and their sources and recipients. This is followed by the identification of other major information flows such as telephone and computer transactions. Once the document flow diagram has been drawn the system boundary should be added.

Data Flow Diagrams - Top Down Expansion

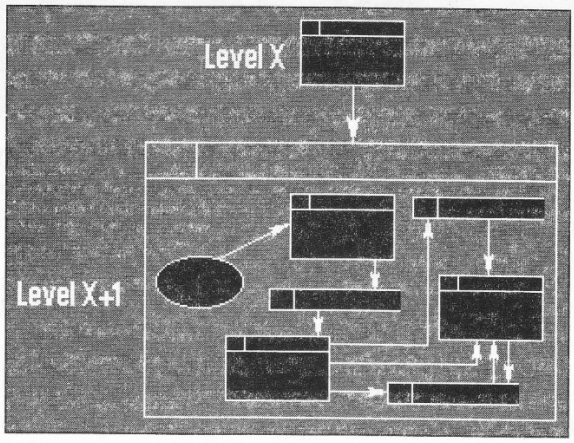
The section explains the process of top down expansion, or leveling. Furthermore, it illustrates that whilst there can only be one context and one level 1 diagram for a given system, these normally give rise to numerous lower level diagrams.

Each process within a given business process diagram may be the subject of further analysis. This involves identifying the lower level processes that together constitute the process as it was originally identified. This procedure is known as top-down expansion or leveling.



As a business process diagram is decomposed, each process box becomes a boundary for the next, lower level, diagram.

Data Flow Diagrams - Top Down Expansion Illustrated



In order to illustrate the process of top-down expansion, consider the three processes shown within this business process diagram. No detail is shown, only the outline of the process boxes, which have been identified during the drawing of a level 1 diagram.

Any area of a level 1 diagram is likely to require further analysis, as the level 1 diagram itself only provides a functional overview of the business system.

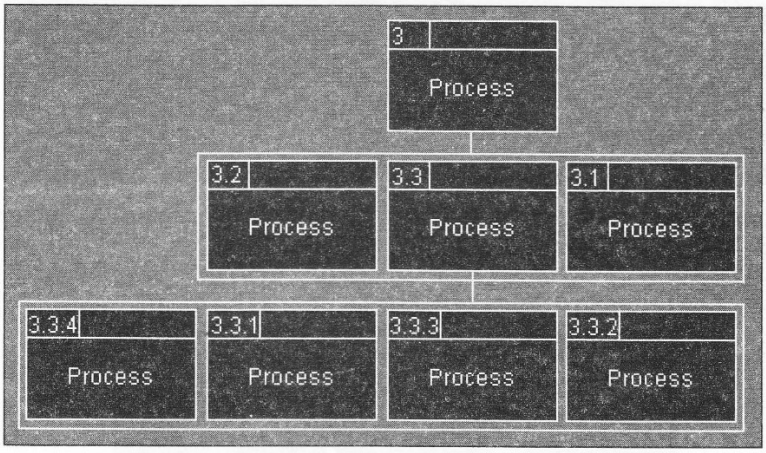
Therefore, below the level 1 diagram there will be a series of lower level diagrams. These are referred to as level 2, level 3, etcetera. In practice, level 2 is usually sufficient and it is unusual to carry out an analysis beyond level 3.

In this example the process numbered 3, at level 1, will be investigated further thereby giving rise to a level 2 diagram.

In the level 2 diagram four processes of interest have been identified and the numbering of these processes must reflect the parent process. Therefore the level 2 processes are numbered 3.1, 3.2, 3.3 and 3.4.

Suppose that of these four level 2 processes, one was of sufficient interest and complexity to justify further analysis. This process, let's say 3.3, could then be further analyzed resulting in a corresponding level 3 diagram. Once again the numbering of these processes must reflect the parent process. Therefore these three level 3 processes are numbered 3.3.1, 3.3.2 and 3.3.3.

Data Flow Diagrams – Numbering Rules



The process boxes on the level 1 diagram should be numbered arbitrarily, so that no priority is implied. Even where data from one process flows directly into another process, this does not necessarily mean that the first one has to finish before the second one can begin.

Therefore the processes on a level 1 diagram could be re-numbered without affecting the meaning of the diagram. This is true within any business process diagram - as these diagrams do not imply time, sequence or repetition.

However, as the analysis continues beyond level 1 it is important that a strict numbering convention is followed. The processes on level 2 diagrams must indicate their parent process within the level 1 diagram. This convention should continue through level 3 diagrams, and beyond, should that level of analysis ever be required.

The diagram on this screen clearly illustrates how processes on lower level diagrams identify their ancestral path.

Data Flow Diagrams - When to Stop

It is important to know when to stop the process of top-down expansion. Usually this will be at level 2 or level 3.

There are 3 useful guidelines to help you to decide when to stop the analysis:

Firstly, if a process has a single input data flow or a single output data flow then it should be apparent that there is little point in analyzing it any further.

Secondly, when a process can be accurately described by a single active verb with a singular object, this also indicates that the analysis has been carried out to a sufficiently low level. For example, the process named validate enquiry contains a single discrete task.

Finally, ask yourself if anything useful will be gained by further analysis of a process. Would any more detail influence your decisions?

If the answer is no, then there is little point in taking the analysis further.

Data Flow Diagrams – Keeping the Diagrams Clear

In this section a variety of simple techniques are introduced to show how a business process diagram can be clarified. The examples used do not relate to any specific scenario but are hypothetical abstracts used for the purpose of illustration.

Entity Relationship Diagram (ER Diagram)

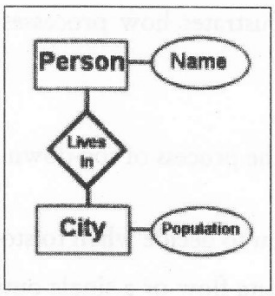
An entity-relationship (ER) diagram is a specialized graphic that illustrates the interrelationships between entities in a database. ER diagrams often use symbols to represent three different types of information. Boxes are commonly used to represent entities. Diamonds are normally used to represent relationships and ovals are used to represent attributes.

Data models are tools used in analysis to describe the data requirements and assumptions in the system from a top-down perspective. They also set the stage for the design of databases later on in the SDLC.

There are three basic elements in ER models:

- Entities are the "things" about which we seek information.
- Attributes are the data we collect about the entities.
- Relationships provide the structure needed to draw information from multiple entities.

Example: Consider the example of a database that contains information on the residents of a city. The ER diagram shown in the image above contains two entities – people and cities. There is a single "Lives In" relationship. In our example, due to space constraints, there is only one attribute associated with each entity. People have names and cities have populations. In a real-world example, each one of these would likely have many different attributes.



Developing an ERD

Developing an ERD requires an understanding of the system and its components. Before discussing the procedure, let's look at a narrative created by Professor Harman.

Consider a hospital:

- Patients are treated in a single ward by the doctors assigned to them. Usually each patient will be assigned a single doctor, but in rare cases they will have two.
- Heathcare assistants also attend to the patients, a number of these are associated with each ward.
- Initially the system will be concerned solely with drug treatment. Each patient is required to take a variety of drugs a certain number of times per day and for varying lengths of time.
- The system must record details concerning patient treatment and staff payment. Some staff are paid part time and doctors and care assistants work varying amounts of overtime at varying rates (subject to grade).
- The system will also need to track what treatments are required for which patients and when and it should be capable of calculating the cost of treatment per week for each patient (though it is currently unclear to what use this information will be put).

How do we start an ERD?

1. *Define Entities:* These are usually nouns used in descriptions of the system, in the discussion of business rules, or in documentation; identified in the narrative (see highlighted items above).
2. *Define Relationships:* These are usually verbs used in descriptions of the system or in discussion of the business rules (entity _____ entity); identified in the narrative (see highlighted items above).
3. *Add attributes to the relations:* These are determined by the queries, and may also suggest new entities, e.g. grade; or they may suggest the need for keys or identifiers.
   (a) What questions can we ask?
   (b) Which doctors work in which wards?
   (c) How much will be spent in a ward in a given week?
   (d) How much will a patient cost to treat?
   (e) How much does a doctor cost per week?
   (f) Which assistants can a patient expect to see?
   (g) Which drugs are being used?

4. Add cardinality to the relations

   Many-to-Many must be resolved to two one-to-manys with an additional entity.

   Usually automatically happens.

   Sometimes involves introduction of a link entity (which will be all foreign key) Examples: Patient-Drug.

5. This flexibility allows us to consider a variety of questions such as:

   (a) Which beds are free?

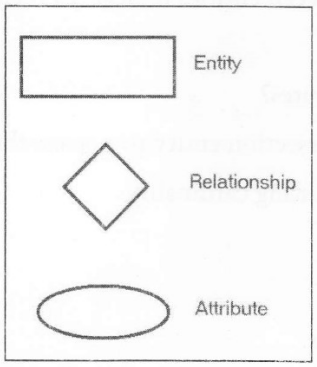
   (b) Which assistants work for Dr. X?

   (c) What is the least expensive prescription?

   (d) How many doctors are there in the hospital?

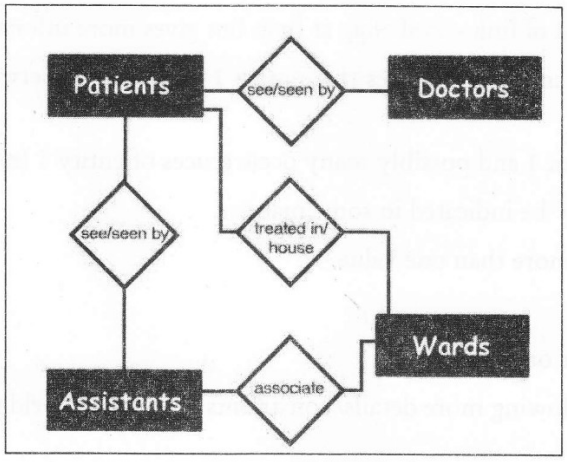
   (e) Which patient is family related?

6. Represent that information with symbols. Generally E-R Diagrams require the use of the following symbols:



Reading an ERD

It takes some practice reading an ERD, but they can be used with clients to discuss business rules. These allow us to represent the information from above such as the E-R Diagram below:



ERD brings out issues:

Many-to-Manys

Ambiguities

Entities and their relationships

What data needs to be stored

The Degree of a relationship

Exercise

Now, think about a university in terms of an ERD. What entities, relationships and attributes might you consider?

Other Styles of E-R Diagram

The text uses one particular style of diagram. Many variations exist.

Some of the variations are:

Diamonds being omitted - a link between entities indicates a relationship.

Less symbols, clearer picture.

What happens with descriptive attributes?

In this case, we have to create an intersection entity to possess the attributes.

Numbers instead of arrowheads indicating cardinality.

Symbols, 1, n and m used.

E.g. 1 to 1, 1 to n, n to m.

Easier to understand than arrowheads.

A range of numbers indicating optionality of relationship.

E.g (0,1) indicates minimum zero (optional), maximum 1.

Can also use (0,n), (1,1) or (1,n).

Typically used on near end of link - confusing at first, but gives more information.

E.g. entity 1 (0,1) -- (1,n) entity 2 indicates that entity 1 is related to between 0 and 1 occurrences of entity 2 (optional).

Entity 2 is related to at least 1 and possibly many occurrences of entity 1 (mandatory).

Multivalued attributes may be indicated in some manner.

Means attribute can have more than one value.

E.g. hobbies.

Has to be normalized later on.

Extended E-R diagrams allowing more details/constraints in the real world to be recorded.

Composite attributes.

Derived attributes.

Subclasses and superclasses.

Generalization and specialization.

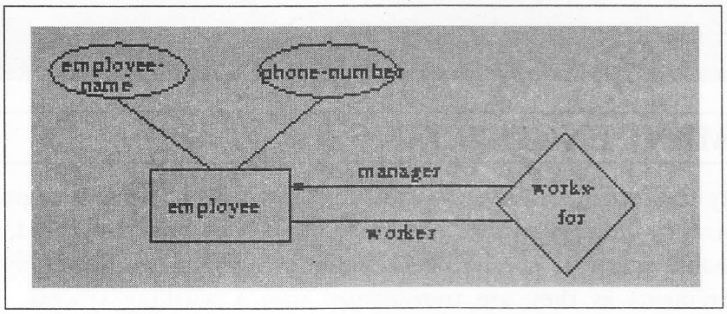
Roles in E-R Diagrams

The function that an entity plays in a relationship is called its role. Roles are normally explicit and not specified.

They are useful when the meaning of a relationship set needs clarification.

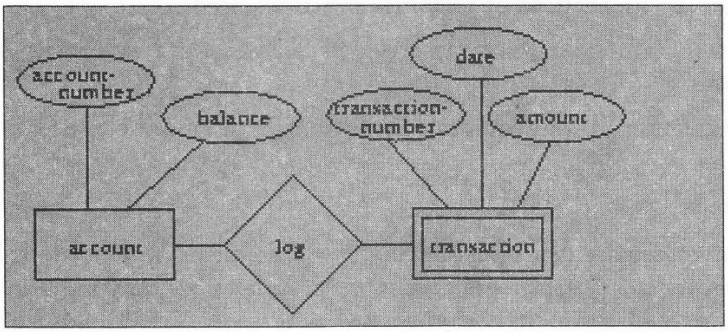
For example, the entity sets of a relationship may not be distinct. The relationship works-for might be ordered pairs of employees (first is manager, second is worker).

In the E-R diagram, this can be shown by labeling the lines connecting entities (rectangles) to relationships (diamonds)



Weak Entity Sets in E-R Diagrams

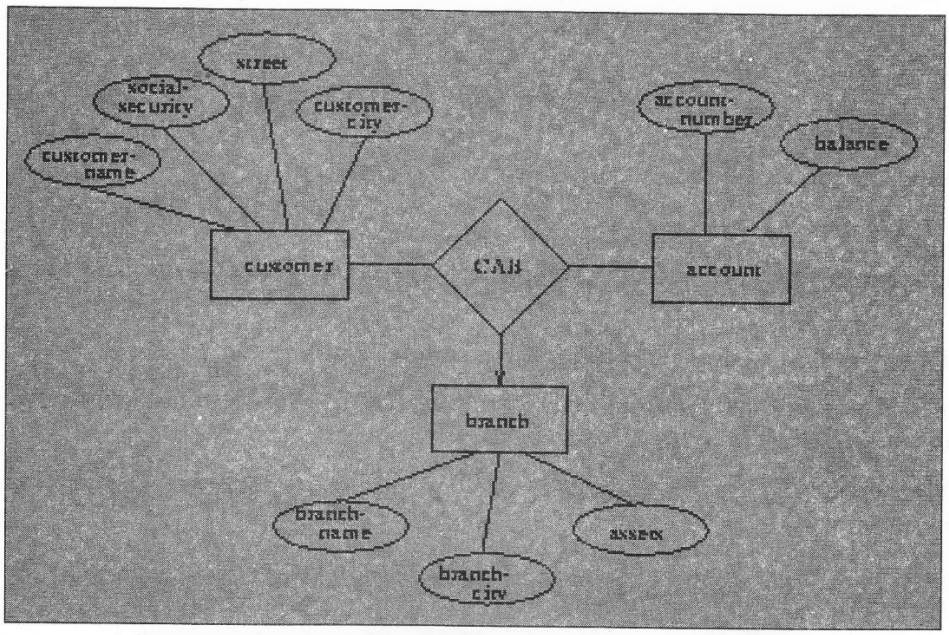
A weak entity set is indicated by a doubly-outlined box. For example, the previously-mentioned weak entity set transaction is dependent on the strong entity set account via the relationship set log.



Non-binary Relationships

Non-binary relationships can easily be represented.

This E-R diagram says that a customer may have several accounts, each located in a specific bank branch, and that an account may belong to several different customers.



4.2 REQUIREMENT ENGINEERING

Requirements specify the "what" of a system, not the "how". Requirements engineering provides the appropriate mechanism to understand what the customer desires, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution clearly, validating the specifications and managing the requirements as they are transformed into a working system. Thus, requirements engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behavior and its associated constraints.

The requirements engineering includes the following activities:

- Identification and documentation of customer and user's needs.
- Creation of a document describing the external behavior and associated constraints that will satisfy those needs.
- Analysis and validation of the requirements document to ensure consistency, completeness and feasibility.
- Evolution of needs.

The primary output of requirements engineering is requirements specification. If it describes both hardware and software it is a system requirement and if it describes only software requirement it is a software requirements specification. In both these cases, the system is treated as a black box.

Requirements Specification

Requirements specification can be done in the form of a document, graphical model, prototype or a combination of all these. System specification can be elicited using a standard template or a flexible approach depending upon the system being developed.

System specification is produced as a result of system and requirements engineering. It contains details about the hardware, software, database etc. It describes both the function and constraints required for developing a system. It also tells about the input and output information from the system.

Requirements Validation

The work product produced as a result of requirements engineering is checked for quality during the validation step. Requirements validation ensures that all the requirements have been stated correctly, inconsistencies and errors have been detected and corrected and that the work product conforms to the standard for process, project and product. This validation is done as a part of formal technical review as discussed in lesson 3. Although the requirements validation can be done in any way that leads to discovery of errors, the requirements can be examined against a checklist. Some of the checklist questions can be:

- Are the requirements clear or they can be misinterpreted?
- Is the source of the requirements identified?
- Has the final statement of requirement been verified against the source?
- Is the requirement quantitatively bounded?
- Is the requirement testable?
- Does the requirement violate domain constraints?

These questions and their likes ensure that validation team does everything possible for a thorough review.

Requirements Management

Requirements may change throughout the life of a computer based system. Requirements management includes the activities that help a project team to identify, control and track requirements and changes to these requirements at any time during the project.

Requirements management begins with identification. Once the requirements have been identified traceability tables are created. This table relates each requirement to one or more aspects of system or its environment. They may be at times used as a requirements database in order to understand how a change in one requirement will affect other aspects of the system being built.

Requirement Analysis

Requirements analysis is a software engineering task that bridges the gap between system level engineering and software design as shown in Figure 4.4.

Requirements analysis allow software engineer to refine the software allocation and build models of data, function and behavioral domain that will be used by the software. It provides the information to the designer to be transformed into data, architectural, interface and component-level design. Finally, it also provides the developer and the customer with the means to assess the quality of the software once it is built.

Software requirements analysis can be divided into five areas of effort:

- Problem recognition
- Evaluation and synthesis

- Modeling
- Specification
- Review

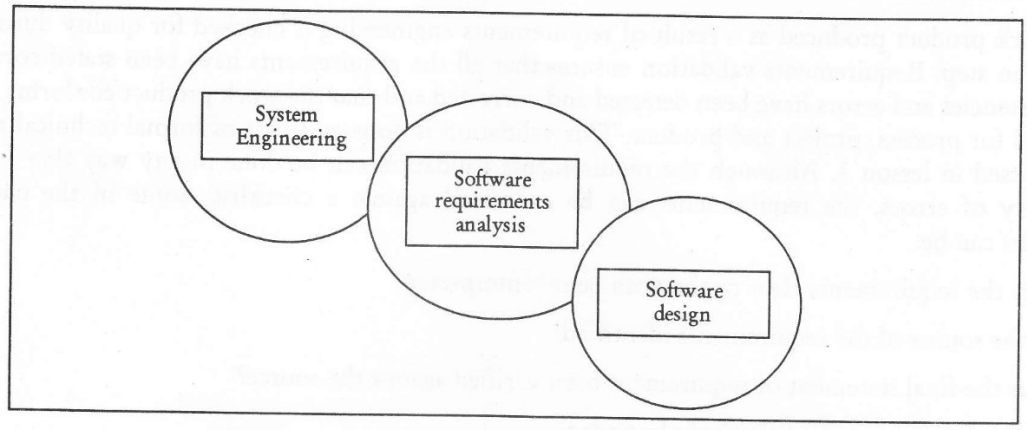


Figure 4.4: Analysis as a Bridge between System Engineering and Software Design

Initially, the analyst studies the system specification and the software project plan to understand the system as a whole in relation to the software and to review the scope of the software as per the planning estimates. After this the communication for analysis is established to ensure problem recognition.

Problem analysis is the next major area of effort for analysis which we will discuss in the next section.

4.3 PROBLEM ANALYSIS

It is the activity that includes learning about the problem to be solved, understanding the needs of the customer and users and understanding all the constraints on the solution. Requirements stage ends with creating a document called the Software Requirements Specification (SRS), which contains the complete details about the external behavior of the software system.

A formal approach to problem analysis is to simplify the job at hand i.e. organization of information, understanding the perspectives of all the people involved, finding and resolving conflicts and avoiding the internal design of the software.

Different people might share different views during problem analysis. It is important to track these views and relate them. A conflict in these perspectives will lead to a good problem analysis technique to resolve them.

There are different analysis techniques for different problems; also different aspects of the same problem need to be tacked differently. Most of the techniques used for problem analysis use a combination of notations, including data flow diagrams, data dictionaries, entity relationship diagrams and Coad object diagrams.

Out of all these techniques all except the Coad object diagrams have been discussed in detail in the previous unit. We will discus the Coad object diagrams in detail here.

4.3.1 Coad Object Diagram

Peter Coad developed a notation that combined the data flow diagrams and the entity relationship diagram with some additional features. Here, in this context, the objects are represented as rounded corner rectangles. Attributes of an object are stated with the object itself. Two types of structural relationship may exist between objects: classification and assembly. Classification depicts abstraction as shown in Figure 4.5.

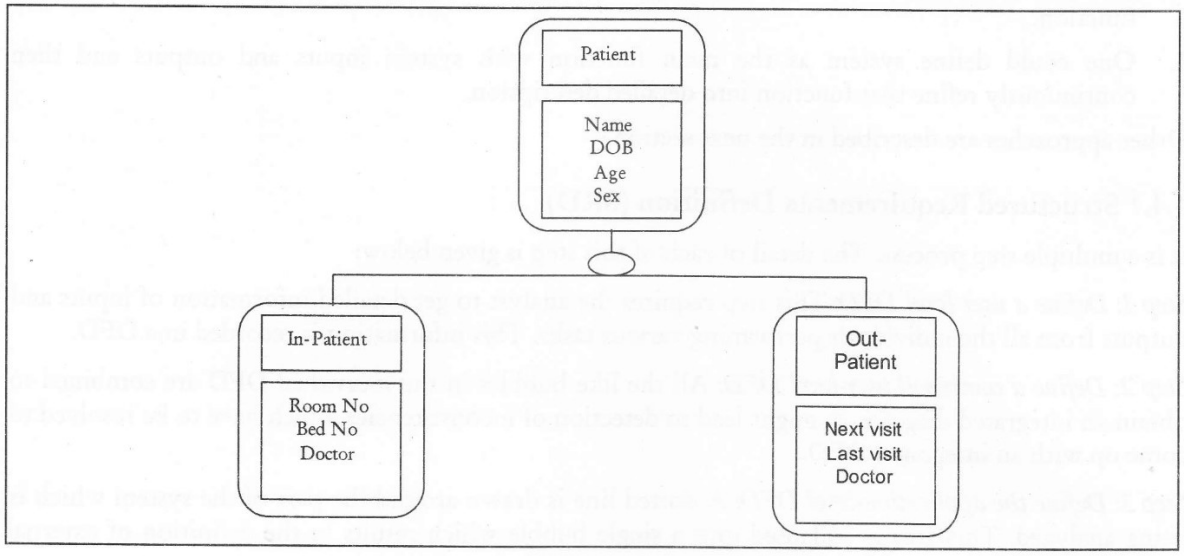


Figure 4.5: Example of a Coad Classification Structure

Using abstraction, we define some problems in abstract terms while the others in detail. Assembly relationships show that the subordinate objects are part of the super ordinate object i.e. patient as shown in Figure 4.6. All lines in the assembly structure have either an O for optional or a bar for mandatory.

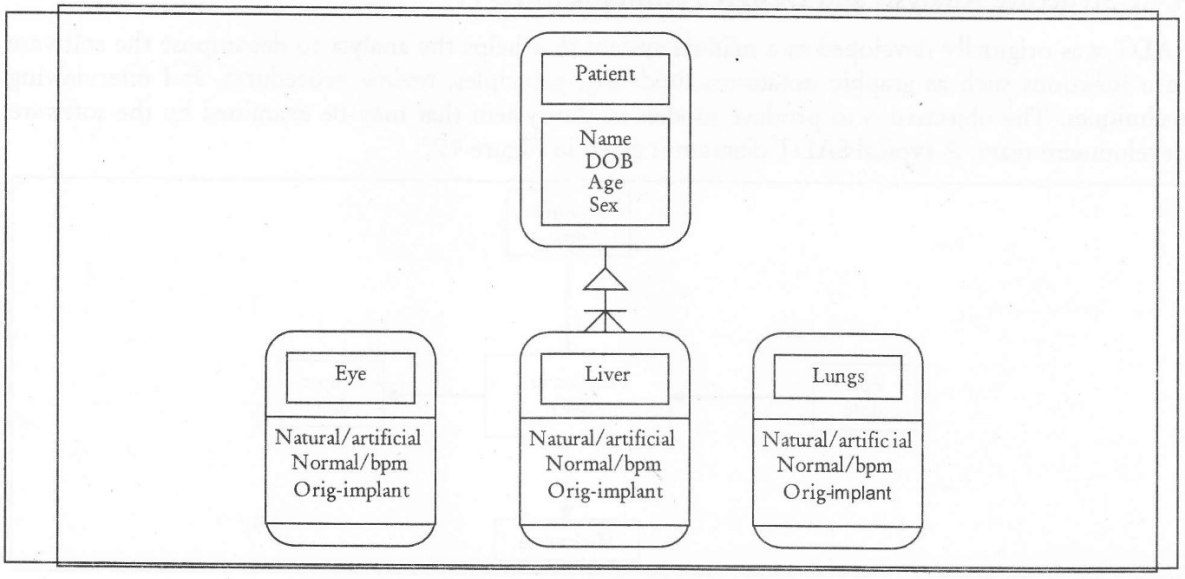


Figure 4.6: Example of a Coad Assembly Structure

4.4 APPROACHES TO PROBLEM ANALYSIS

All the aspects of problem and probably defined system boundary must be understood by the time problem analysis is complete. There are a number of ways to complete this task.

1. One can list all the inputs of the system, followed by all the outputs and then all the interfaces.
2. One can list all the interfaces first and then list all the inputs and outputs associated with each function.
3. One could define system as the main function with system inputs and outputs and then continuously refine that function into detailed description.

Other approaches are described in the next section.

4.4.1 Structured Requirements Definition (SRD)

It is a multiple step process. The detail of each of this step is given below:

Step 1: Define a user level DFD: This step requires the analyst to get detailed information of inputs and outputs from all the individuals performing various tasks. This information is recorded in a DFD.

Step 2: Define a combined user-level DFD: All the like bubbles in the individual DFD are combined to obtain an integrated diagram. It might lead to detection of inconsistencies which have to be resolved to come up with an integrated DFD.

Step 3: Define the application-level DFD: A dotted line is drawn around the part of the system which is being analyzed. This area is collapsed into a single bubble which results in the definition of external inputs and outputs.

Step 4: Define the application-level functions: For each function X performed by the system, all the inputs and outputs are numbered as X1, X2, X3, ... to depict the order of events in the function.

4.4.2 Structure Analysis and Design Technique (SADT)

SADT was originally developed as a manual system that helps the analyst to decompose the software into functions such as graphic notations, modeling, principles, review procedures, and interviewing techniques. The objective is to produce models of the system that may be examined by the software development team. A typical SADT diagram is given in Figure 4.7.

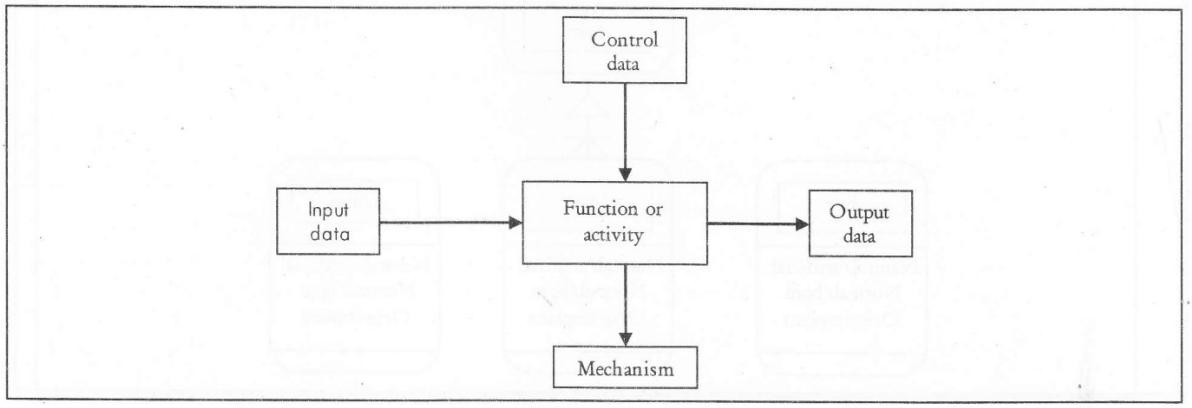


Figure 4.7: SADT Diagram (Context Diagram)

Using SADT a problem model is built which consists of a hierarchy of diagrams. Each diagram is composed of boxes and arrows. The uppermost diagram, called the context diagram defines the problem in an abstract manner. The problem, is refined into sub-problems, and is documented in another diagram. Each box is now detailed into a separate SADT diagram.

Boxes are arranged along a diagonal using arrow and should be given unique names for functions. Boxes should be numbered in the right corner to show their relative dominance. Although this is not a design technique but it helps to analyze a problem in detail. Each box represents a component of the problem. Once the analysis is complete prior to writing the SRS it is better to decide so as to what will be automated. Also, optimal decomposition leads to a better understanding of the design.

4.4.3 Software Prototyping

Prototyping is the techniques of constructing a partial implementation of a system so that the users, customers and developers can have a better knowledge of the system. It is a partial implementation and that is the reason it is called a prototype.

Prototyping has evolved around two technologies: evolutionary and throw-away. In the throw-away approach, the prototype is discarded after extracting the desired knowledge about its problem or solution. In the evolutionary approach, the prototype is constructed to learn about its problem or solution in successive steps. Once the prototype has been used and the desired knowledge attained, the prototype is modified as per the new better-understood needs. Thus, the benefits of an early prototype are as under:

- Identifications of misunderstandings between the customer and the developers as the functions are specified.
- Missing user requirements can be detected.
- Difficult-to-use or confusing requirements may be identified and improved.
- A working system is available early to exhibit the capability and usefulness of the application to the management.
- It serves as the basis for writing specification of the system.

The differences between the throw-away and evolutionary prototyping have been illustrated in the Table 4.1.

Table 4.1: Differences between Throw-away and Evolutionary Prototyping

Sr. No.	Approach and Characteristics	Throw-away	Evolutionary
1	Development approach	Quick and dirty. No rigor.	No sloppiness. Rigorous
2	What to build	Build only difficult parts	Build understood parts first and build on solid foundations
3	Design drivers	Optimize development time	Optimize modifiability
4	Ultimate objective	Throw it away	Evolve it

Prototyping Pitfalls

A common problem with using prototyping technology is high expectations for productivity with insufficient effort. It can have execution inefficiencies with the associated tools and this may be considered as a negative aspect of prototyping.

The process of providing an early feedback to the user results in a problem related to the behavior of end-user and developers. An end-user with a poor past development experience can be biased with the developments in future.

Prototyping Opportunities

Software prototyping cannot be got rid of totally. The benefits of prototyping are obvious and established. The end user cannot expect the developer to come up with a full fledged system with incomplete software needs.

Prototyping must be used on critical portions of the existing software. The idea of completely re-engineering software is not a good idea. Total re-engineering must be planned and should not be used in reaction to a crisis situation.

Software prototyping must be implemented in an organization with the help of trainings, case-studies and library development. The end user involvement can be enhanced when the requirements are prototyped and communicated before starting with the development. Moreover, during the maintenance stage, the requirement change should lead to prototype being enhanced before the actual changes are confirmed.

4.5 SOFTWARE REQUIREMENT SPECIFICATION (SRS)

The SRS is contains specifications for a particular software product, program or a set of programs that perform a particular function in a specific environment. It serves a number of purposes depending upon its creator.

(a) SRS could be written by the customer. It must address the needs and expectations of the users.

(b) SRS could be written by the developer of the system. It serves a different purpose and mentions the contrast between the customer and the developer.

4.5.1 Nature of SRS

The SRS must address the following issues:

Functionality: What the software does?

External Interfaces: How does the software interact with other system's hardware, software, people, etc?

Performance: What is the speed, response time, recovery time, etc of the various software functions?

Attributes: What are the considerations for portability, correctness, maintainability, security, reliability, etc?

Design constraints imposed on implementation: Are there any standards on implementation language, database integrity, operating environments, etc.

4.5.2 Roles of SRS

SRS plays a specific role in the software development process. So, its creators must not go beyond a certain limit of this role. This means that a SRS:

- Should correctly define all the software requirements in order to involve a particular task in the system.
- Should not describe any design or implementation details but these should be contained in the design stage.
- Should not impose additional constraints on software. These should be defined in other documents like software quality assurance plan.

4.5.3 Characteristics of a Good SRS

An SRS should have the following characteristics:

- *Correct:* An SRS is said to be correct if it contains the requirements that the software should meet. The correctness cannot be measured through a tool or a procedure.
- *Unambiguous:* An SRS is unambiguous only if the requirements contained in it have only one interpretation. It must be unambiguous to both who create and who use it. However, the two groups may use different languages for description. But the language should be a natural language that can be easily understood by both the groups.
- *Complete:* An SRS is complete if it includes the following:
  - ❖ All significant requirements related to functionality, performance, design, attributes or external interfaces.
  - ❖ Definition of the software responses to all possible input classes in all possible situations.
  - ❖ Full references and labels to figures, tables, diagrams and the definition of all terms and units of measure.
- *Consistent:* An SRS is consistent if no subset of a requirement defines it in entirety. The conflicts can be of three types:
  - ❖ Characteristics of real-world objects.
  - ❖ Logical or temporal conflicts.
  - ❖ The same real world object defined in more than one places in different terminologies.
- *Ranked for importance and stability:* An SRS is ranked for importance and stability if each requirement contained in it has an identifier indicating its importance or stability of that requirement. Another way to implement the importance of requirements is to classify them as essential, conditional and optional.
- *Verifiable:* An SRS is verifiable if every requirement contained in it is verifiable i.e. there exists some defined method to judge if a software meets all the requirements. Non-verifiable requirements should be removed or revised.
- *Modifiable:* An SRS is modifiable if its styles and structures can be retained easily, completely and consistently while changing a requirement.

- *Traceable:* An SRS is traceable if each of its requirements is clear and can be referenced in future development or enhancement documentation. Two types of traceability are recommended:
  - *Backward traceability:* This depends upon each requirement explicitly referencing its source in earlier documents.
  - *Forward traceability:* This depends upon each requirement in SRS having a unique name or reference number.

4.5.4 Organization of the SRS

The Institute of Electrical and Electronics Engineers (IEEE) has published guidelines and standards to organize an SRS document. There is no single method that is suitable for all projects. So, different ways are proposed to structure the SRS. Although the first two sections are the same, the third section containing the "specific requirements" can differ.

The general organization of the SRS is given in Table 4.2.

Table 4.2: Organization of an SRS

1	Introduction
1.1	Purpose
1.2	Scope
1.3	Definitions, Acronyms and Abbreviations
1.4	References
1.5	Overview
2.	Overall Description
2.1	Product Perspective
2.2	Product Functions
2.3	User Characteristics
2.4	Constraints
2.5	Assumptions and Dependencies
3.	Specific Requirements

4.6 BEHAVIORAL AND NON-BEHAVIORAL REQUIREMENTS

Behavioral requirements define precisely what inputs are expected by the software, what outputs will be generated by the software, and the details of the relationships that exist between those inputs and outputs. In short behavioral requirements describe all aspects of interfaces between the software and its environment (that is hardware, humans, and other software). Therefore a well written SRS specifies both behavioral and non-behavioral requirements. These requirements can be specified using a lot of techniques like finite state machines, decision tables and decision trees and program design language.

Knowing how to specify behavioral requirements is only half the battle. All applications, from the most trivial to the most complex, have additional requirements that define the overall qualities or

attributes to be exhibited by the software system. In general an SRS need not specify every one of these; it is necessary to emphasize these factors of particular importance to the particular application. For example if the application is life critical, reliability becomes paramount. If the product is to be long lived, then portability and modifiability are usually critical. These are the non-behavioral requirements of software. They can be expressed in terms of the following:

Portability, Reliability and Efficiency: Portability is the degree to which the software running on one host computer can be converted to one which can run on another host computer environment. A short-lived application is impossible to be ported or lives a long life without getting upgraded. Portability cannot be quantified.

Reliability of software is defined to be the ability of the software to behave in a user-acceptable manner when subjected to an environment in which it was intended to be used.

Efficiency refers to the level at which the software uses scarce system resources like memory, disk space, buffers, communication channels, etc.

Testability, Understandability and Modifiability: Testability, Understandability and Modifiability are very closely related and it is difficult to quantify these in an SRS. These contribute highly to the full life cycle cost of the software.

Check Your Progress

1. An SRS is inconsistent if no subset of a requirement defines it in entirety. (True/False)
2. The end user involvement can be enhanced when the requirements are prototyped and communicated before starting with the development. (True/False)
3. Requirements stage ends with creating a document called the .....

4.7 LET US SUM UP

System engineering demands vast communication between the customer and the system engineer. This is achieved through a set of activities called system engineering–elicitation, analysis and negotiation, specification, modeling, validation and management.

After the requirements have been isolated, a system model is produced and a representation of each major subsystem can be developed. The system engineering task terminates with the creation of system specification - a document that forms the foundation for all engineering work that follows.

Requirements analysis allow software engineer to refine the software allocation and build models of data, function and behavioral domain that will be used by the software. Problem Analysis is the activity that includes learning about the problem to be solved, understanding the needs of the customer and users and understanding all the constraints on the solution. Requirements stage ends with creating a document called the Software Requirements Specification (SRS), which contains the complete details about the external behavior of the software system.

The system requirements can be classified into behavioral and non-behavioral requirements.

4.8 KEYWORDS

SRS: Software Requirements Specification

SRD: Software Requirements Design

IEEE: Institute of Electrical and Electronic Engineers

SADT: Software Analysis and Design Technique

4.9 QUESTIONS FOR DISCUSSION

1. What is the degree of a relationship? Explain using examples.
2. Explain the relationship between minimum cardinality and optional and mandatory participation.
3. Discuss the significance and use of requirements engineering. What are the problems in the formulation of the requirements?

Check Your Progress: Model Answers

1. False
2. True
3. Software Requirements Specification

4.10 SUGGESTED READINGS

R.S. Pressman, *Software Engineering-A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Belady L, *Foreword to Software Design: Methods and Techniques*, Yourdon Press, 1981.

R Fairly, *software Engineering Concepts*, Tata McGraw Hill Pub., 2000.

Sommerville, *Software Engineering*, Addison-Wesley, 3rd Edition, 1989.

Paul Feld, *An Introduction to Object Oriented Design*, Paulfield@dial.pipex.com, 2001.

UNIT III

LESSON

5

SOFTWARE DESIGN

CONTENTS

5.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Explain the basic concepts of software design, conceptual design and technical design
- Describe fundamentals of modularity and dependence matrix

5.1 INTRODUCTION

The purpose of software design is to produce a workable (implement able) solution to a given problem. Although methods, processes, and tools can help guide a designer, software design is essentially a creative process.

5.2 SOFTWARE DESIGN

5.2.1 Design Views

Architectural Design: A high-level modular structure for the software is created. Modules are identified, data and functionality are modeled, and module relationship are described.

Data Design: The information analysis models are translated into data structure design.

Interface Design: A description of how the software communicates with other systems and with human users is developed.

Temporal Design: The events and signals, state transitions, processes, and timing conditions and constraints are incorporated into the design model.

Detailed Design (procedural design): Algorithms are designed to satisfy the required functionality of each module.

5.2.2 Design Concepts

- The design should be based on requirements specification
- The design should be documented (so that it supports implementation, verification, and maintenance)
- The design should use abstraction (to reduce complexity and to hide unnecessary detail)
- The design should be modular (to support abstraction, verification, and maintenance)
- The design should be assessed for quality as it is being created, not after the fact
- Design should produce modules that exhibit independent functional characteristics
- Design should support verification and maintenance.

A number of design concepts exist which provide the software designer sophisticated design methods to be applied. These concepts answer a lot of questions like the following:

- Which criteria should be used to decompose the software into multiple components?
- How is the data or functional detail isolated from the conceptual representation of the software?
- What uniform criteria must be applied to define the design quality?

Abstraction

Abstraction is the elimination of the irrelevant and amplification of the essentials. A number of levels of abstraction exist in a modular design. At the highest level, the solution is mentioned at a very broad level in terms of the problem. At the lower levels, a procedure oriented action is taken wherein problem oriented design is coupled with implementation level design. At the lowest level, the solution is mentioned in the form that can be directly used for coding.

The various levels of abstraction are as follows:

- *Procedural Abstraction:* It is a sequence of instructions that have limited functions in a specific area. E.g. the word "prepare" for tea. Although it involves a lot of actions like going to the kitchen,

boiling water in the kettle, adding tea leaves, sugar and milk, removing the kettle from the gas stove and finally putting the gas off.

- *Data Abstraction:* It is a named collection of data that describes the data object. E.g. data abstraction for prepare tea will used kettle as a data object which in turn would contain a number of attributes like brand, weight, color, etc.
- *Control Abstraction:* It implies a program control mechanism without specifying its internal details. E.g. synchronization semaphore in operating system which is used for coordination amongst various activities.

Refinement

Stepwise refinement is a top-down strategy wherein a program is developed by continually refining the procedural details at each level. This hierarchy is obtained by decomposing a statement of function step-by-step until the programming language statements are obtained.

At each step one or more instructions are decomposed into more detailed instructions until all the instructions have been expressed in terms of programming language. Parallel to the instructions, data is also refined along with program and specifications. Each refinement involves design decisions.

Refinement is nothing but elaboration. A statement of function is defined at a high level of abstraction which mentions the functions conceptually without specifying the internal working or structure. With continuous refinement designer provides more detail at each step continuously.

Software Architecture

Software architecture refers to the overall structure of the software and the way in which this structure provides conceptual integrity for a system. Architecture is the hierarchical structure of the program components, the way these components interact and the structure of data that is being used in these components.

Software architecture design must possess the following properties:

1. *Structural properties:* This design specifies the components of a system and the manner in which these components are included and interact in the system.
2. *Extra-functional properties:* The design must include the requirements for performance, capacity, security, reliability, adaptability, etc.
3. *Families of related systems:* The design must include those components that can be re-used while designing similar kinds of systems.

An architectural design can be represented in one of the following ways:

1. *Structural model:* It represents the architecture as a collection of components.
2. *Framework model:* It increases the design abstraction level by identifying the similar kinds of design frameworks contained in similar applications.
3. *Dynamic model:* It answers the behavioral aspect of the architecture, specifying the structural or system configuration changes as a result of changes in the external environment.
4. *Process model:* It focuses on the designing of business or technical processes that must be included in the system.
5. *Functional model:* It represents the functional hierarchy of the system.