Software Requirements Engineering

1.0 INTRODUCTION

In the software development process, requirements phase is the first software engineering activity, which translates the ideas or views into a requirements document. This phase is user-dominated phase. Defining and documenting the user requirements in a concise and unambiguous manner is the first major step to achieve a high quality product. Requirements phase encompasses a set of tasks, which helps to specify the impact of the software on the organisation, customers' needs, and how users will interact with the developed software. The requirements are the basis of system design. If requirements are not correct the end product will also contain errors. Note that requirement activity like all other software engineering activities should be adapted to the needs of the process, the project, the product, and the people involved in the activity. Also, the requirements should be specified at different levels of detail. This is because requirements are meant for (such as users, managers, system engineers, and so on). For example, managers may be interested in knowing how the system is implemented rather than knowing the detailed features of the system. Similarly, end-users are interested in knowing whether the specified requirements meet their desired needs or not.

1.1 UNIT OBJECTIVES

After reading this unit, the reader will understand:

- What is Software Requirement?
- Feasibility study, which includes technical, operational, and economic feasibility
- Requirements elicitation, which is a process of collecting information about software requirements from different individuals.
- How requirements analysis helps to understand, interpret, classify, and organize the software requirements.
- Requirements document that lays a foundation for software engineering activities and is created when entire requirements are elicited and analyzed.
- Requirements validation phase where work products are examined for consistency, omissions, and ambiguity.
- Requirements management phase which can be defined as a process of eliciting, documenting, organizing and controlling changes to the requirements.

1.2 WHAT IS SOFTWARE REQUIREMENT?

Requirements are description of the services provided by the system and its operational constraints reflecting needs of the customer that helps to solve their problem. Requirement is a condition or a capability possessed by software or system component in order to solve a real world problem. The problems can be to automate a part of a system, to correct shortcomings of an existing system, to control a device, and so on. IEEE defines requirement as "(1) *A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system* *or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3)A documented representation of a condition or capability as in (1) or (2)*".

Requirements describe how a system should act, appear, or perform. For this, when users request for software, they possess an approximation of what the new system should be capable of doing. Requirements differ from one user to another user and from one business process to another business process.
Note: *Information about requirements is stored in a database, which helps software development team to understand user requirements and develop software according to those requirements.*

Therefore **Requirements Engineering** is the process of finding out, analyzing, documenting and checking services and operational constraints called requirements.

1.2.1 Guidelines for Expressing Requirements

The purpose of the requirements document is to provide a basis for the mutual understanding between users and designers of the initial definition of the software development life cycle (SDLC), including the requirements, operating environment, and development plan. The requirements document should include, in the overview, the proposed methods and procedures, a summary of improvements, a summary of impacts, security, privacy, and internal control considerations, cost considerations and alternatives. The requirements section should state the functions required of the software in quantitative and qualitative terms, and how these functions will satisfy the performance objectives. The requirements document should also specify the performance requirements, such as accuracy, validation, timing, and flexibility. Inputs and outputs need to be explained, as well as data characteristics. Finally, the requirements document needs to describe the operating environment and provide, or make reference to, a development plan. There is no standard method to express and document the requirements. Requirements can be stated efficiently by the experience of knowledgeable individuals, observing past requirements, and by following guidelines. Guidelines act as an efficient method of expressing requirements, which also provide a basis for software development, system testing, and user satisfaction. The guidelines that are commonly followed to document requirements are listed below:

- Sentences and paragraphs should be short and written in active voice. Also, proper grammar, spelling, and punctuation should be used.
- Conjunctions, such as 'and' and 'or' should be avoided as they indicate the combination of several requirements in one requirement.
- Each requirement should be stated only once so that it does not create redundancy in the requirements specification document.

1.2.2 Types of Requirements

The requirements, which are commonly considered, are classified into three categories, namely, functional requirements, nonfunctional requirements, and domain requirements.

(***a***) ***Functional Requirements:*** The functional requirements (also known as **behavioral requirements**) describe the functionality or services that software should provide. For this functional requirements describe the interaction of software with its environment and specify the inputs, outputs, external interfaces, and the functions that should not be included in the software. Also, the services provided by functional requirements specify the procedure by which the software should react to particular inputs or behave in particular situations. IEEE defines functional requirements as "*a function that a system or component must be able to perform*."

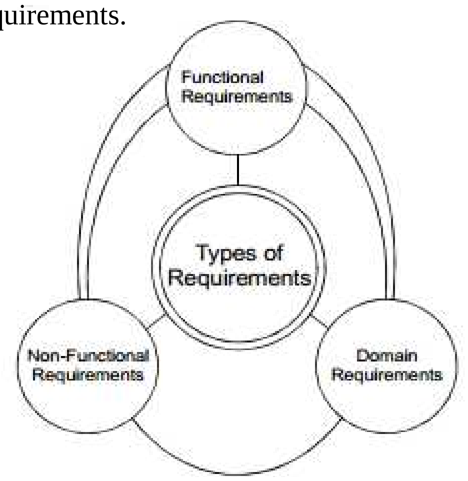


Figure 1.1 Types of requirements

To understand functional requirements properly, let us consider an example of an online banking system, which is listed below:

• The user of the bank should be able to search the desired services from the available services.
• There should be appropriate documents for users to read. This implies that when a user wants to open an account in the bank, the forms must be available so that the user can open an account.
• After registration, the user should be provided with a unique acknowledgement number so that the user can later be given an account number.

The above-mentioned functional requirements describe the specific services provided by the online banking system. These requirements indicate user requirements and specify that functional requirements may be described at different levels of detail in online banking system. With the help of these functional requirements, users can easily view, search, and download registration forms and other information about the bank. On the other hand, if requirements are not stated properly, then they are misinterpreted by the software engineers and user requirements are not met.

The functional requirements should be complete and consistent. **Completeness** implies that all the user requirements are defined. **Consistency** implies that all requirements are specified clearly without any contradictory definition. Generally, it is observed that completeness and consistency cannot be achieved in large software or in a complex system due to the errors that arise while defining the functional requirements of these systems.

(***b***) ***Non-functional Requirements:*** The non-functional requirements (also known as **quality requirements**) relate to system attributes, such as reliability and response time. Non-functional requirements arise due to user requirements, budget constraints, organizational policies, and so on. These requirements are not related directly to any particular function provided by the system. Non-functional requirements should be accomplished in software to make it perform efficiently. For example, if an airplane is unable to fulfill reliability requirements, it is not approved for safe operation. Similarly, if a real time control system is ineffective in accomplishing non-functional requirements, the control functions cannot operate correctly.

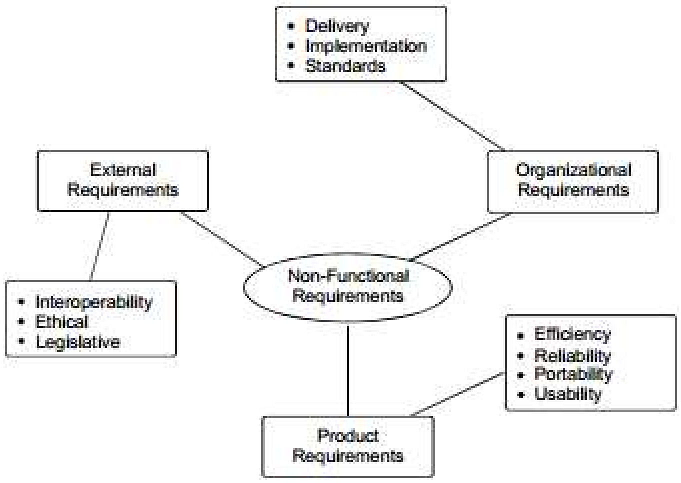


Figure 1.2 types of Non-functional Requirements

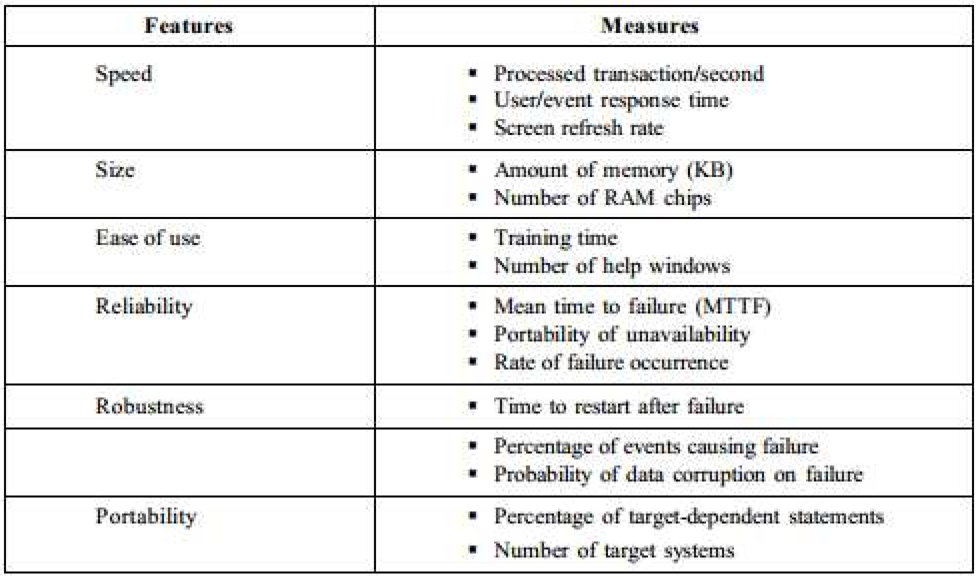
Different types of non-functional requirements are shown in the above Figure. The description of these requirements is listed below:

- **Product requirements:** These requirements specify how software product performs. Product requirements comprise of the following:
    - **Efficiency requirements:** Describe the extent to which software makes optimal use of resources, the speed with which system executes, and the memory it consumes for its operation. For example, system should be able to operate at least three times faster than the existing system.
    - **Reliability requirements:** Describe the acceptable failure rate of the software. For example, software should be able to operate even if a hazard occurs.
    - **Portability requirements:** Describe the ease with which software can be transferred from one platform to another. For example, it should be easy to port software to different operating system without the need to redesign the entire software.
    - **Usability requirements:** Describe the ease with which users are able to operate the software. For example, software should be able to provide access to functionality with fewer keystrokes and mouse clicks.
- **Organizational requirements:** These requirements are derived from the policies and procedures of an organization. Organizational requirements comprise of the following:
    - **Delivery requirements:** Specify when software and its documentation are to be delivered to the user.
    - **Implementation requirements:** Describe requirements, such as programming language and design method.
    - **Standards requirements:** Describe the process standards to be used during software development. For example, software should be developed using standards specified by ISO (International Organization for Standardization) and IEEE standards.

- **External requirements**: These requirements include all the requirements that affect the software or its development process externally. External requirements comprise of the following:
  - **Interoperability requirements:** Define the way in which different computer-based systems interact with each other in one or more organizations.
  - **Ethical requirements:** Specify the rules and regulations of the software so that they are acceptable to users.
  - **Legislative requirements:** Ensure that software operates within the legal jurisdiction. For example, pirated software should not be sold.

Non-functional requirements are difficult to verify. Hence, it is essential to write nonfunctional requirements quantitatively so that they can be tested. For this, non-functional requirements metrics are used. These metrics are listed in the Table 1.1.

Table 1.1 Metrics for Non-functional Requirements



Features	Measures
Speed	▪ Processed transaction/second ▪ User/event response time ▪ Screen refresh rate
Size	▪ Amount of memory (KB) ▪ Number of RAM chips
Ease of use	▪ Training time ▪ Number of help windows
Reliability	▪ Mean time to failure (MTTF) ▪ Portability of unavailability ▪ Rate of failure occurrence
Robustness	▪ Time to restart after failure
	▪ Percentage of events causing failure ▪ Probability of data corruption on failure
Portability	▪ Percentage of target-dependent statements ▪ Number of target systems

(*c*) ***Domain Requirements:*** Requirements derived from the application domain of a system, instead from the needs of the users are known as **domain requirements**. These requirements may be new functional requirements or specify a method to perform some particular computations. In addition, these requirements include any constraint that may be present in existing functional requirements. As domain requirements reflect fundamentals of the application domain, it is important to understand these requirements. Also, if these requirements are not fulfilled, it may be difficult to make the system work as desired. A system can include a number of domain requirements. For example, a system may comprise of design constraint that describes the user interface, which is capable of accessing all the databases used in a system. It is important for a development team to create databases and interface design as per established standards. Similarly, the requirements requested by the user, such as copyright restrictions and security mechanism for the files and documents used in the system are also domain requirements.

When domain requirements are not expressed clearly, it can result in various problems, such as:

- **Problem of understandability:** When domain requirements are specified in the language of application domain (such as mathematical expressions), it becomes difficult for software engineers to understand these requirements.
- **Problem of implicitness:** When domain experts understand the domain requirements but do not express these requirements clearly, it may create a problem (due to incomplete information) for the development team to understand and implement the requirements in the system.

Requirements can also be classified based on the level of detail they contain while they are specified.

User Requirements: this is more abstract and written for readers having no technical knowledge. The statements are in natural language plus diagrams of what services the system is expected to provide and constraints under which it must operate.

System requirements: this is more precise and has detailed explanation of system functions and services. Since readers of system requirements are involved in system implementation they need to know more precisely what the system will do.

Requirements for Critical Systems

Critical systems are

- Systems whose '<u>failure</u>' causes significant economic, physical or human damage to organizations or people.
- There are <u>three</u> principal types of critical system:
  - ‹ Business critical systems: Systems whose Failure leads to significant economic damage.
  - Mission critical systems: Systems whose Failure leads to the abortion of a mission.
  - ‹ Safety critical systems: Systems whose Failure endangers human life.
- The cost of failure in Critical Systems is likely to exceed the cost of the system itself.

Examples - Communication systems such as telephone switching systems, aircraft radio systems, etc.

- Embedded control systems such as air-traffic control systems, disaster management systems, etc.
- Financial systems such as foreign exchange transaction systems, account management systems, etc.

Criticality attributes

- Reliability
  - Number of times a system fails to deliver the specified services.
  - Metrics used
    - MTTF –Mean Time To Failure

        - Time between observed system failures
    - ROCOF –Rate Of Occurrence Of Failure
        - number of failures in a given time period
- Availability (especially for nonstop systems)
  - Determine how likely the system is available to meet a demand for service.
  - Metrics used
    - POFOD –Probability Of Failure On Demand
        - Probability that the system will fail to complete a service request
- Maintainability
  - The ease of repairing the system after failure has been discovered or changing the system to include new features.
  - No useful metrics – judged intuitively
- Security
  - The ability of the system to protect itself and its data from deliberate or accidental damage.
- Safety
  - Ability to deliver its services in such away the human life and its environment will not be damaged by the system.
  - E.g. loss of 911 service will result in injury or death

Characteristics of Good Requirements

- Complete- Functionality is described completely
- Correct- According to stakeholders‘ intentions
- Consistent- There are no two requirements that cannot be reached in one single system
- Checkable- Requirements can be used to generate tests on the final software
- Comprehensible- Requirements are to be understood by all stakeholders
- Necessary- Requirement should be needed by customer and user
- Well-Defined- Requirement can only be understood in one single way and Leaves no space for interpretation
- Traceable

1.2.3 Requirements Engineering Process

Requirements engineering (RE) process is a series of activities that are performed in requirements phase in order to express requirements in software requirements specification (SRS) document. This process focuses on understanding the requirement and its type so that an appropriate technique is determined to carry out the requirements engineering process. The new software developed after collecting requirements either replaces the existing software or enhances its features and functionality. For example, the payment mode of existing software can be changed from payment through hand-written cheques to electronic payment of bills.

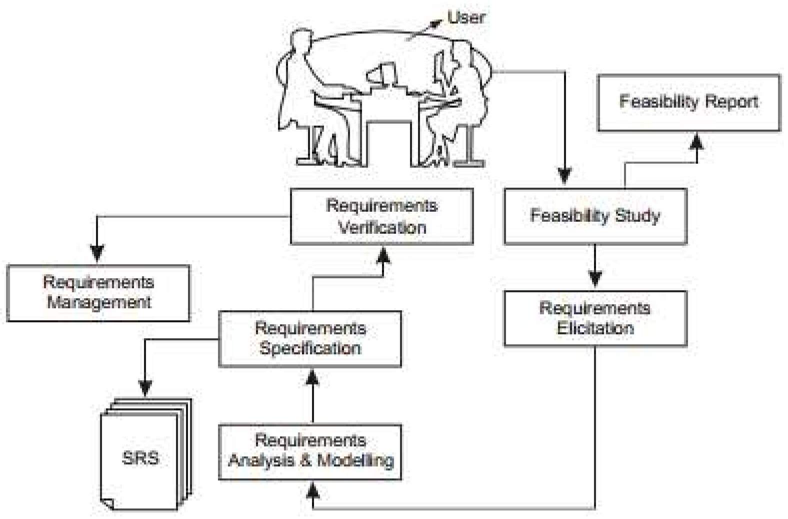


Figure 1.3 Requirements Engineering Process

In the Figure 1.3, a requirements engineering process is shown, which comprises of various steps. These steps include *feasibility study*, *requirements elicitation*, *requirements analysis*, *requirements specification*, *requirements validation*, and *requirements management*. Requirements engineering process begins with feasibility study of the requirements. Then, requirements elicitation is performed, which focuses on gathering user requirements. After the requirements are gathered, analysis is performed, which further leads to requirements specification. The output of this is stored in the form of software requirements specification document. Next, the requirements are checked for their completeness and correctness in requirements validation. Lastly, to understand and control changes to system requirements, requirements management is performed.

1.3 FEASIBILITY STUDY

Feasibility is defined as the practical extent to which a project can be performed successfully. To evaluate feasibility, a **feasibility study** is performed, which determines whether the solution considered to accomplish the requirements is practically workable in the software or not. For this, it considers information, such as resource availability, cost estimates for software development, benefits of software to organization after it is developed, and cost to be incurred on its maintenance.

The objective of feasibility study is to establish the reasons for developing software that is acceptable to users, adaptable to change, and conformable to established standards. Various other objectives of feasibility study are listed below:

- Analyze whether the software will meet organizational requirements or not.
- Determine whether the software can be implemented using current technology and within the specified budget and schedule or not.
- Determine whether the software can be integrated with other existing software or not.

1.3.1 Types of Feasibility

Various types of feasibility that are commonly considered include *technical feasibility*, *operational feasibility*, and *economic feasibility*.

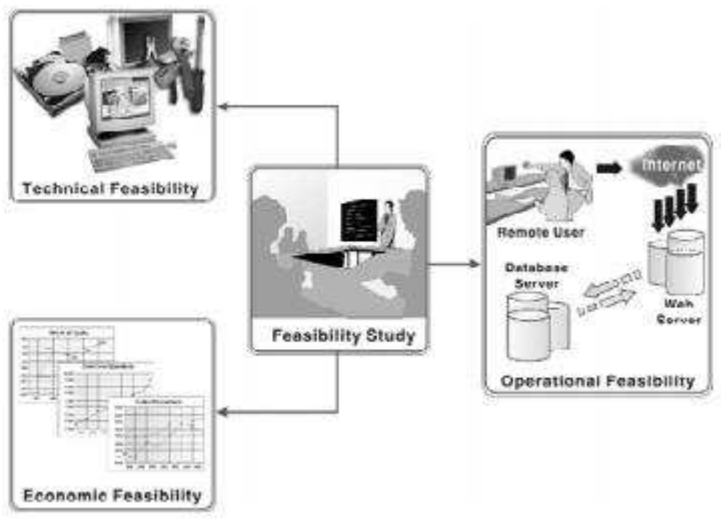


Figure 1.4 Types of Feasibility

***(a) Technical Feasibility*:** Technical feasibility assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget. For this, software development team ascertains whether the current resources and technology can be upgraded or added in the software to accomplish specified user requirements. Technical feasibility performs the tasks listed below:
- Analyzes the technical skills and capabilities of software development team members.
- Determines whether the relevant technology is stable and established or not.
- Ascertains that the technology chosen for software development has large number of users so that they can be consulted when problems arise or improvements are required.

***(b) Operational Feasibility*:** Operational feasibility assesses the extent to which the required software performs series of steps to solve business problems and user requirements. This feasibility is dependent on human resources (software development team) and involves visualizing whether or not the software will operate after it is developed and be operated once it is installed. In addition, operational feasibility performs the tasks listed below:

- Determines whether the problems proposed in user requirements are of high priority or not.
- Determines whether the solution suggested by software development team is acceptable or not.
- Analyzes whether users will adapt to new software or not.
- Determines whether the organization is satisfied by the alternative solutions proposed by software development team or not.

(c) ***Economic Feasibility*****:** Economic feasibility determines whether the required software is capable of generating financial gains for an organization or not. It involves the cost incurred on software development team, estimated cost of hardware and software, cost of performing feasibility study, and so on. For this, it is essential to consider expenses made on purchases (such as hardware purchase) and activities required to carry out software development. In addition, it is necessary to consider the benefits that can be achieved by developing the software. Software is said to be economically feasible if it focuses on the issues listed below:

- Cost incurred on software development produces long-term gains for an organization.
- Cost required to conduct full software investigation (such as requirements elicitation and requirements analysis).
- Cost of hardware, software, development team, and training.

Note: *As economic feasibility assesses cost and benefits of the software, cost-benefit analysis is performed for it. Economic feasibility uses several methods to perform cost-benefit analysis, such as payback analysis, return on investment (ROI), and present value analysis.*

1.3.2 Feasibility Study Process

Feasibility study comprises of various steps, which are listed below:

1. Information assessment: Identifies information about whether the system helps in achieving the objectives of the organization. In addition it verifies that the system can be implemented using new technology and within the budget. It also verifies whether the system can be integrated with the existing system.

2. Information collection: Specifies the sources from where information about software can be obtained. Generally, these sources include users (who will operate the software), organization (where the software will be used), and software development team (who understands user requirements and knows how to fulfill them in software).

3. Report writing: Uses a *feasibility report*, which is the conclusion of the feasibility by the software development team. It includes the recommendation of whether the software development should continue or not. This report may also include information about changes in software scope, budget, schedule, and suggestion of any requirements in the system.

1.4 REQUIREMENTS ELICITATION

Requirements elicitation (also known as **requirements capture** or **requirements acquisition**) is a process of collecting information about software requirements from different individuals, such as users and other stakeholders. Stakeholders are individuals who are affected by the system, directly or indirectly. These include project managers, marketing people, consultants, software engineers, maintenance engineers, and user.

Various issues may arise during requirements elicitation and may cause difficulty in understanding the software requirements. Some of the problems are listed below:

- **Problems of scope:** This problem arises when the boundary of software (that is, scope) is not defined properly. Due to this, it becomes difficult to identify objectives as well as functions and features to be accomplished in software.
- **Problems of understanding:** This problem arises when users are not certain about their requirements and thus are unable to express what they require in software and which requirements are feasible. This problem also arises when users have no or little knowledge of the problem domain and are unable to understand the limitations of computing environment of software.
- **Problems of volatility:** This problem arises when requirements change over time.

Requirements elicitation uses *elicitation techniques*, which facilitate a software engineer to understand user requirements and software requirements needed to develop the proposed software.

1.4.1 Elicitation Techniques

Various elicitation techniques are used to identify the problem, determine its solution, and identify different approaches for the solution. These techniques also help the stakeholders to clearly express their requirements by stating all the important information. The commonly followed elicitation techniques are listed below:

- **Interviews:** These are conventional ways for eliciting requirements, which help software engineer, users, and software development team to understand the problem and suggest solution for the problem. For this, the software engineer interviews the users with a series of questions. When an interview is conducted, rules are established for users and other stakeholders. In addition, an agenda is prepared before conducting interviews, which includes the important points (related to software) to be discussed among users and other stakeholders.
- **Scenarios:** These are descriptions of a sequence of events, which help to determine possible future outcome before the software is developed or implemented. Scenarios are used to *test* whether the software will accomplish user requirements or not. Also, scenarios help to provide a framework for questions to software engineer about users' tasks. These questions are asked from users about future conditions (what-if) and procedure (how) in which they think tasks can be completed. Generally, a scenario comprises of the information listed below:
  - Description of what users expect when scenario starts.
  - Description of how to handle the situation when software is not operating correctly.
  - Description of the state of software when scenario ends.

- **Questionnaire**: questionnaires are one of the methods of gathering requirements in less cost and reach a large number of people, not only in less time but also in a lesser cost. But the results extracted from the questionnaire should be clearly analyzed. The results from the questionnaire mainly depend on two factors. 1. Effectiveness and the design of the questionnaire 2. Honesty of the respondent
- **Observation**: it is a method of collecting requirements by observing the people during their normal work.
- **Ethnography:** this method is also called participant observation where the observer is completely immersed in the work and gathers requirements while participating in the real work environment.
- **Brainstorming:** this is conducted as a conference around a table with six to ten members that are from different departments and domain experts are included. The conference is headed by the organizer, who states the issue to be discussed. Every member is allotted with a certain time interval to explain their ideas. At last the team will decide the best idea by voting.
- **Joint Application Development:** this is conducted in the same manner as brainstorming, except that stakeholders and the users are also allowed to participate and discuss on the proposed system. The participant in these sessions does not exceed 20 to30.
- **Prototyping:** Prototype is the representations or visualizations of the actual system parts. It provides the general idea of the actual system functions and the work flow and used to gather requirements from users by presenting GUI based system functions. A prototype represents the actual product in both functional and graphical sense. It provides the flexibility to the users and stakeholders to work with the initial version of the product to understand the system and discuss them to think of the additional and missed requirements. It is most expensive than all other methods of elicitation.

1.5 REQUIREMENTS ANALYSIS

IEEE defines requirements analysis as "(1) *the process of studying user needs to arrive at a definition of a system, hardware, or software requirements*. (2) *the process of studying and refining system, hardware, or software requirements*". Requirements analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements. Various other tasks performed using requirements analysis are listed below:

- Detect and resolve conflicts that arise due to unclear and unstated requirements.
- Determine operational characteristics of software and how it interacts with its environment.
- Understand the problem for which software is to be developed.
- Develop analysis model to analyze the requirements in the software.

Software engineers perform analysis modelling and create analysis model to provide information of 'what' software should do instead of 'how' to fulfill the requirements in software. This model emphasizes on information, such as the functions that software should perform, behaviour it should exhibit, and constraints that are applied on the software. This model also determines the relationship of one component with other components. The clear and complete requirements specified in analysis model help software development team to develop software according to those requirements. An analysis model is created to help the development team to assess the quality of software when it is developed. An analysis model helps to define a set of requirements that can be validated when the software is developed.

Let us consider an example of constructing a study room, where user knows the dimensions of the room, the location of doors and windows, and the available wall space. Before constructing the study room, user provides information about flooring, wallpaper, and so on to the constructor. This information helps the constructor to analyze the requirements and prepare an analysis model that describes the requirements. This model also describes what needs to be done to accomplish those requirements. Similarly, an analysis model created for software facilitates software development team to understand what is required in software and then develop it.

The guidelines followed while creating an analysis model are listed below:

- The model should concentrate on requirements that are present within the problem with detailed information about the problem domain. However, an analysis model should not describe the procedure to accomplish requirements in the system.
- Every element of analysis model should help in understanding the software requirements.
- This model should also describe the information domain, function and behaviour of the system.
- The analysis model should be useful to all stakeholders because every stakeholder uses this model in their own manner. For example, business stakeholders use this model to validate requirements whereas software designers view this model as a basis for design.
- The analysis model should be as simple as possible. For this, additional diagrams that depict no new or unnecessary information should be avoided. Also, abbreviations and acronyms should be used instead of complete notations.
- The choice of representation is made according to requirements to avoid inconsistencies and ambiguity.

Due to this, analysis model comprises of *structured analysis*, *object-oriented modelling*, and *other approaches*. Each of these describes a different manner to represent the functional and behavioral information. Structured analysis expresses this information through data flow diagrams, whereas object oriented modelling specifies the functional and behavioral information using objects.

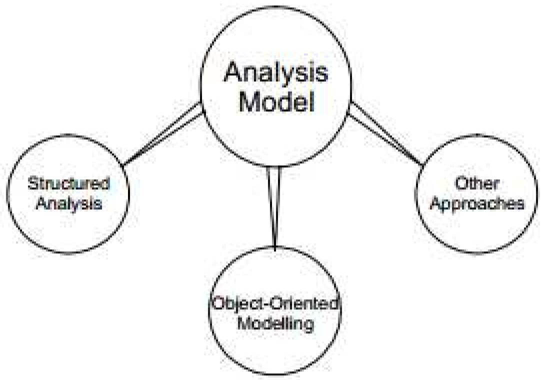


Figure 1.5 Analysis Model

Other approaches include ER modelling and several requirements specification languages and processors.

1.5.1 Structured Analysis

Structured analysis is a top-down approach, which focuses on refining the problem with the help of functions performed in the problem domain and data produced by these functions. The basic principles of this approach are:

- To facilitate software engineer in order to determine the information received during analysis and to organize the information to avoid complexity of the problem.
- To provide a graphical representation to develop new software or enhance the existing software.

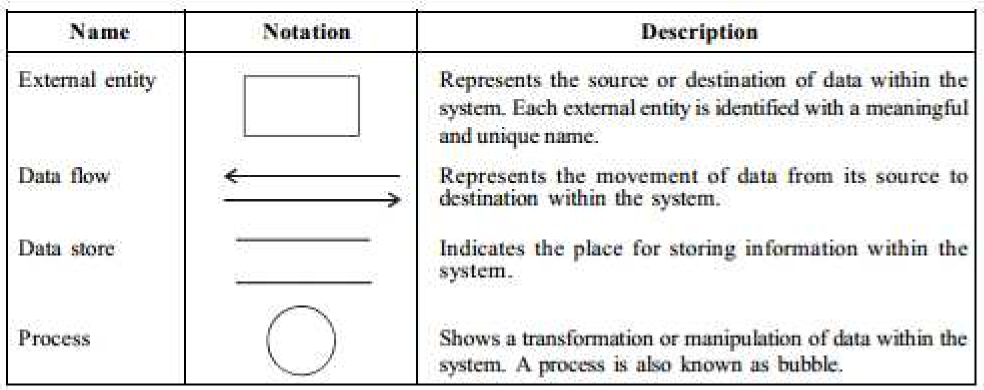
Generally, structured analysis is represented using *data flow diagram*.

(*a*) *Data Flow Diagram (DFD)*: IEEE defines data flow diagram (also known as **bubble chart** or **work flow diagram**) as "*a diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes*". DFD allows software development team to depict flow of data from one or more processes to another. In addition, DFD accomplishes the objectives listed below:

•Represents system data in hierarchical manner and with required level of detail.
• Depicts processes according to defined user requirements and software scope.

DFD depicts the flow of data within a system and considers a system that transforms inputs into the required outputs. When there is complexity in a system, data needs to be transformed by using various steps to produce an output. These steps are required to refine the information. The objective of DFD is to provide an overview of the transformations that occur to the input data within the system in order to produce an output. *DFD should not be confused with a flowchart. A DFD represents the flow of data* whereas flowchart depicts the flow of *control*. Also, a DFD does not depict the information about the procedure to be used for accomplishing the task. Hence, while making DFD, procedural details about the processes should not be shown. DFD helps a software designer to describe the transformations taking place in the path of data from input to output DFD comprises of four basic notations (symbols), which help to depict information in a system. These notations are listed below.

Table 1.2 DFD Notations



Name	Notation	Description
External entity	▭	Represents the source or destination of data within the system. Each external entity is identified with a meaningful and unique name.
Data flow	← →	Represents the movement of data from its source to destination within the system.
Data store	═	Indicates the place for storing information within the system.
Process	○	Shows a transformation or manipulation of data within the system. A process is also known as bubble.

While creating a DFD, certain guidelines are followed to depict the data flow of system requirements effectively. These guidelines help to create DFD in an understandable manner. The commonly followed guidelines for creating DFD are listed below:

- DFD notations should be given meaningful names. For example, verb should be used for naming a process whereas nouns should be used for naming external entity, data store, and data flow.
- Abbreviations should be avoided in DFD notations.
- Each process should be numbered uniquely but the numbering should be consistent.
- DFD should be created in an organized manner so that it is easily understandable.
- Unnecessary notations should be avoided in DFD in order to avoid complexity.
- DFD should be logically consistent. For this, processes without any input or output and any input without output should be avoided.
- There should be no loops in DFD.
- DFD should be refined until each process performs a simple function so that it can be easily represented as a program component.
- DFD should be organized in a series of levels so that each level provides more detail than the previous level.
- The name of a process should be carried to the next level of DFD.
- Each DFD should not have more than six processes and related data stores.
- The data store should be depicted at the context level where it first describes an interface between two or more processes. Then, the data store should be depicted again in the next level of DFD that describes the related processes.

There are various levels of DFD, which provide detail about the input, processes, and output of a system. Note that the level of detail of process increases with increase in level(s). However, these levels do not describe the systems' internal structure or behaviour. These levels are listed below:

- **Level 0 DFD** (also known as **context diagram**): Show an overall view of the system.
- **Level 1 DFD**: Elaborates level 0 DFD and splits the process into a detailed form.
- **Level 2 DFD**: Elaborates level 1 DFD and displays the process(s) in a more detailed form.
- **Level 3 DFD**: Elaborates level 2 DFD and displays the process(s) in a detailed form.

To understand various levels of DFD, let us consider an example of banking system. In Figure 1.6, level 0 DFD is drawn, this DFD represents how 'user' entity interacts with 'banking system' process and avails its services. The level 0 DFD depicts the entire banking system as a single process. There are various tasks performed in a bank, such as transaction processing, pass book entry, registration, demand draft creation, and online help. The data flow indicates that these tasks are performed by both the user and bank. Once the user performs transaction, the bank verifies whether the user is registered in the bank or not.

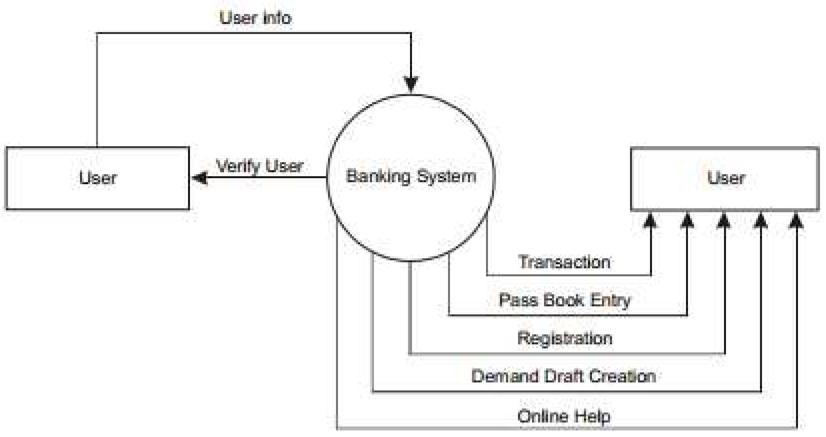


Figure 1.6 Level 0 DFD of Banking System

The level 0 DFD is expanded in level 1 DFD (Figure 1.7). In this DFD, ‘user’ entity is related to several processes in the bank, which include ‘register’, ‘user support’, and ‘provide cash’. Transaction can be performed if user is already registered in the bank. Once the user is registered, user can perform transaction by the processes, namely, ‘deposit cheque’, ‘deposit cash’, and ‘withdraw cash’. Note that the line in the process symbol indicates the level of process and contains a unique identifier in the form of a number. If user is performing transaction to deposit cheque, the user needs to provide cheque to the bank. The user’s information, such as name, address, and account number is stored in ‘user_detail’ data store, which is a database. If cash is to be deposited and withdrawn, then, the information about the deposited cash is stored in ‘cash_detail’ data store. User can get demand draft created by providing cash to the bank. It is not necessary for the user to be registered in that bank to have demand draft. The details of amount of cash and date are stored in ‘DD_detail’ data store. Once the demand draft is prepared, its receipt is provided to the user. The ‘user support’ process helps users by providing answers to their queries related to the services available in the bank.

Level 1 DFD can be further refined into level 2 DFD for any process of banking system that has detailed tasks to perform. For instance, level 2 DFD can be prepared to deposit cheque, deposit cash, withdraw cash, provide user support, and to create demand draft. However, it is important to maintain the continuity of information between the previous levels (level 0 and level 1) and level 2 DFD. As mentioned earlier, the DFD is refined until each process performs a simple function, which is easy to implement.
Let us consider the ‘withdraw cash’ process (as shown in Figure 1.7) to illustrate level 2 DFD. The information collected from level 1 DFD acts as an *input* to level 2 DFD. Not that ‘withdraw cash’ process is numbered as ‘3’ in Figure 1.7 and contains further processes, which are numbered as ‘3.1’, ‘3.2’, ‘3.3’, and ‘3.4’ in Figure 1.8. These numbers represent the sublevels of ‘withdraw cash’ process. To withdraw cash, bank checks the status of balance in user’s account (as shown by ‘check account status’ process) and then allots token (shown as ‘allot token’ process). After the user withdraws cash, the balance in user’s account is updated in the ‘user_detail’ data store and statement is provided to the user.

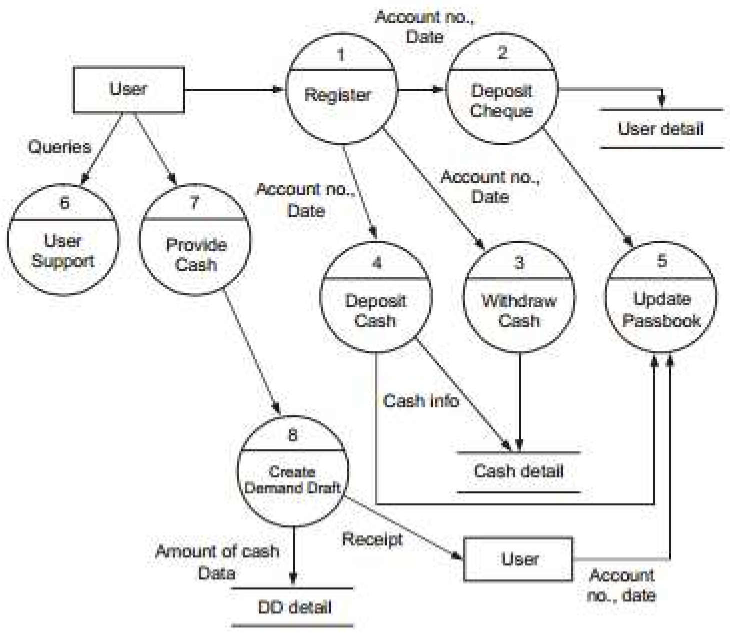
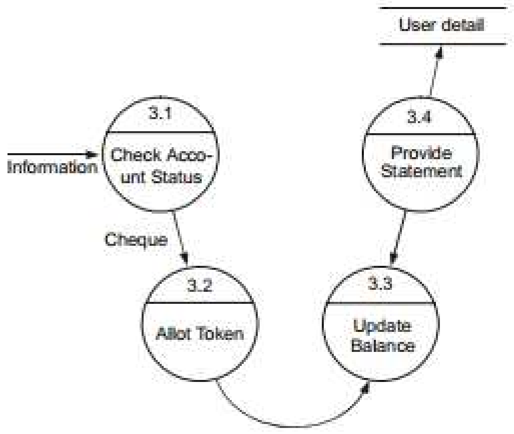


Figure 1.7 Level 1DFD to Perform Transaction



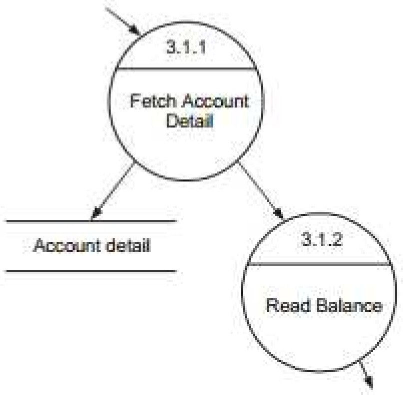


Figure 1.9 Level 3 DFD to Check Account Status

Figure 1.8 Level 2 DFD to Withdraw Cash

If a particular process of level 2 DFD requires elaboration, then this level is further refined into level 3 DFD. Let us consider the process ‘check account status’ (see Figure 1.8) to illustrate level 3 DFD. In Figure 1.9, this process contains further processes numbered as ‘3.1.1’ and ‘3.1.2’, which describe the sublevels of ‘check account status’ process. To check the account status, the bank fetches the account detail (shown as ‘fetch account detail’ process) from the ‘account_detail’ data store. After fetching the details, the balance is read (shown as ‘read balance’ process) from the user’s account. Note that the requirements engineering process of DFDs continues until each process performs a function that can be easily implemented as an individual program component.

(***b***) ***Data Dictionary***: Although data flow diagrams contain meaningful names of notations, they do not provide complete information about the structure of data flows. For this, data dictionary is used, which is a *repository* that stores description of data objects to be used by the software. Data dictionary stores an organized collection of information about data and their relationships, data flows, data types, data stores, processes, and so on. In addition, a data dictionary helps users to understand the data types and processes defined along with their uses. It also facilitates the validation of data by avoiding duplication of entries and provides online access to definitions to the users. Data dictionary comprises of the source of data, which are data objects and entities. In addition, it comprises of the elements listed below:

- **Name**: Provides information about the primary name of the data store, external entity, and data flow.
- **Alias**: Describes different names of data objects and entities used.
- **Where-used/how-used**: Lists all the processes that use data objects and entities and how they are used in the system. For this, it describes the inputs to the process, output from the process, and the data store.
- **Content description**: Provides information about the content with the help of data dictionary notations (such as '=', '+', and '* *').
- **Supplementary information**: Provides information about data types, values used in variables, and limitations of these values.

1.5.2 Object-oriented Modelling (Object Oriented Analysis OOA)

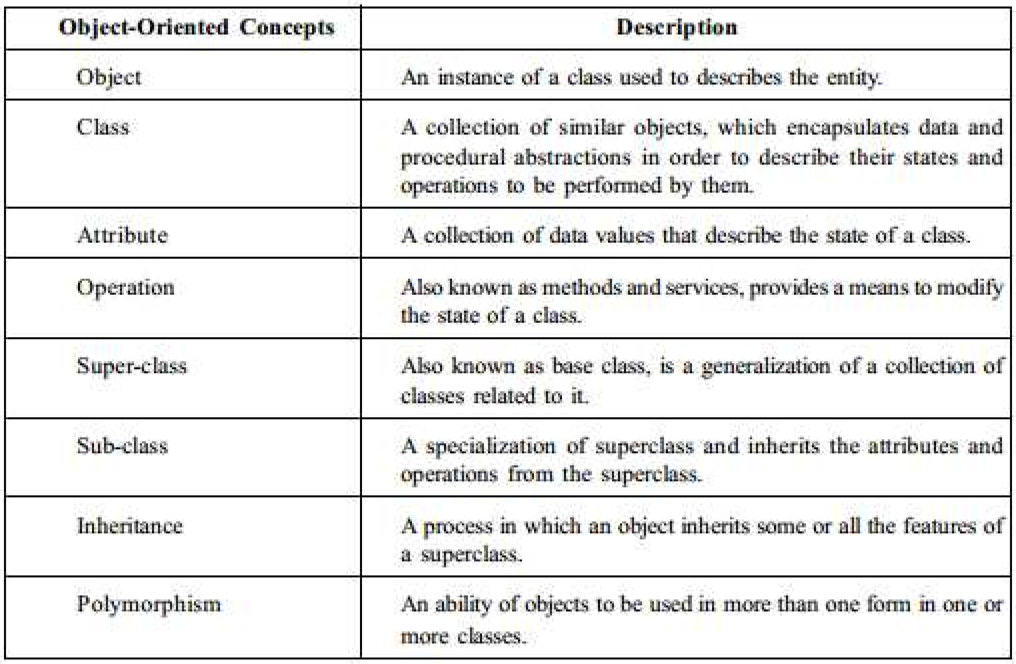
Nowadays object-oriented approach is used to describe system requirements using prototypes. This approach is performed using object-oriented modelling (also known as **object-oriented analysis**), which analyzes the problem domain and then partitions the problem with the help of objects.

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects. The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, *"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain"*.

An object is an entity that represents a concept and performs a well-defined task in the problem domain. For this, an object contains information of the state and provides services to entities, which are outside the object(s). The state of an object changes when it provides services to other entities. The object-oriented modelling defines a system as a set of objects, which interact with each other by the services they provide. In addition, objects interact with users through their services so that they can avail the required services in the system. To understand object oriented analysis, it is important to understand the various concepts used in object-oriented environment. Some of the commonly used these concepts are listed in Table 1.3.

Table 1.3 Object Oriented Concepts



Object-Oriented Concepts	Description
Object	An instance of a class used to describes the entity.
Class	A collection of similar objects, which encapsulates data and procedural abstractions in order to describe their states and operations to be performed by them.
Attribute	A collection of data values that describe the state of a class.
Operation	Also known as methods and services, provides a means to modify the state of a class.
Super-class	Also known as base class, is a generalization of a collection of classes related to it.
Sub-class	A specialization of superclass and inherits the attributes and operations from the superclass.
Inheritance	A process in which an object inherits some or all the features of a superclass.
Polymorphism	An ability of objects to be used in more than one form in one or more classes.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modelling, dynamic modelling, and functional modelling.

Object Modelling

Object modelling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.
The process of object modelling can be visualized in the following steps:

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Review glossary

Dynamic Modelling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined which is the purpose of dynamic modelling. Dynamic Modelling can be defined as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world”. It is concerned with the temporal changes in the states of the objects in a system. The main concepts are:

- State, which is the situation at a particular condition during the lifetime of an object.
- Transition, a change in the state

- Event, an occurrence that triggers transitions
- Action, an uninterrupted and atomic computation that occurs due to some event, and
- Concurrency of transitions.

A state machine models the behavior of an object as it passes through a number of states in its lifetime due to some events as well as the actions occurring due to the events. A state machine is graphically represented through a state transition diagram. The process of dynamic modelling can be visualized in the following steps:

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modelling

Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model defines the function of the internal processes in the system with the aid of Data Flow Diagram (DFDs).
The process of functional modelling can be visualized in the following steps:

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

Generally, it is considered that object-oriented systems are easier to develop and maintain. Also, it is considered that the transition from object-oriented analysis to object-oriented design can be done easily. This is because object-oriented analysis is resilient to changes as objects are more stable than functions that are used in structured analysis. Note that object oriented analysis comprises a number of steps, which includes *identifying objects*, *identifying structures, identifying attributes, identifying associations,* and *defining services.*

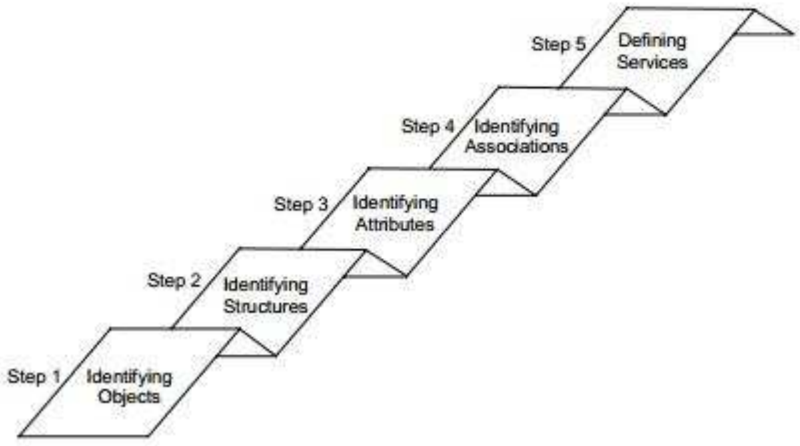


Figure 1.10 Steps in Object-oriented Analysis

(a) ***Identifying Objects*****:** While performing analysis, an object encapsulates the attributes on which it provides the services. Note that an object represents entities in a problem domain. The identification of the objects starts by viewing the problem space and its description. Then, a summary of the problem space is gathered to consider the 'nouns'. Nouns indicate the entities used in problem space and which will further be modelled as objects. Some examples of nouns that can be modelled as objects are structures, events, roles, and locations.

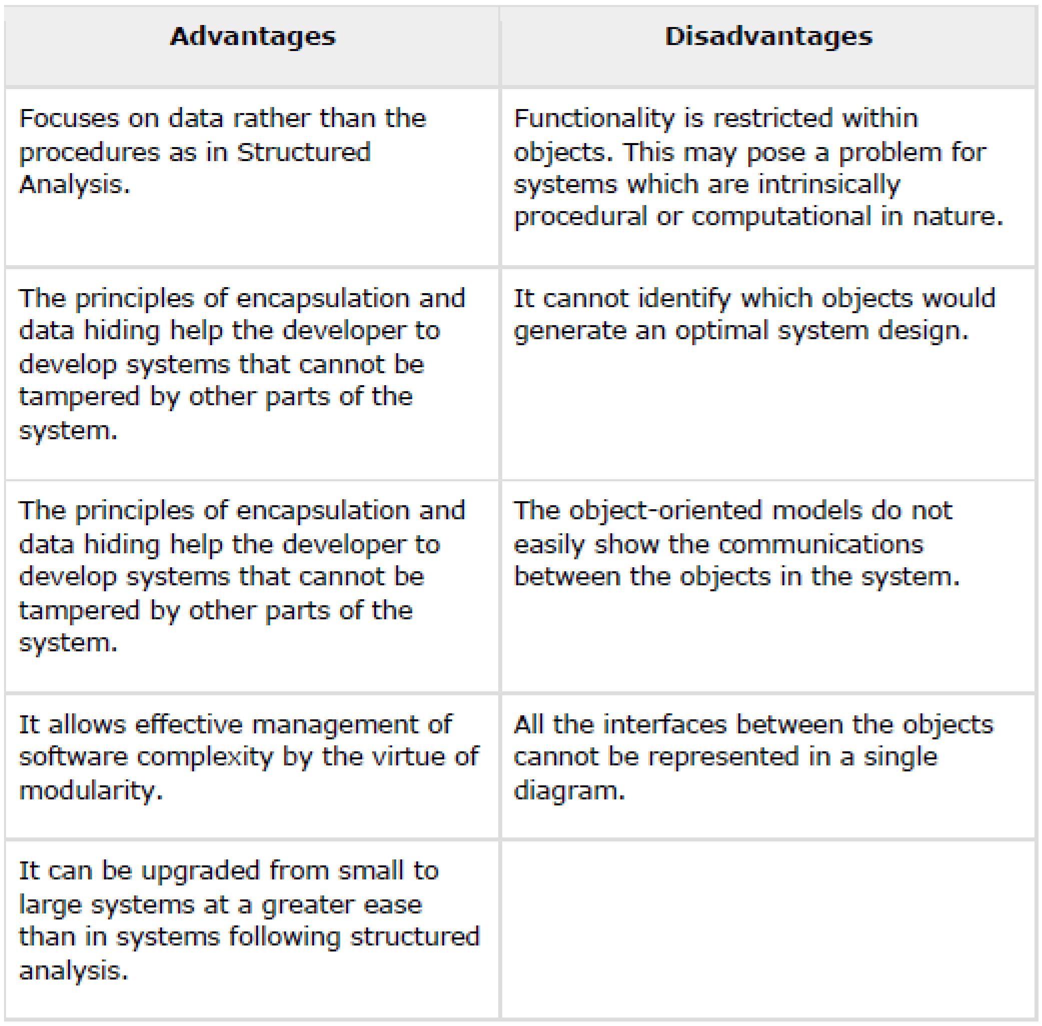
(b) ***Identifying Structures*****:** Structures depict the hierarchies that exist between the objects. Object modelling applies the concept of generalization and specialization to define hierarchies and to represent the relationships between the objects. As mentioned earlier, superclass is a collection of classes, which can further be refined into one or more subclasses. Note that a subclass can have its own attributes and services apart from the attributes and services inherited from its superclass. To understand generalization and specialization, consider an example of class 'car'. Here, 'car' is a superclass, which has attributes, such as wheels, doors, and windows. There may be one or more subclasses of a superclass. For instance, superclass 'car' has subclasses 'Mercedes' and 'Toyota', which have the inherited attributes along with their own attributes, such as comfort, locking system, and so on. It is essential to consider the objects that can be identified as generalization so that the classification of structure can be identified. In addition, the objects in the problem domain should be determined to check whether they can be classified into specialization or not. Note that the specialization should be meaningful for the problem domain.

(c) ***Identifying Attributes*****:** Attributes add details about an object and store the data for the object. For example, the class 'book' has attributes, such as author name, ISBN, and publication house. The data about these attributes is stored in the form of values and are hidden from outside the objects. However, these attributes are accessed and manipulated by the service functions used for that object. The attributes to be considered about an object depend on the problem and the requirement for that attribute. For example, while modelling the student admission system, attributes, such as age and qualification are required for the object ‘student’. On the other hand, while modelling for hospital management system, the attribute ‘qualification’ is unnecessary and requires other attributes of class ‘student’, such as gender, height, and weight. In short, it can be said that while using an object, only the attributes that are relevant and required by the problem domain should be considered.

(d) ***Identifying Associations*****:** Associations describe the relationship between the instances of several classes. For example, an instance of class ‘University’ is related to an instance of class ‘person’ by ‘educates’ relationship. Note that there is no relationship between the class ‘University’ and class ‘person’, however only the instance(s) of class ‘person’ (that is, student) is related to class ‘University’. This is similar to entity relationship modelling, where one instance can be related by 1:1, 1: M, and M: M relationships. An association may have its own attributes, which may or may not be present in other objects. Depending on the requirement, the attributes of the association can be ‘forced’ to belong to one or more objects without losing the information. However, this should not be done unless the attribute itself belongs to that object.

(e) ***Defining Services*****:** As mentioned earlier, an object performs some services. These services are carried out when an object receives a message for it. Services are a medium to change the state of an object or carry out a process. They describe the tasks and processes provided by a system To identify the services, the system states are defined and then the external events and the required responses are described. For this, the services provided by objects should be considered.

Advantages/ Disadvantages of Object Oriented Analysis



Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

Advantages/Disadvantages of Structured Analysis

Advantages	**Disadvantages**
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.
The specifications in it are written in simple English language, and hence can be more	It does not support reusability of code. So, the time and cost of development is

easily analyzed by non-technical personnel.	inherently high.

1.5.3 Other Approaches

Many other approaches have been proposed for requirements analysis and specification. These approaches help to arrange information and provide an automated analysis of requirements specification of the software. In addition, these approaches are used for organizing and specifying a requirement. The specification language used for modelling can be either graphical (depicting requirements using diagrams) or textual (depicting requirements in text form). Generally, approaches used for analysis and specification include *structured analysis and design technique* and *entity relationship modelling.*

(a) Structured Analysis and Design Technique: Structured analysis and design technique (SADT) uses a graphical notation and is generally applied in information processing systems. The SADT language is known as **language of structured analysis** (SA). SADT comprises of two parts, namely, structured analysis and design technique (DT). SA describes the requirements with the help of diagrams, whereas DT specifies how to interpret the results. The model of SADT consists of an organized collection of SA diagrams. These diagrams facilitate a software engineer to identify the requirements in a structured manner by following top-down approach and decomposing system activities, data, and their relationships. The text embedded in these diagrams is written in natural language, thus specification language is a combination of both graphical language and natural language. The commonly used SA diagrams include activity diagram (actigram) and data diagram (datagram). These diagrams use input, output, control, and mechanism for providing a reference in an SA diagram. For this, both activity diagram and data diagram comprise of nodes and arcs. Note that each diagram must consist of 3 to 6 nodes including the interconnecting arcs. These diagrams are similar to data flow diagram as they follow top-down approach but differ from DFD as they may use loops, which are not used in it.

In Figure 1.11, an activity diagram is shown with nodes and arcs. The nodes represent the activities and arcs describe the data flow between the activities. Four different types of arcs can be connected to each node, namely, *input data, control data, processor,* and *output data*.

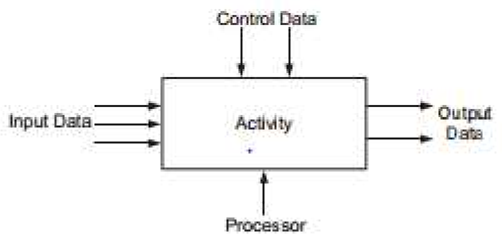


Figure 1.11 Activity Diagram

Input data is the data that are transformed to output(s). **Control data** is the data that constrain the kind or extent of process being described. **Processor** describes the mechanism, which is in the form of tools and techniques to perform the transformation.

Output data is the result produced after sending input, performing control activity, and mechanism in a system. The arcs on the left side of a node indicate inputs and the arcs on the right-side indicate outputs. The arcs entering from the top of a node describe the control, whereas the arcs entering from the bottom describe the mechanism. The data flows are represented with the help of inputs and outputs while the processors represent the mechanism.

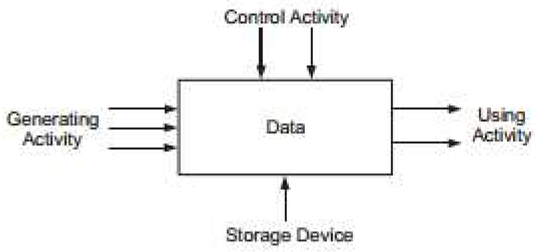


Figure 1.12 Data Diagram

In Figure 1.12, a data diagram is shown with nodes and arcs, which are similar to that of an activity diagram. The nodes describe the data objects and the arcs describe the activities. A data diagram also uses four different types of arcs. The arcs on the left side indicate inputs and the arcs on the right side indicate the output. Here, input is the activity that creates a data object, whereas output is the activity that uses the data object.

The 'control activity' (arcs entering from top) controls the conditions in which the node is activated and the 'storage device' (arcs entering from bottom) indicates the mechanism for storing several representations of a data object. Note that in both the diagrams, controls are provided by the external environment and by the outputs from other nodes.

Structured analysis and design technique provides a notation and a set of techniques, which facilitate to understand and record the complex requirements clearly and concisely. The top-down approach used in SADT helps to decompose high-level nodes into subordinate diagrams and to differentiate between the input, output, control, and mechanism for each node. In addition, this technique provides actigrams and datagrams and the management techniques to develop and review an SADT model. Note that SADT can be applied to all types of systems and is not confined only to software applications.

***(b) Entity Relationship Modelling*:** IEEE defines entity relationship (ER) diagram as "*a diagram that depicts a set of real-world entities and the logical relationships among them*". This diagram depicts entities, the relationships between them, and the attributes pictorially in order to provide a high-level description of conceptual data models. ER diagram is used in different phases of software development. Once an ER diagram is created, the information represented by it is stored in the database. Note that the information depicted in an ER diagram is independent of the type of database and can later be used to create database of any kind, such as relational database, network database, or hierarchical database. ER diagram comprises of *data objects and entities*, *data attributes*, *relationships*, and *cardinality* and *modality.*

- **Data Objects and Entities:** Data object is a representation of composite information used by software. Composite information refers to different features or attributes of a data object and this object can be in any of the form listed below:
- **External entity:** Describes the data that produces or accepts information. For example, a report.
- **Occurrence:** Describes an action of a process. For example, a telephone call.
- **Event**: Describes a happening that occurs at a specific place or time. For example, an alarm.
- **Role:** Describes the actions or activities assigned to an individual or object. For example, a systems analyst.
- **Place:** Describes location of objects or storage area. For example, a wardrobe.
- **Structure:** Describes the arrangement and composition of objects. For example, a file.

An entity is the data that stores information about the system in a database. Examples of an entity include real world objects, transactions, and persons.

Data Attributes Data attributes describe the properties of a data object. Attributes that identify entities are known as **key attributes**. On the other hand, attributes that describe an entity are known as **non-key attributes**. Generally, a data attribute is used to perform the functions listed below:

- Naming an instance (occurrence) of data object.
- Description of the instance.
- Making reference to another instance in another table.

Data attributes help to identify and classify an occurrence of entity or a relationship. These attributes represent the information required to develop software and there can be several attributes for a single entity. For example, attributes of ‘account’ entity are ‘number’, ‘balance’, and so on. Similarly, attributes of ‘user’ entity are ‘name’, ‘address’, and ‘age’. However, it is important to consider the maximum attributes during requirements elicitation because with more attributes, it is easier for software development team to develop software. In case, some of the data attributes are not applicable, they can be discarded at later stage.

Relationships: Entities are linked to each other in different ways. This link or connection of data objects or entities with each other is known as **relationship**. Note that there should be at least two entities to establish relationship between them. Once the entities are identified, software development team checks whether relationship exists between them or not. Each relationship has a name, optionality (the state when relationship can be possible but not necessary), and degree (how many). These attributes confirm the validity of a given relationship. Based on this, three types of relationships exist among entities. These relationships are listed below:

- **One-to-one relationship** (1:1): Indicates that one instance of an entity is related only to another instance of another entity. For example, in a database of users in a bank, each user is related to only one account number.

- **One-to-many relationship** (1:M): Indicates that one instance of an entity is related to several instances of an entity. For example, one user can have many accounts in different banks.
- **Many-to-many relationship** (M:M): Indicates that many instances of entities are related to several instances of another entity. For example, many users can have their accounts in many banks.

To understand *entities*, *data attributes*, and *relationship*, let us consider an example. Suppose in a computerized banking system, one of the processes is to use saving account, which includes two entities, namely, 'user' and 'account'. Each 'user' has a unique 'account number', which makes it easy for the bank to refer to a particular registered user. On the other hand, account entity is used to deposit cash and cheque and to withdraw cash from the saving account. Depending upon the type and nature of transactions, it can be of various types, such as current account, saving account, or over draft account. The relationship between user and account can be described as 'user has account in a bank'. In Figure 1.13, entities are represented by rectangles, attributes are represented by ellipses, and relationships are represented by diamond symbols. A key attribute is also depicted by an ellipse but with a line below it. This line below the text in the ellipse indicates the uniqueness of each entity.

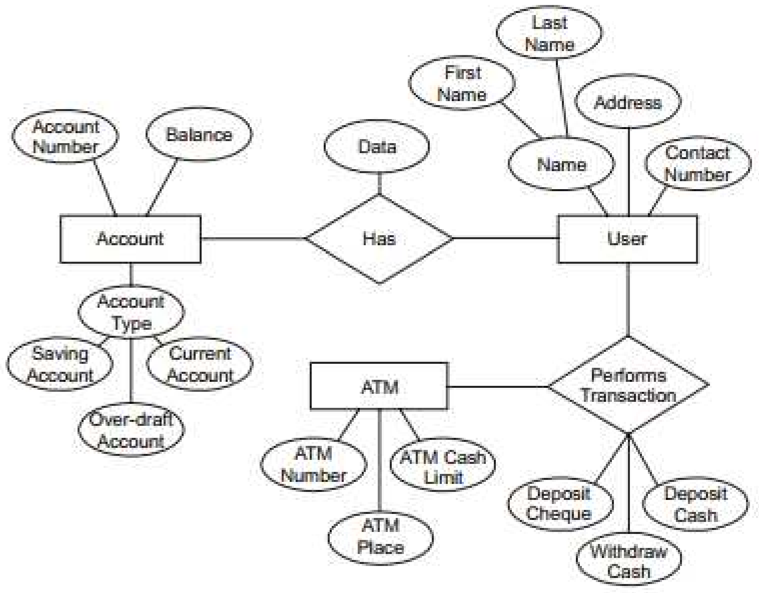


Figure 1.13 ER Diagram of Banking System

Cardinality and Modality: Although data objects, data attributes, and relationships are essential for structured analysis, additional information about them is required to understand the information domain of the problem. This information includes cardinality and modality.

Cardinality specifies the number of occurrences (instances) of one data object or entity that relates to the number of occurrence of another data object or entity. It also specifies the number of entities that are included in a relationship.

Modality describes the possibility whether a relationship between two or more entities and data objects is required or not. The modality of a relationship is 0 if the relationship is optional. However, the modality is 1 if an occurrence of the relationship is essential.

To understand the concept of cardinality and modality properly, let us consider an example. In Figure 1.14, user entity is related to order entity. Here, cardinality for 'user' entity indicates that user places an order, whereas modality for 'user' entity indicates that it is necessary for a user to place an order. Cardinality for 'order' indicates that a single user can place many orders, whereas modality for 'order' entity indicates that a user can arrive without any 'order'.

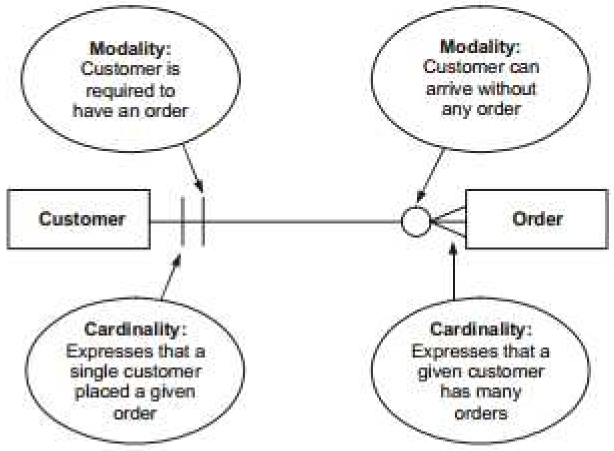

Modality: Customer is required to have an order
Modality: Customer can arrive without any order
Customer
Order
Cardinality: Expresses that a single customer placed a given order
Cardinality: Expresses that a given customer has many orders

Figure 1.14 Cardinality and Modality