2. Software Requirements

Objectives:

- To introduce the concepts of user and system requirements.
- To describe functional and non-functional requirements.
- To explain two techniques for describing system requirements.
- To explain how software requirements may be organized in a requirements document.
- To explain about the Software prototyping.
- To define about the Functional and Behavioural Models.
- To introduce the concept of Structured Analysis.
- To know the concept about the Data Dictionary.

Topics covered:

- Functional and non-functional requirements
- User requirements
- System requirements
- The software requirements document.
- Feasibility studies
- Software Prototyping
- Prototyping in the Software Process
- Functional and Behavioral Models
- Structured Analysis
- Data Dictionary.
Chapter 1 Functional and non-functional requirements

Objectives
- To describe the principal requirements engineering activities
- To introduce techniques for requirements elicitation and analysis
- To describe requirements validation
- To discuss the role of requirements management in support of other requirements engineering processes

Introduction:

1. Functional requirements
   - Statements of services the system should provide how the system should react to particular inputs and how the system should behave in particular situations.

2. Non-functional requirements
   - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

Functional requirements
- Describe functionality or system services
- Depend on the type of software, expected users and the type of system where the software is used
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions \( \{f_i\} \).

The functional view of the system is shown in fig below. Each function \( f_i \) of the system can be considered as a transformation of a set of input data \( (ii)_i \) to the corresponding set of output data \( (o)_i \). The user can get some meaningful piece of work done using a high-level function.
Examples of functional requirements

- The user shall be able to search either all of the initial set of databases or select a subset from it.
- The system shall provide appropriate viewers for the user to read documents in the document store.
- Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account’s permanent storage area.

Requirements imprecision

- Problems arise when requirements are not precisely stated
- Ambiguous requirements may be interpreted in different ways by developers and users
- Consider the term “appropriate viewers”
  - User intention - special purpose viewer for each different document type
  - Developer interpretation - Provide a text viewer that shows the contents of the document

Requirements completeness and consistency

- In principle requirements should be both complete and consistent
  - They should include descriptions of all facilities required
  - There should be no conflicts or contradictions in the descriptions of the system facilities

In practice, it is impossible to produce a complete and consistent requirements document.
Nonfunctional requirements:-

- Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.
- Nonfunctional requirements may include: # reliability issues, # accuracy of results, # human-computer interface issues, # constraints on the system implementation, etc.
- Define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional classifications

- Product requirements
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organizational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.
Non-functional requirement types

Non-functional requirements examples

- **Product requirement**
  It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard character set.

- **Organizational requirement**
  The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95

- **External requirement**
  The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

**Key points:**

- Requirements set out what the system should do and define constraints on its operation and implementation.
- Functional requirements set out services the system should provide.
• Non-functional requirements constrain the system being developed or the development process.
• User requirements are high-level statements of what the system should do.
• User requirements should be written in natural language, tables and diagrams.
• System requirements are intended to communicate the functions that the system should provide.
• System requirements may be written in structured natural language, a PDL or in a formal language.
• A software requirements document is an agreed statement of the system requirements.

SUMMARY:
Requirements analysis is the first technical step in the software process. It is at this point that a general statement of software scope is refined into a concrete specification that becomes the foundation for all software engineering activities that follow. Analysis must focus on the information, functional, and behavioral domains of a problem.

Assessment Questions:
1. What are the difference between requirements definition and requirement specification?
2. Briefly describe the desirable characteristics and structure of a requirements document.
3. With an example, explain the use of viewpoint template and service template in the VORD method.
4. Discuss in detail the various steps of requirements engineering.
Chapter 2 Documentation

Objectives:

- To describe about the usage about the documentation
- To know about the different types of documentation
- To explain about the Phase, Stage and Step Documents

In this section we will briefly overview three kinds of documents that result from the engineering of the steps, stages and phases. It is important that the reader keeps the universe of discourse in mind, either the domain, the requirements, the software, the two first (domain and requirements), the two last (requirements and software) or all three (an entire development).

Document Kinds

There are basically three kinds of documents that emerge from the development process, and which the developer hence should be aiming at. These are:

(1) Informative documents, or document parts, such as partners and current situation, needs and ideas, product concepts and facilities, scope and span delineations, assumptions and dependencies, implicit/derivative goals, synopsis, design briefs, contracts, logbook;

(2) The description documents, or document parts, such as rough sketches (records of "brainstorming"), terminologies, narratives, and formal models; and finally

(3) The analytic documents, or document parts, such as description property verifications, verification of correctness of development transition (i.e., development step), and validation of formal and informal descriptions. We will briefly review these kinds of documents, both as concerns their pragmatics: why they are necessary, and as concerns their multitude: why there are so many seemingly different kinds of documents.

Phase, Stage and Step Documents

A development phase results in a comprehensive, definitive set of informative, descriptive and analytic documents. A development stage results, similarly, in a comprehensive set of informative, descriptive and analytic documents, or in a set of relatively complete domain, interface or machine requirements prescriptions. The
boundaries between a sub phase and a stage, and the comprehensiveness of either, are not sharp. There is nothing else remaining from steps, stages and phases than documents, on paper or electronically. It is important that the reader keeps the universe of discourse in mind, either the domain, the equipments, the software, the two first (domain and requirements), the two last (requirements and software) or all three (an entire development). It must first be clearly stated, lest one of the "parties" of a development contract gets confused from the very start!

**Informative Documents:**

**Characterisation.** By an informative document we mean a document, or a document part, which informs, it does not necessarily describe a design table, manifest phenomena or concept. As the name implies, informative documents give information which takes any forms. Informative documents include those of perceived or already announced needs, product concepts and facilities, scope and span delineations, assumptions and dependencies, implicit/derivative goals, synopsis, contracts, design briefs, and so on.

**Current Situation Documentation**

Need for software development, or for requirements prescription, or for domain description usually arise out of a current situation. A current situation may be that the domain is not well-understood, or that software is required.

**Needs Documentation**

Needs refer to perceived or actual needs for the product being desired, whether a domain description, a requirements prescription, a software design (i.e., specification), or just plainly, as is most often the case, the software itself.

**Descriptive Documents**

**Characterization.** By a descriptive document we mean a document, or a document part, which describes a manifest phenomenon or a concept. The term describe, and hence the terms description, and descriptive, are here used in a rather specific, narrow sense. A description designates (i.e., is some text that sets forth, in words) either some physically
existing part of nature (one that centre around physical behaviors usually governed by laws of physics) or some man-made part of the world (one that centres around human activities, including their interaction with artifacts) or some combination of these two classes of worlds. Hence opinions, emotions, metaphysical, political or such other similar subjective texts are not here considered descriptions. It can be seen from the above, and it will reappear, again and again later, that it is not a simple, straightforward matter to delineate precisely when something is a description (a prescription, a specification), and what can be described, that is, what can exist.

Analytic Documents

Characterization. By an analytic document we mean a document whose subject is a descriptive document. The text of an analytic document analyses a descriptive document.

• As the term indicates, analytic documents are documents whose content represents analyses of other documents, here the descriptive documents. We consider four kinds of analytic documents: those that represent
  1) Formation of concepts from rough sketches (during brainstorming),
  2) Validation of formal and informal description documents,
  3) Description property verifications, and
  4) Verification of the correctness of development transitions (i.e., development steps).

There may be other analytic documents. Examples: documents whose content analyses behavioral aspects of the intended computing system, such as expected interface response times based on queuing theoretic studies; expected machine computation times based on complexity theoretic studies; details of dictionary or database hashing algorithms based on statistical studies of reference patterns; and so on. Also included may be documents whose contents analyze pragmatic issues such as, production line flow (congestion), based on statistical studies, for a project and production planning, monitoring and control computing system; company cash flow, based on similar studies, for a financial services or an electronic trading computing system; and so on. Further kinds of analytic documents can be imagined. We shall, in these volumes, only cover those just mentioned.
REQUIREMENTS ENGINEERING

The outcome of the system engineering process is the specification of a computer based system or product at the different levels described generically. Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. The requirements engineering process can be described in five distinct steps:

- Requirements elicitation
- Requirements analysis and negotiation
- Requirements specification
- System modeling
- Requirements validation
- Requirements management

Requirements Elicitation

It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn’t simple—it’s very hard.

- Problems of scope.
- Problems of understanding.
- Problems of volatility.

The requirements change over time. To help overcome these problems, system engineers must approach the requirements gathering activity in an organized manner.

Somerville and Sawyer suggest a set of detailed guidelines for requirements elicitation, which are summarized in the following steps:

- Assess the business and technical feasibility for the proposed system.
- Identify the people who will help specify requirements and understand their organizational bias.
• Define the technical environment (e.g., computing architecture, operating system, telecommunications needs) into which the system or product will be placed.

• Identify “domain constraints” (i.e., characteristics of the business environment specific to the application domain) that limit the functionality or performance of the system or product to be built.

• Define one or more requirements elicitation methods (e.g., interviews, focus groups, team meetings).

• Solicit participation from many people so that requirements are defined from different points of view; be sure to identify the rationale for each requirement that is recorded.

• Identify ambiguous requirements as candidates for prototyping.

The work products produced as a consequence of the requirements elicitation activity will vary depending on the size of the system or product to be built. For most systems, the work products include

• A statement of need and feasibility.

• A bounded statement of scope for the system or product.

• A list of customers, users, and other stakeholders who participated in the requirements elicitation activity.

• A description of the system’s technical environment.

• A list of requirements (preferably organized by function) and the domain constraints those apply to each.

• A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.

• Any prototypes developed to better define requirements. Each of these work products is reviewed by all people who have participated in the requirements elicitation.

Requirements Analysis and Negotiation

Once requirements have been gathered, the work products noted earlier form the basis for requirements analysis. Analysis categorizes requirements and organizes them into related subsets; explores each requirement in relationship to others; examines requirements for consistency, omissions, and ambiguity; and ranks requirements based on the needs of customers/users.
Requirements Specification

In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a graphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

The System Specification is the final work product produced by the system and requirements engineer. It serves as the foundation for hardware engineering, software engineering, database engineering, and human engineering. It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification bounds each allocated system element. The System Specification also describes the information (data and control) that is input to and output from the system.

System Modeling

Assume for a moment that you have been asked to specify all requirements for the construction of a gourmet kitchen. You know the dimensions of the room, the location of doors and windows, and the available wall space. You could specify all cabinets and appliances and even indicate where they are to reside in the kitchen. Would this be a useful specification? The answer is obvious. In order to fully specify what is to be built, you would need a meaningful model of the kitchen, that is, a blueprint or three-dimensional rendering that shows the position of the cabinets and appliances and their relationship to one another. From the model, it would be relatively easy to assess the efficiency of work flow (a requirement for all kitchens), the aesthetic “look” of the room (a personal, but very important requirement). Rough estimates of development effort are made and used to assess the impact of each requirement on project cost and delivery time.
Using an iterative approach, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

**Requirements Specification**

A specification can be a written document, a graphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these. Some suggest that a “standard template” should be developed and used for a system specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. The *System Specification* is the final work product produced by the system and requirements engineer. It serves as the foundation for hardware engineering, software engineering, database engineering, and human engineering. It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification bounds each allocated system element. The *System Specification* also describes the information (data and control) that is input to and output from the system.

**Requirements Validation**

The work products produced as a consequence of requirements engineering (a system specification and related information) are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product. The primary requirements validation mechanism is the formal technical review. The review team includes system engineers, customers, users, and other stakeholders who examine the system specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

**Requirements Management**

*Requirements management* is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project
proceeds. Like SCM, requirements management begins with identification. *Features traceability table.* Shows how requirements relate to important customer observable system/product features.

*Source traceability table.* Identifies the source of each requirement.

*Dependency traceability table.* Indicates how requirements are related to one another.

*Subsystem traceability table.* Categorizes requirements by the subsystem(s) that they govern.

*Interface traceability table.* Shows how requirements relate to both internal and external system interfaces.

![Traceability Table Example](image)

**Feasibility studies**

A feasibility study decides whether or not the proposed system is worthwhile. A short focused study that checks

- If the system contributes to organisational objectives.
- If the system can be engineered using current technology and within budget.
- If the system can be integrated with other systems that are used

Based on information assessment (what is required), information collection and report writing Questions for people in the organisation

- What if the system wasn’t implemented?
What are current process problems?
How will the proposed system help?
What will be the integration problems?
Is new technology needed? What skills?
What facilities must be supported by the proposed system?

**Elicitation and analysis**
- Sometimes called requirements elicitation or requirements discovery.
- It involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system’s operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called stakeholders.

**Self assessment Question:**
- What are the problem make elicitations difficult?

**Chapter 4 SOFTWARE PROTOTYPING**
Analysis should be conducted regardless of the software engineering paradigm that is applied. However, the form that analysis takes will vary. In some cases it is possible to apply operational analysis principles and derive a model of software from which a design can be developed. Finally, some circumstances require the construction of a prototype at the beginning of analysis, since the model is the only means through which requirements can be effectively derived. The model then evolves into production software.

**Selecting the Prototyping Approach**
The prototyping paradigm can be either close-ended or open-ended. The close-ended approach is often called *throwaway prototyping*. Using this approach, a prototype serves solely as a rough demonstration of requirements. It is then discarded, and the software is engineered using a different paradigm. An open-ended approach, called *evolutionary*
prototyping, uses the prototype as the first part of an analysis activity that will be continued into design and construction. The prototype of the software is the first evolution of the finished system. Before a close-ended or open-ended approach can be chosen, it is necessary to determine whether the system to be built is amenable to prototyping. If a candidate application (one that has the characteristics noted) will require the development of tens of thousands of lines of code before any demonstrable function can be performed, it is likely to be too complex for prototyping. If, however, the complexity can be partitioned, it may still be possible to prototype portions of the software. Because the customer must interact with the prototype in later steps, it is essential that the

1. Customer resources be committed to the evaluation and refinement of the prototype,
2. The customer is capable of making requirements decisions in a timely fashion.

**Prototyping Methods and Tools**

For software prototyping to be effective, a prototype must be developed rapidly so that the customer may assess results and recommend changes. To conduct rapid prototyping, three generic classes of methods and tools are available:

**Fourth generation techniques.** Fourth generation techniques (4GT) encompass a broad array of database query and reporting languages, program and application generators, and other very high-level nonprocedural languages. Because 4GT enable the software engineer to generate executable code quickly, they are ideal for rapid prototyping.

**Reusable software components.** Another approach to rapid prototyping is to assemble, rather than build, the prototype by using a set of existing software components. Melding

<table>
<thead>
<tr>
<th>Question</th>
<th>Throwaway prototype</th>
<th>Evolutionary prototype</th>
<th>Additional preliminary work required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is the application domain understood?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can the problem be modeled?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Is the customer certain of basic system requirements?</td>
<td>Yes/No</td>
<td>Yes/No</td>
<td>No</td>
</tr>
<tr>
<td>Are requirements established and stable?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Are requirements ambiguous?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Are there contradictions in the requirements?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
prototyping and program component reuse will work only if a library system is developed so that components that do exist can be cataloged and then retrieved. It should be noted that an existing software product can be used as a prototype for a "new, improved" competitive product. In a way, this is a form of reusability for software prototyping.

**Formal specification and prototyping environments.** Over the past two decades, a number of formal specification languages and tools have been developed as a replacement for natural language specification techniques. Today, developers of these formal languages are in the process of developing interactive environments that

1. Enable an analyst to interactively create language-based specifications of a system or software,
2. Invoke automated tools that translate the language-based specifications into executable code, and
3. Enable the customer to use the prototype executable code to refine formal requirements.

**DATA MODELING**

Data modeling answers a set of specific questions that are relevant to any data processing application:

What are the primary data objects to be processed by the system?
What is the composition of each data object and what attributes describe the object?
Where do the objects currently reside?
What are the relationships between each object and other objects?
What are the relationships between the objects and the processes that transform them? To answer these questions, data modeling methods make use of the entity relationship diagram. The ERD, described in detail later in this section, enables software engineer to identify data objects and their relationships using a graphical notation. In the context of structured analysis, the ERD defines all data that are entered, stored, transformed, and produced within an application. The entity relationship diagram focuses solely on data (and therefore satisfies the first operational analysis principles), representing a "data network" that exists for a given system. The ERD is especially useful for applications in which data and the relationships that govern data is complex. Unlike the data flow it is
used to represent how data are transformed), data modeling considers data independent of the processing that transforms the data. Data Objects, Attributes, and Relationships The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

**Data objects.** A *data object* is a representation of almost any composite information that must be understood by software. By *composite information*, we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object. A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., a salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes. Data objects (represented in bold) are related to one another. For example, person can *own* car, where the relationship *own* connotes a specific "connection" between Person and car. The relationships are always defined by the context of the problem that is being analyzed. A data object encapsulates data only—there is no reference within a data objects to operations that act on the data. Therefore, the data object can be represented as a table. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.

**Attributes.** Attributes define the properties of a data object and take on one of three different characteristics. They can be used to

1. Name an instance of the data object,
2. Describe the instance, or
3. Make reference to another instance in another table. In addition, one or more of the attributes must be defined as an *identifier*—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the
identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number. The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for car might serve well for an application that would be used by a Department of Motor Vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for car might also include ID number, body type and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

Relationships. Data objects are connected to one another in different ways. Consider two data objects, book and bookstore. A connection is established between book and bookstore because the two objects are related. But what are the relationships?

To determine the answer, we must understand the role of books and bookstores within the context of the software to be built. We can define a set of object/relationship pairs that define the relevant relationships. For example,

• A bookstore orders books.
• A bookstore displays books.
• A bookstore stocks books.
• A bookstore sells books.
• A bookstore returns books.
The relationships orders, displays, stocks, sells, and returns define the relevant connections between book and bookstore. Figure 12.4b illustrates these object/relationship pairs graphically. It is important to note that object/relationship pairs are bidirectional. That is, they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.

Cardinality and Modality
The elements of data modeling—data objects, attributes, and relationships—provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood. We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states: object X relates to object Y does not provide enough information for software engineering purposes. We must understand how many occurrences of object X are related to how many occurrences of object Y. This leads to a data modeling concept called cardinality.

Cardinality. The data model must be capable of representing the number of occurrences objects in a given relationship. Tillman defines the cardinality of an object/relationship pair in the following manner: Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object]. Cardinality is usually expressed as simply 'one' or 'many.' For example, a husband can have only one wife (in most cultures), while a parent can have many...
children. Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as

- One-to-one (1:1)—An occurrence of [object] 'A' can relate to one and only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'
- One-to-many (1:N)—One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.' For example, a mother can have many children, but a child can have only one mother.
- Many-to-many (M:N)—An occurrence of [object] 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.' For example, an uncle can have many nephews, while a nephew can have many uncles.

Cardinality defines “the maximum number of objects that can participate in a relationship.” It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

**Modality.** The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. That is, a single customer can be provided with zero or many repair actions. The symbols on the relationship connection closest to the data object rectangles indicate cardinality. The vertical bar indicates one and the three-pronged fork indicates many. Modality is indicated by the symbols that are further away from the data object rectangles. The second vertical bar on the left indicates that there must be a customer for a repair action to occur. The circle on the right indicates that there may be no repair action required for the type of problem reported by the customer. The object/relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the entity/relationship diagram. The ERD was originally proposed by Peter Chen [CHE77]
for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality. The relationship between the data objects car and manufacturer would be represented. One manufacturer builds one or many cars.

Given the context implied by the ERD, the specification of the data object car (data object table in Figure) would be radically different from the earlier specification. By examining the symbols at the end of the connection line between objects, it can be seen that the modality of both occurrences is mandatory (the vertical lines). Expanding the model, we represent a grossly oversimplified ERD of the distribution element of the
automobile business. New data objects, shipper and dealership, are introduced. For example, the data object car can be categorized as domestic, European, or Asian. The ERD notation represents this categorization in the form of a hierarchy. ERD notation also provides a mechanism that represents the associativity between objects. In the associative data object, each of the data objects that model the individual subsystems is associated with the data object car.

Keywords:
Cardinality, modality, data object, throwaway prototyping, evolutionary prototyping

Objective type Questions:
1) __________ is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].
   a).Cardinality  b).Modality
2) The __________ of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional.
   a).Cardinality  b).Modality
3) __________ are connected to one another in different ways.
4) A __________ is a representation of almost any composite information that must be understood by software.
6) The close-ended approach is often called __________.
   a). Evolutionary prototyping  b). Throwaway prototyping
7) An open-ended approach, called __________, uses the prototype as the first part of an analysis activity that will be continued into design and construction.
   a). Evolutionary prototyping  b). Throwaway prototyping
Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies hardware, software, and human elements to transform it; and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output may light a single LED or produce a 200-page report. In effect, we can create a flow model for any computer-based system, regardless of size and complexity. Structured analysis began as an information flow modeling technique. A computer-based system is represented as an information transform. A rectangle is used to represent an external entity; that is, a system element (e.g., hardware, a person, another program) or another system that produces information for transformation by the software or receives information produced by the software. A circle (sometimes called a bubble) represents a process or transform that is applied to data (or control) and changes it in some way. An arrow represents one or more data items (data objects). All arrows on a data flow diagram
should be labeled. The double line represents a data store—stored information that is used by the software. The simplicity of DFD notation is one reason why structured analysis techniques are widely used.

It is important to note that no explicit indication of the sequence of processing or conditional logic is supplied by the diagram.

**Data Flow Diagrams**

As information moves through software, it is modified by a series of transformations. A *data flow diagram* is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram, also known as a *data flow graph* or a *bubble chart*. The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Therefore, the DFD provides a mechanism for functional modeling as well as information flow modeling. In so doing, it satisfies the second operational analysis principle (i.e., creating a functional model). A level 0 DFD, also called a *fundamental system model* or a *context model*, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. Additional processes (bubbles) and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example, a level 1 DFD might contain five or six bubbles with interconnecting arrows. Each of the processes represented at level 1 is a sub function of the overall system depicted in the context model. As we noted earlier, each of the bubbles may be refined or layered to depict more detail. A fundamental model for system $F$ indicates the primary input is $A$ and ultimate output is $B$. We refine the $F$ model into transforms $f1$ to $f7$. Note that *information flow*
continuity must be maintained; that is, input and output to each refinement must remain the same. This concept, sometimes called balancing, is essential for the development of consistent models. Further refinement of $f4$ depicts detail in the form of transforms $f41$ to $f45$. Again, the input ($X, Y$) and output ($Z$) remain unchanged. The basic notation used to develop a DFD is not in itself sufficient to describe requirements for software. For example, an arrow shown in a DFD represents a data object that is input to or output from a process. A data store represents some organized collection of data. But what is the content of the data implied by the arrow or depicted by the store? DFD graphical notation must be augmented with descriptive text. A process specification (PSPEC) can be used to specify the processing details implied by a bubble within a DFD. The process specification describes the input to a function, the algorithm that is applied to transform the input, and the output that is produced. In addition, the PSPEC indicates restrictions and limitations imposed on the process (function), performance characteristics that are relevant to the process, and design constraints that may influence the way in which the process will be implemented.
BEHAVIORAL MODELING

Behavioral modeling is an operational principle for all requirements analysis methods. The state transition diagram represents the behavior of a system by depicting its states and the events that cause the system to change state. In addition, the STD indicates what actions (e.g., process activation) are taken as a consequence of a particular event. A state is any observable mode of behavior. For example, states for a monitoring and control system for pressure vessels. Each of these states represents a mode of behavior of the system. A state transition diagram indicates how the system moves from state to state.

To illustrate the use of the Hatley and Pirbhai control and behavioral extensions, consider software embedded within an office photocopying machine. Data flow arrows have been lightly shaded for illustrative purposes, but in reality they are not shown as part of a control flow diagram.

Control flows are shown entering and exiting individual processes and the vertical bar representing the CSPEC "window." For example, the paper feed status and start/stop events flow into the CSPEC bar. This implies that each of these events will cause some process represented in the CFD to be activated. If we were to examine the CSPEC
internals, the **start/stop** event would be shown to activate/deactivate the *manage copying* process. Similarly, the **jammed** event (part of *paper feed status*) would activate *perform problem diagnosis*. It should be noted that all vertical bars within the CFD refer to the same CSPEC. An event flow can be input directly into a process as shown with **repro fault**. However, this flow does not activate the process but rather provides control information for the process algorithm. The rectangles represent system states and the arrows represent transitions between states. Each arrow is labeled with a ruled expression. The top value indicates the event(s) that cause the transition to occur. The bottom value indicates the action that occurs as a consequence of the event. Therefore, when the paper tray is **full** and the **start** button is pressed, the system moves from the *reading commands* state to the *making copies* state. Note that states do not necessarily correspond to processes on a one-to-one basis. For example, the state *making copies* would encompass both the *manage copying* and *produce user displays* processes.

**THE MECHANICS OF STRUCTURED ANALYSIS**

To be used effectively in software requirements analysis, this notation must be combined with a set of heuristics that enable a software engineer to derive a good analysis model. To illustrate the extensions to the basic structured analysis notation will be used throughout. In the sections that follow, we examine each of the steps that should be applied to develop complete and accurate models using structured analysis. Through this discussion, the notation will be used, and other notational forms, alluded to earlier, will be presented in some detail.

**Creating an Entity/Relationship Diagram**

The entity/relationship diagram enables a software engineer to fully specify the data objects that are input and output from a system, the attributes that define the properties of these objects, and their relationships. Like most elements of the analysis model, the ERD is constructed in an iterative manner. The following approach is taken:

**1.** During requirements elicitation, customers are asked to list the “things” that the application or business process addresses. These “things” evolve into a list of input and output data objects as well as external entities that produce or consume information.
2. Taking the objects one at a time, the analyst and customer define whether or not a connection (unnamed at this stage) exists between the data object and other objects.

3. Wherever a connection exists, the analyst and the customer create one or more object/relationship pairs.

4. For each object/relationship pair, cardinality and modality are explored.

5. Steps 2 through 4 are continued iteratively until all object/relationships have been defined. It is common to discover omissions as this process continues. New objects and relationships will invariably be added as the number of iterations grows.

6. The attributes of each entity are defined.

7. An entity relationship diagram is formalized and reviewed.

8. Steps 1 through 7 are repeated until data modeling is complete.

Creating a Data Flow Model

The data flow diagram enables the software engineer to develop models of the information domain and functional domain at the same time.

A few simple guidelines can aid immeasurably during derivation of a data flow diagram:

1. The level 0 data flow diagram should depict the software/system as a single bubble;
2. Primary input and output should be carefully noted;
3. Refinement should begin by isolating candidate processes, data objects, and stores to be represented at the next level;
4. All arrows and bubbles should be labeled with meaningful names;
5. Information flow continuity must be maintained from level to level, and
6. One bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram. This occurs when the analyst attempts to show too much detail too early or represents procedural aspects of the software in lieu of information flow.

The Process Specification

The process specification (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm, mathematical equations, tables, diagrams, or charts. By providing a PSPEC
to accompany each bubble in the flow model, the software engineer creates a "minispec" that can serve as a first step in the creation of the *Software Requirements Specification* and as a guide for design of the software component that will implement the process.

**SUMMARY:**

Structured analysis, a widely used method of requirements modeling, relies on data modeling and flow modeling to create the basis for a comprehensive analysis model. Using entity-relationship diagrams, the software engineer creates a representation of all data objects that are important for the system. Data and control flow diagrams are used as a basis for representing the transformation of data and control. At the same time, these models are used to create a functional model of the software and to provide a mechanism for partitioning function. A behavioral model is created using the state transition diagram and data content is developed with a data dictionary. Process and control specifications provide additional elaboration of detail. The original notation for structured analysis was developed for conventional data processing applications, but extensions have made the method applicable to real time systems. Structured analysis is supported by an array of CASE tools that assist in the creation of each element of the model and also help to ensure consistency and correctness.

**Keywords:**

- Data Flow Diagram
- Behavioral Modeling
- A Computer-Based System
- Process Specification (Pspec)
- Entity/Relationship Diagram
- Data Flow Graph
- A Bubble Char.

**Objective type Questions:**

1) The __________ enables the software engineer to develop models of the information domain and functional domain at the same time.
2) The ____________ is used to describe all flow model processes that appear at the final level of refinement.

3) The ____________ enables a software engineer to fully specify the data objects that are input and output from a system, the attributes that define the properties of these objects, and their relationships.

4) ____________ is an operational principle for all requirements analysis methods.

5) A ____________ is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output.

6) The basic form of a data flow diagram, also known as a _____________.
   a). data flow graph   b). bubble char.   c). Both a and b

7) Information is transformed as it flows through a _____________.

Chapter 6  
THE DATA DICTIONARY

Definition: A data dictionary is a table of information about each data element in the system. Initially, in the requirements phase the data dictionary will be the data items from the problem domain. A typical entry will include the name of the item, in which class it is located, the type of the data item, and the semantics of the item.

- Data dictionaries are lists of all of the names used in the system models.
- Descriptions of the entities, relationships and attributes are also included
- Advantages
  - Support name management and avoid duplication
  - Store of organizational knowledge linking analysis, design and implementation
Many CASE workbenches support data dictionaries

DESCRIPTION:
The analysis model encompasses representations of data objects, function, and control. In each representation data objects and/or control items play a role. Therefore, it is necessary to provide an organized approach for representing the characteristics of each data object and control item. This is accomplished with the data dictionary. The data dictionary is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations. Today, the data dictionary is always implemented as part of a CASE "structured analysis and design tool." Although the format of dictionaries varies from tool to tool, most contain the following information:

- **Name**—the primary name of the data or control item, the data store or an external entity.
- **Alias**—other names used for the first entry.
- **Where-used/how-used**—a listing of the processes that use the data or control item and how it is used (e.g., input to the process, output from the process, as a store, as an external entity.
- **Content description**—a notation for representing content.
- **Supplementary information**—other information about data types, preset values (if known), restrictions or limitations, and so forth. Once a data object or control item name and its aliases are entered into the data dictionary, consistency in naming can be enforced. That is, if an analysis team member decides to name a newly derived data item **xyz**, but **xyz** is already in the dictionary, the CASE tool supporting the dictionary posts a warning to indicate duplicate names. This improves the consistency of the analysis model and helps to reduce errors.

“Where-used/how-used” information is recorded automatically from the flow models. When a dictionary entry is created, the CASE tool scans DFDs and CFDs to determine which processes use the data or control information and how it is used. Although this may appear unimportant, it is actually one of the most
important benefits of the dictionary. During analysis there is an almost continuous stream of changes.

**Sample data dictionary module:**

The data dictionary entry begins as follows:

Name: telephone number

Aliases: none

Where used/how used: assess against set-up (output)

Dial phone (input)

Description:

Telephone number = [local number|long distance number]

Local number = prefix + access number

Long distance number = 1 + area code + local number

Area code = [800 | 888 | 561]

Prefix = *a three digit number that never starts with 0 or 1*

Access number = * any four number string *

**SUMMARY**

The content description is expanded until all composite data items have been represented as elementary items (items that require no further expansion) or until all composite items are represented in terms that would be well-known and unambiguous to all readers. It is also important to note that a specification of elementary data often restricts a system. For example, the definition of area code indicates that only three area codes (two toll-free and one in South Florida) are valid for this system.

The data dictionary defines information items unambiguously. Although we might assume that the telephone number represented by the DFD in Figure 12.22 could accommodate a 25-digit long distance carrier access number, the data dictionary content description tells us that such numbers are not part of the data that may be used.
Self assessment:

Build a data dictionary for the library problem:

Self assessment Solution:

<table>
<thead>
<tr>
<th>Name</th>
<th>Class</th>
<th>Type</th>
<th>Size</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>Book</td>
<td>String</td>
<td>&lt; 40 char</td>
<td>Last name, first name (may be truncated)</td>
</tr>
<tr>
<td>Book</td>
<td>Book</td>
<td>Object</td>
<td></td>
<td>Abstract concept of the book</td>
</tr>
<tr>
<td>Book ID</td>
<td>Copy</td>
<td>Key</td>
<td></td>
<td>Key to info about the book</td>
</tr>
<tr>
<td>Borrower</td>
<td>Loan</td>
<td>Key</td>
<td></td>
<td>Key to patron who made this loan</td>
</tr>
<tr>
<td>Copy</td>
<td>Copy</td>
<td>Object</td>
<td></td>
<td>Library’s physical copy of a book</td>
</tr>
<tr>
<td>Copy ID</td>
<td>Copy</td>
<td>Key</td>
<td></td>
<td>Key to physical copy being borrowed</td>
</tr>
<tr>
<td>Loan</td>
<td>Loan</td>
<td>Object</td>
<td></td>
<td>A borrowing that is still active</td>
</tr>
<tr>
<td>Name</td>
<td>Patron</td>
<td>String</td>
<td>&lt; 40 char</td>
<td>Last name, first name (may be truncated)</td>
</tr>
<tr>
<td>Patron</td>
<td>Patron</td>
<td>Object</td>
<td></td>
<td>Registered holder of library card</td>
</tr>
<tr>
<td>Title</td>
<td>Book</td>
<td>String</td>
<td>&lt; 50 char</td>
<td>First 50 char of title from title page</td>
</tr>
</tbody>
</table>

Important Questions:

1. State and explain requirement engineering tasks in detail.
2. Describe the primary difference between structured analyses and object oriented analysis.
3. Write a detailed note on scenario based modeling.
4. Describe functional and non functional requirements in detail.
5. Define data dictionary.
6. Explain about software prototyping.