System Analysis and Design

The Icfai University

System Analysis and Design

The Icfai University Press

Ref. No. SA&D 042009PG171INTRL – 110320045UG071INTRL-TB – 11109200 4UG05622**

For any clarification regarding this book, the students may please write to the Icfai University Press giving the above reference number of this book specifying chapter and page number.

While every possible care has been taken in type-setting and printing this book, the Icfai University Press welcomes suggestions from students for improvement in future editions.

Contents

Detailed Curriculum

Overview of IS Development: Concept of a System – Categories of Information Systems – Users of Information Systems – Role of a System Analyst – System Development Approaches – Structured Analysis Development Method – Systems Prototype Method – Tools for Systems Development – Structured Approach – CASE Tools.

Requirements Analysis: Stakeholder – Software Requirements Analysis – Requirements Determination – Fact-Finding Techniques – Joint Application Design – Structured Walkthrough – Analyzing and Documenting Requirements – Tools for Documenting Procedures and Decisions – Structured Analysis – Data Flow Diagram – Data Dictionary – Entity-Relationship Diagrams – Software Requirements Specification.

System Design: Design Objectives – Designing an Information System – Design Specifications – System Flowcharts – Structured Flowcharts – Database Design – File Organization – Design of Computer Output – Design of Input – User Interface Design – Designing Interfaces and Dialogues – Coupling and Cohesion.

System Testing and Implementation: System Verification – System Validation – Software Testing – Installing a System – Training and Training Methods – Conversions – Post-Implementation Review – System Audit.

Object-Oriented System Development Life Cycle: Software Development Process – Object-Oriented Systems Development – Object-Oriented Analysis – Object-Oriented Design – Prototyping – Component-Based Development – Incremental Testing – Reusability – Object-Oriented Methodologies – Unified Approach – Modelling based on the Unified Modelling Language.

UML Models: Static and Dynamic Models – Unified Modeling Language – The Meta-Model – Use-Case Diagrams – Activity Diagrams – Interaction Diagrams – Sequence Diagrams – Collaboration Diagrams – Class Diagrams – Object Diagrams – State Chart Diagrams – Implementation Diagrams – Component Diagrams – Deployment Diagrams.

Object-Oriented Analysis: Use Case Model – Developing Effective Documentation – Approaches for Identifying Classes – Identifying Attributes and Methods – Defining Attributes by Analyzing Use Cases and Other UML Diagrams.

Object-Oriented Design: Object-Oriented Design Process – Designing Classes: Refining Attributes – Designing Methods and Protocols – Object Storage and Persistence – User Interface Design.

Chapter I

Overview of IS Development

After reading this chapter, you will be conversant with:

- Concept of a System
- Categories of Information Systems
- Users of Information Systems
- Role of a System Analyst
- System Development Approaches
- Structured Analysis Development Method
- Systems Prototype Method
- Tools for Systems Development
- Structured Approach
- CASE Tools

Systems development has two major components: systems analysis and systems design. Systems design is the process of planning a new business system or replacing an existing system. Systems analysis undertaken by system analysts, is the process of gathering and interpreting facts, diagnosing problems and using information to recommend improvements to the system.

For example, consider the stockroom operations of a cloth store. In order to control its inventory and gain access to more up-to-date information about stock levels and reordering, the stores manager takes the help of a system analyst to computerize its stockroom operations. Before the system analyst can design a system to capture data, update files and produce reports, he needs to know the current state of operations: the various forms that are being used to store information manually, such as requisitions, purchase orders, invoices, and the types of reports, if any, that are being prepared.

Then, information about lists or reorder notices, outstanding purchase orders, records of stock on hand and other reports are examined. One needs to find out the source of this information, that is, whether the information is originating in the purchasing department, stockroom or the accounting department. This means one must understand how the existing system works and, more specifically, how the flow of information through the system is structured. It must be known as to why the store wants to change its current operations. Does the business have problems tracking orders, merchandise, or money? Does it seem to fall behind in handling inventory records? Does it need a more efficient system before it can expand operations?

Only after these facts have been collected, one can determine the way a computer information system can benefit all the users of the system. This accumulation of information, called systems study, must precede all other activities.

Systems analysts not only solve current problems but also handle the planned expansion of a business. Analysts assess the future needs of the business and the changes that should be considered to meet these needs.

Working with managers and employees in the organization, systems analysts recommend which alternative to adopt, based on such factors as the suitability of the solution to the particular organizational setting. They also make the employees familiar with the system so as to gain their support for the solution. The time required to develop an alternative may be the most critical issue. Costs and benefits are also determinants. Finally, management takes a decision regarding the selection of the best possible alternative.

Once this decision is made, a plan is developed to implement the recommendation. The plan includes all systems design features such as capture of particular type of data, file specifications, operating procedures, and equipment and personnel needs. Systems design is like the blueprint for a building which specifies all the features that are to be present in the finished product.

Designs for the stockroom will provide ways to capture data about orders and sales to customers and specify the way the data will be stored, whether on paper forms or on a medium such as magnetic tape or disc. The designs will also specify the work to be performed by people and computers.

The stockroom personnel will also need information about the business. Each design describes output to be produced by the system, such as inventory reports, sales analysis, purchasing summaries and invoices. The systems analysts will actually decide the type of outputs to be produced.

Analysis also specifies the tasks to be performed by the system. Design states how to accomplish the objective. Each of the processes involves people. Managers and employees are familiar with what works and what does not, what flows smoothly and what causes problems, where change is needed and where it is not, and especially where change will be accepted and where it will not. Despite technology, it is the people who make the organizations work. Thus, communicating and dealing with people are the important parts of the system analyst's job.

1. CONCEPT OF A SYSTEM

Systems are created to solve problems. One can think of the systems approach as an organized way of dealing with a problem. In the field of information technology, the subject System Analysis and Design mainly deals with the software development activities.

System analysis and design for information systems had its origin in general systems theory. General systems theory is concerned with developing a systematic, theoretical framework upon which decisions are made. It encourages taking into account all the activities of an organization and its external environment. The idea of systems has become most practical and necessary in conceptualizing the interrelationships and integration of operations, especially when using computers. Thus, a system is a way of thinking about organizations and their problems in total perspective.

1.1 Definition of System

The term system is derived from the Greek word systema, which means "an organized relationship among functioning units or components". A system comes into existence because it is designed to achieve one or more objectives. The common definition is "a collection of components that work together to realize some objective". The objective of a system is to produce some output as a result of processing suitable inputs.

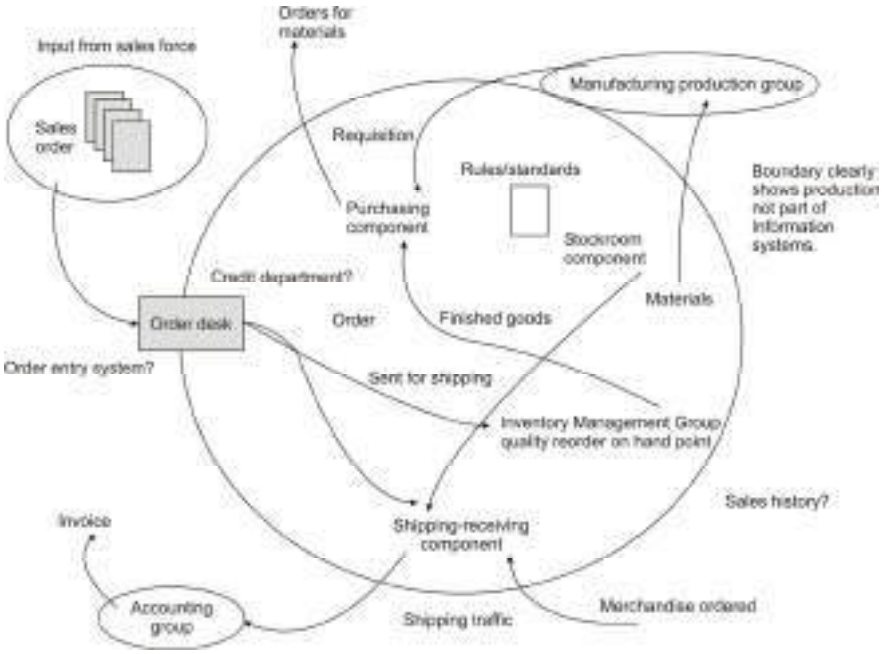
The word component may refer to physical parts (engines, wings of aircraft, wheels of a car), managerial functions (planning, organizing, directing, and controlling), or a subsystem in a multilevel structure. The components may be simple or complex, basic or advanced. The components are independent. In a system, the different components are connected with each other and have to do their share of work for the system to achieve the intended goal. This orientation requires an orderly grouping of the components for the design of a successful system.

For example, human body represents a complete natural system. A single computer with a keyboard, memory and a printer or a series of intelligent terminals linked to a mainframe is known as a computer system. In either case, each component is part of the total system.

A business is another example of a system. Its components are marketing, manufacturing, sales, research, shipping, accounting and personnel, all working together to achieve profitability that benefits the employees and stockholders of the firm. Each of these components in itself, constitutes a system. For example, the accounting department may consist of accounts payable, accounts receivable, billing, auditing, and so on.

Figure 1 illustrates the concept of a system together with its interrelated elements, applied to a business situation. The interrelationship is important to successful operation of systems.

Figure 1: System's Concept Applied to a Business Situation



The study of systems concept has three basic implications:

i. A system must be designed to achieve a predetermined objective.

ii. Interrelationships and interdependence must exist among the components.

iii. The objectives of an organization as a whole have precedence over the objectives of its subsystems.

Every business system depends to some extent on an abstract entity called an information system. Information means data that has been selected and processed in response to a question. Information systems serve all the systems of a business, linking the different components in such a way that they effectively work towards the same purpose.

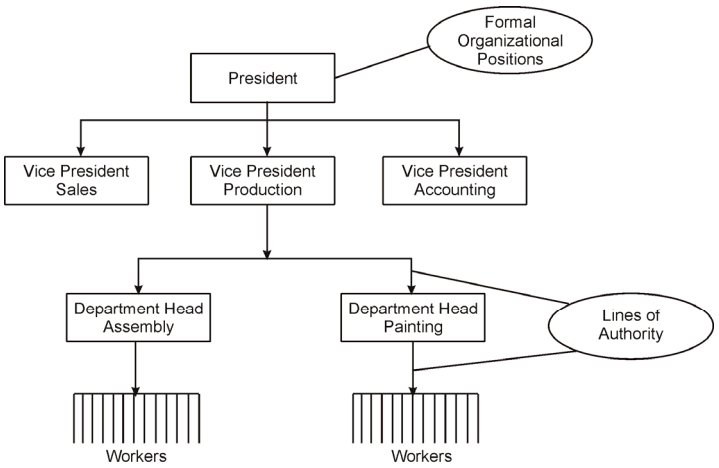
1.2 Characteristics of a System

Following are the characteristics of a system:

- Organization
- Interaction
- Interdependence
- Integration
- Central objective.

i. **Organization:** It implies the structure and order in the arrangement of the components that help to achieve objectives. For example, in the design of the business system, the hierarchical relationship starting with the president on the top and moving towards the workers at the bottom represents the organizational structure. Such an arrangement defines the authority structure, specifies the formal flow of communication and formalizes the chain of command. An example of such a structure is shown in figure 2.

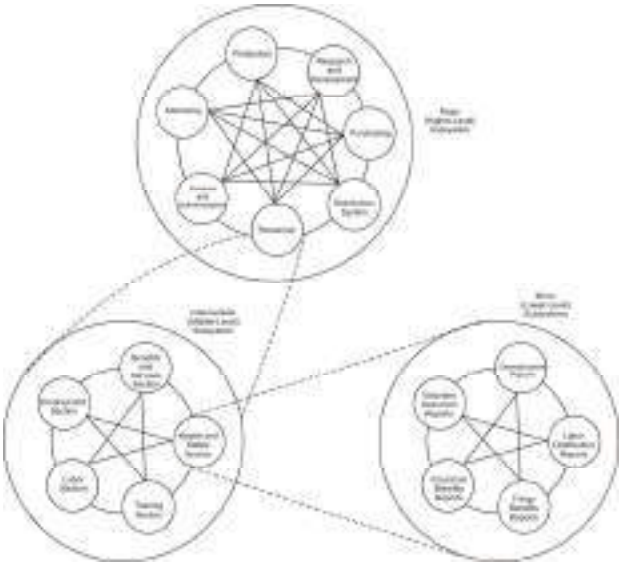
Figure 2: Example of an Organizational Structure



ii. **Interaction:** It represents the relationships among the components of the system i.e., the manner in which each component interacts with other components. For example, in a computer system, the Central Processing Unit (CPU) must interact with the input device to obtain input data. In an organization, the purchase department must interact with the production department; the advertising department must interact with the sales department and so on.

iii. **Interdependence:** Interdependence means that parts of an organization depend on one another. They are coordinated and linked according to a plan. The output of one subsystem serves as an input for another subsystem for proper functioning. Figure 3 shows three levels of subsystems. Each of the top inner circles represents a major subsystem of a production firm. The subsystems of the production firm are production, marketing, finance and administration, personnel, and so on. The personnel subsystem may be viewed as consisting of other subsystems such as labor section, training section, health and safety section and so on. Again, the health and safety subsystem consists of the lower-level elements, such as insurance benefits reports, unemployment reports, labor distribution reports and so on.

Figure 3: Levels of Subsystems



In a nutshell, no subsystem can function in isolation because it is dependent on the data it receives from other subsystems to perform its required tasks.

iv. **Integration:** Integration is concerned with the way the system is bound together i.e., parts of the system work together within the system even though each part performs a unique function. Successful integration will typically produce greater total impact than if each component works separately.

v. **Central objective:** Objectives may be real or stated. It is not uncommon for an organization to state one objective and work practically on another objective. The important point is that users must know the central objective of a computer application early in the analysis for successful design and conversion.

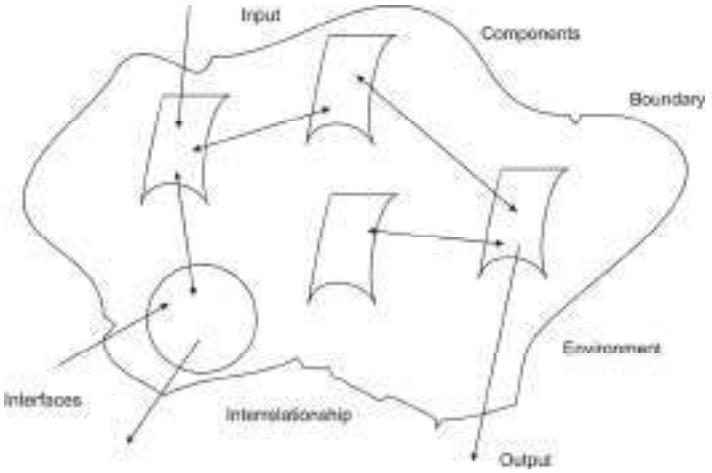
1.3 Elements of a System

In most cases, the system analyst operates in a dynamic environment where change is a way of life. The following key elements must be considered for constructing a system:

- Components
- Interrelated components
- Boundary
- Purpose
- Environment
- Interfaces
- Input
- Output
- Constraints.

Figure 4 depicts the various elements of a system. The arrows in the figure show the interaction between the system and the world outside of it known as environment.

Figure 4: Elements of System



Let us discuss about these elements in detail:

i. **Components:** A component is an irreducible part or aggregation of parts that make up a system. It is also called a subsystem. A system is made up of components. A component has all the characteristics of a system. For example, we can repair or upgrade an automobile or a stereo system by changing individual components without having to make changes to the entire system in total.

ii. **Interrelated Components:** The components are interrelated i.e., the dependence of one part of the system on one or more other parts of the system. For example, producing a daily report of customer orders received may not progress successfully until the work of another component is finished.

iii. **Boundary:** Boundary is the line that demarcates the system from its environment and that sets off one system from other systems in the organization. Components within the boundary can be changed whereas systems outside the boundary cannot be changed.

iv. **Purpose:** Purpose is the overall goal or function of a system. All the components work together to achieve some overall purpose for the larger system.

v. **Environment:** A system exists within an environment. An environment is everything that lies outside the system's boundary and interacts with the system. For example, the environment of ICFAI University includes prospective students, foundations and funding agencies, higher education department of the government, and the print and electronic media. An information system interacts with its environment by receiving and sending data and information.

vi. **Interfaces:** The points at which the system meets its environment or where the subsystems meet one another are called interfaces.

vii **Input and Output:** Inputs are the elements that enter the system for processing. Output is the outcome of the processing. A system feeds on input to produce output in much the same way that business brings in human, financial and material resources to produce goods and services. Determining the output is the first step in specifying the nature, amount and regularity of the input needed to operate a system.

vii. **Constraints:** Constraint is a limit to the extent to which a system can accomplish its goals. A system faces constraints because there are limits to its abilities and also its purpose within its environment. Some of the constraints arise within the interior of the system while others are imposed by the environment. For instance, a production system is constrained if the availability of electricity is irregular.

1.4 System Models

A model is a representation of a real or a planned system. The use of models makes it easier for the analyst to visualize relationships in the system. The objective is to point out the significant elements and the key interrelationship of a complex system. The system analyst begins by creating a model of a real system. For example, a telephone switching system is made up of subscribers, telephone handsets, dialing, conference calls, etc. The analyst begins by modeling these elements before considering the functions that the system would perform. The major system models are discussed here:

- **Schematic Models:** It is a two-dimensional chart representing the system elements and their linkages.
- **Flow System Models:** It shows the logical order among the system elements.
- **Static System Models:** It exhibits one pair of relationships such as activity-time or cost-quantity.
- **Dynamic System Models:** It represents an ongoing, constantly changing system. It consists of,
    a. Inputs that enter the system,
    b. The Processor through which transformation takes place,
    c. The programs required for processing, and
    d. Outputs that result from processing.

1.5 Types of Systems

Systems have been classified in different ways. Some common classifications are:

- Physical and Abstract.
- Open and Closed.
- Information Systems.

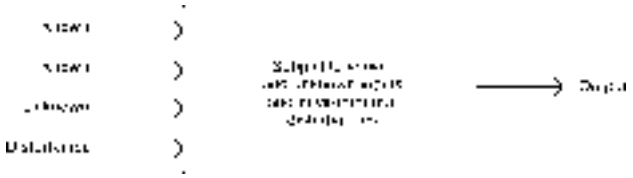
i. **Physical and Abstract Systems:** Physical systems are tangible (real) entities that may be static or dynamic in operation. For example, the physical parts of a computer center are the offices, desks and chairs that facilitate operation of the computer center. They can be seen and counted; they are static. In contrast, a programmed computer is a dynamic system. Data, programs, output, and applications change based on the user's demands or the priority of the information requested. Abstract systems are conceptual or non-physical entities. They may be as straightforward as relationships among sets of variables or models.

ii. **Open and Closed Systems:** To achieve its purpose, a system interacts with its *environment*, which includes the entities outside the boundary of the system. Systems that interact with their environments (receive input and produce output), are *open systems*. In contrast, systems that do not interact with their environments are closed systems. All systems presently operating in nature are open. Thus, closed systems exist only as a concept. In systems analysis, organizations, applications and computers are invariably open dynamic systems influenced by their environment.

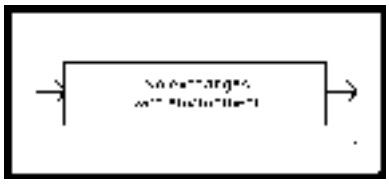
System analysis for an open system tends to expand the scope of analysis so as to include relationships between the user area and other users and also the environmental factors that must be considered before a new system is finally approved. Being open to suggestions implies that the analyst has to be flexible and the system being designed has to respond to the changing needs of the user and the environment. Figure 5 distinguishes between open and closed systems.

Figure 5: Open and Closed Systems

Figure 5(a): Open System



5(b): Closed System



Characteristics of an open system:

- **Input from Outside –** Inputs received from outside are self-regulating and self-adjusting. When functioning properly, an open system reaches a steady state or equilibrium.
- **Entropy –** All dynamic systems are depleted over time due to entropy or loss of energy. Open systems resist entropy by seeking new inputs or modifying the processes to return to a steady state.
- **Process, Output and Cycles –** Open systems produce useful output and operate in cycles following a continuous flow path.

- **Differentiation –** Open systems differentiate their components. This characteristic motivates system analysts to incorporate open system concept in their thinking.
- **Equifinality –** It implies that goals are achieved through differing courses of action through a variety of paths.

Understanding system characteristics helps analysts to identify their role and relate their activities to the attainment of the firm's objective as they undertake a system project.

iii. **Information Systems:** Computerized Information System (IS) is a collection of computer personnel, components and procedures used and designed to provide services, collect, process, and store data, and disseminate information. Computer systems that store data and supply information often rely on databases, i.e., a system that is designed to capture, transmit, store, retrieve, manipulate, and, or display information used in one or more business processes. The subject area of information systems includes an understanding of:

- Businesses/organizations – their aims, management, structure and methods of working.
- The intended use of information systems within organizations.
- The information technology used in information systems.
- The process and techniques of analyzing and designing an information system.
- The professional, legal, social and ethical issues involved in the application of information systems and information technology.
- Qualities of information.

The information that is supplied to the managers through information systems must have the following qualities:

i. Information must be **accurate** i.e., the correctness of the input data and that of the processing rules should be ensured so that the resulting information is accurate. The information should be **complete** i.e., it should include all data that is needed.

ii. It should also be **trustworthy**. The processing should not hide some vital information which may, for example, point out the inefficiency of some individuals.

iii. Information should be **timely**. It should be given to the manager when he needs it. Delayed information may sometimes be of no value. For example, if a daily newspaper is delivered a day later, it would lose its importance.

iv. Information should also be **up-to-date**. It should include all data available at the time of processing. A newspaper delivered in time early in the morning but reporting old news is timely but not up-to-date.

v. Information should be tailored to the needs of the user and be **relevant**. Massive volumes of irrelevant information would waste a lot of manager's time and there is a danger of bypassing important relevant information.

vi. It is essential to give brief **summarized** information to ensure quick action.

vii. The information should be presented in real time and at the place it is needed, in such a way that its significance is immediately perceived. For example, presentation of information in a graphical form such as bar chart or pie chart, ensures quick recognition of the significance of the information. It is also essential to present the information in an attractive format which a user can immediately understand.

The information system present in an organization can be seen as an intermediary between the business and the information technology infrastructure of the organization. Examples of commonly used information systems are:

- **Payroll:** This system starts with details of employees and their rates of pay, and processes this data to produce bank transfers, pay slips, etc. The payroll system also provides information on, for example, the payroll cost of staff in various departments and grades within the organization.
- **Office Automation Systems:** Office Automation Systems (OAS) try to improve the productivity of employees by automating data and information processing. Perhaps the best example is the wide range of software systems that exist to improve the productivity of employees working in an office (e.g., Microsoft Office XP) or systems that allow employees to work from home or during their travel.
- **Order Processing:** The main input transaction is the customer order which is processed using customer and product data to output the delivery note and invoice transactions. In addition to processing the business transactions, the system can produce a wealth of management information on what is being, sold, the profile of customers and the overall sales totals for each month and year.

Examples of organizations having their own information systems are:

i. A supermarket with its sales and stock replenishment system.

ii. A manufacturing company with its Materials Requirement Planning (MRP) and production control systems.

iii. A college or university with its student registration and records system.

In addition to these central systems on which the operations of the organizations depend, there will be a number of other systems for functions such as marketing, accounts, customer came and so on. Some of these applications will be **formal information systems** and some will make use of standard desk top packages.

2. CATEGORIES OF INFORMATION SYSTEMS

Most businesses require different types of information. Senior managers need information to help in their business planning. Middle management needs more detailed information to help them monitor and control business activities. Employees with operational roles need information to help them carry out their duties. As a result, businesses tend to have several types of information systems operating at the same time. System analysts develop different types of information systems to meet a variety of business needs.

2.1 Transaction Processing Systems

As the name implies, Transaction Processing Systems (TPS) are designed to process routine transactions efficiently and accurately. Transaction is an elementary activity conducted during business operations such as merchandise sale, airline reservation, credit-card purchase, or inquiry about inventory. Transaction Processing Systems are process oriented.

Transaction Processing Systems (TPS), also referred to as **Online Transaction Processing (OLTP)** systems, capture data, store it reliably and securely in a database, and retrieve it when requested.

A business will have several types of TPS. Some examples are:

Billing systems to send invoices to customers.

Systems to calculate the weekly and monthly payroll and tax payments.

Production and purchasing systems to calculate raw material requirements.

Stock control systems to process all movements into, within and out of the business.

The common characteristics of different types of transactions are:

There is a high volume of transactions.

Each transaction is similar.

The procedures for processing the transactions are well-understood and can be described in detail.

Few exceptions to the normal procedures occur.

These characteristics allow routines to be established for handling the transactions. The routines describe the substance in each transaction, the steps to be taken, the procedures to be followed and the action to be taken when exceptions occur. Transaction processing procedures are often called **standard operating procedures**.

Consider the example of an Automated Teller Machine (ATM) system that allows the teller to use a computer terminal to enter details of the transaction while the customer is at the bank window. The procedures are built into the computer software that runs the system. Similarly, when customers make withdrawals at automated teller machines, the software used to operate the system ensures that the following procedure is followed:

Customer Activity	System Activity
Enters account number	Verifies that the account number is correct.
Enters password	Verifies that the password is valid for the account.
Enters withdrawal amount	i. Verifies that the amount is within limits set by the bank. ii. Verifies that the account has balance. iii. Records transaction in ledger. iv. Dispenses money. v. Issues receipt for transaction.
Removes receipt and money	Prepares for next transaction.

This activity is repeated many times in a single day at most ATMs.

Transaction processing systems provide speed and accuracy, and can be programmed to follow routines without any variance. System analysts design the systems and the processes to handle activities such as bank transactions, ticket bookings etc.

2.2 Management Information Systems

Management Information System (MIS) is mainly concerned with the organization's internal sources of information. Management Information Systems usually take data from the transaction processing systems and summarize it into a series of management reports. MIS reports tend to be used by middle management and operational supervisors. Transaction systems are operations-oriented, whereas Management Information Systems are data-oriented. MIS assists managers in decision-making and problem-solving.

A key element of MIS is the database. In any organization, decisions must be made on many issues that persist regularly (weekly, monthly, quarterly, etc.) and require a certain set of information to make the decision. Because the decision process is well understood, the information that will be needed to formulate decisions can be identified. In turn, the information system can be developed so that reports are prepared regularly to support these recurring decisions.

The decisions supported by MIS are structured decisions. This means managers know what factors to consider in making the decision and which variables most significantly influence the outcome of the decisions. System analysts develop well-structured reports containing the information that is needed for the decisions

or that tells the state of the important variables. The primary users of MIS are middle and top management, operational managers and support staff. Once entered into the system, the information is in the public domain of all authorized users.

A management information system, or management reporting system, will feature reports based on the transaction level activities. For instance, regular reports on deposits and withdrawals in total and by branch office are routinely used by bank officers to keep informed on the performance of individual branches, to monitor the ratio of loans made to deposits received, the level of cash reserves, interest paid to depositors and other common performance indicators.

The information is often combined with other external information such as details about economic trends, demand for loans, rate of consumer spending and cost of borrowing. Bank managers can make informed decisions about the rate of interest they will charge the following week for various types of loans or about whether they must raise the interest rates they pay customers to attract more deposits. The need to make each of these decisions recurs frequently and the information needed to formulate the decisions is also prepared regularly.

Most of the MIS reports are historical and tend to be outdated. Many installations have databases that are not in line with user requirements. An inadequate or incomplete update of the database raises the reliability issues for all the users.

2.3 Decision Support Systems

Decision Support Systems (DSSs) are specifically designed to help managements make decisions in situations where there is uncertainty about the possible outcomes of those decisions. A decision is considered unstructured if there are no clear procedures for making the decision and if not all the factors to be considered in the decision can be readily identified in advance. DSSs comprise tools and techniques to help gather relevant information and analyze the options and alternatives. DSSs are used in data warehouses and Executive Information Systems (EIS).

A key factor in the use of decision support systems is determining what information is needed. In well-structured situations it is possible to identify information needs in advance, but in an unstructured environment it is difficult to do so. As information is acquired, the manager may realize that additional information is required, that is, having information may lead to the realization of other requirements.

Consider the decision process followed by banking officers who must decide whether to begin offering cash management accounts or installing automatic teller machines – both completely new banking services. The many questions that will arise are: What will each service cost? How many teller locations will be needed? How will the competitors respond to this? What limits should be placed on withdrawls at any point of time? Can a charge be imposed for this service? Will this service result in additional deposits and thus more cash inflow for the bank?

In such cases, it is impossible to pre-design system report formats and contents. A decision support system must therefore have greater flexibility than other information systems. The user must be able to request reports by defining their content and even by specifying how the information is to be produced. Similarly, the data needed to develop the information may originate from many different files or databases, rather than from a single master file, as is often the case with transaction systems and many reporting systems.

Subjective managerial judgment plays a vital role in decision-making where the problem is not structured. The decision support system supports, but does not replace, managerial judgment.

Information systems are expressly designed to support individual and collective decision-making by making it possible to apply decision models to large collections of data. These systems are designed to support the decision-making process rather than render a decision.

2.4 Expert Systems

Expert Systems are man-machine systems with specialized problem-solving expertise. The "expertise" consists of knowledge about a particular domain, understanding of problems within that domain, and "expertise" at solving some of these problems. Present day expert systems deal with the domains of narrow specialization.

The primary goal of research in expert systems is to make expertise available to decision makers and technicians who need answers quickly. Enough expertise is not always available at the right place and the right time. Portable computers loaded with in-depth knowledge of specific subjects can bring and apply decade's worth of knowledge as a solution to a problem. The same systems can assist supervisors and managers with situation assessment and long-range planning. Many small systems now exist that bring a narrow slice of in-depth knowledge to a specific problem and provide evidence that the broader goal is achievable.

These knowledge-based applications of Artificial Intelligence (AI) have enhanced productivity in business, science, engineering and military. With advances in the last decade, today's expert systems clients can choose from dozens of commercial software packages with easy-to-use interfaces. Each new deployment of an expert system yields valuable data thus fueling the AI research that provides even better applications.

2.5 Scope of Information Systems

The scope of information systems includes:

- Effective utilization of information technology in organizational context.
- Interdependencies of information technologies and organizational structure, relationships and interaction.
- Evaluation and management of information systems.
- Analysis, design, construction, modification and implementation of computer-based information systems for organizations.
- Management of knowledge, information, and data in organizations.
- Information systems applications in organizations such as transaction processing, routine data processing, decision support, office support, computer-integrated manufacturing, expert support, executive support and support for strategic advantage plus the coordination and interaction of such applications.
- Relevant research and practice from associated fields such as computer science, operations management, economics, organizational theory, cognitive science, knowledge engineering and systems theory.

3. USERS OF INFORMATION SYSTEMS

The degree of involvement of the managers and employees in an organization interacting with information systems is different and it depends on the type of the user. The end-users refer to people who are not professional information systems specialists but who use computers to perform their jobs. There are four types of end-users:

- **Hands-on End-users:** They actually interact with the system by feeding input data and receiving output. For example, clerical staff at computerised airline reservation counters use terminals to query the system about passenger, flight, and ticket information.
- **Indirect End-users:** They benefit from the results or reports produced by these systems but do not directly interact with the hardware or software. These users may be functional managers of business functions using the system. For example, marketing managers make use of sales analysis applications that result in monthly reports.

- **User Managers:** They manage application systems like managers. These users may be upper-level managers for business functions that make use of information systems extensively. While user managers may not actually use the systems directly or indirectly, they retain authority to approve or disapprove investment in the development of applications and have organizational responsibility for the effectiveness of the systems (in the same way that a vice president of marketing is responsible for the success of all sales and marketing programs). These upper-level users must be involved in major systems development efforts.
- **Senior Managers:** They take increased responsibility for the development of information systems. The best-managed organizations consider the possible impact and benefit of information systems when formulating their organization's competitive strategy.

All four types of end-users are important. Each has essential information about how the organization functions and where it is going. System analysts are often the ones who supply the ideas – the imagination – about ways to use computers effectively. The analysts collect information about a business system that forms the basis for the design of a new system or for modifying an existing one.

The characteristics of different end-users are different. Some may have never used a computer, others are intermittent users, and still others may interact daily with information system. Each group must be able to use the information system easily and in a timely manner when required, even though its use may not be part of their daily routine. At the same time, the features of the systems required to meet the needs of the infrequent user (such as the capability to receive extra assistance) should not impede the frequent users. Analysts strive to balance systems features to suit the needs of all potential users.

The end-user can also be a competitor, not an employee of the firm. For example, some information systems are used by airline travel agents or corporate purchasing agents who have terminals linked to a suppliers' computer. Additional considerations must be built into the system for this type of end-user interaction with the system and to make sure that only the necessary information reaches the end-user while there is no breach of organization's information systems.

Users are becoming highly involved in systems development for several reasons. They are:

- Users have accumulated experience from working with applications that were developed for them earlier. They have better insight about the information content generated by these systems in a specific format and also the manner in which the desired information can be obtained. If they have experienced systems failures, they have also formed ideas about avoiding problems.
- Microcomputers, in the form of workstations, personal computers, or home computers, and software that meet the user needs, whether for business requirements or personal management necessities, have become common.
- Users entering business organizations have college or university training in various aspects of information systems, often in systems analysis and design.
- The applications being developed in organizations are becoming complex. System analysts need the continual involvement of users to understand the business functions.
- Better systems development tools are emerging. Some allow users to design and develop applications without involving trained system analysts.

4. ROLE OF A SYSTEM ANALYST

A System Analyst is a person responsible for studying the requirements, feasibility, cost, design, specification and implementation of a computer-based system for an organization/business.

Systems analysts solve computer problems and enable computer technology to meet individual needs of an organization. They help an organization realize the maximum benefit from its investment in equipment, personnel and business processes. This process may include planning and developing new computer systems or devising ways to apply existing systems' resources to additional operations. System analysts may design new systems, including both hardware and software, or add a new software application to harness more of the computer's power. Most systems analysts work with a specific type of system that varies with the type of organization they work for – for example, business, accounting or financial systems, or scientific and engineering systems.

Analysts begin an assignment by discussing the problem with managers and users to determine its exact nature. They define the goals of the system and divide the solutions into individual steps and separate procedures. Analysts use techniques such as structured analysis, data modeling, information engineering, mathematical model building, sampling and cost accounting to plan the system. They specify the inputs to be accessed by the system, design the processing steps and format the output to meet the users' needs. They also may prepare cost-benefit and Return-on-Investment (ROI) analyses to help management decide whether implementing the proposed system will be financially feasible.

When a system is accepted, analysts determine the type of computer hardware and software that will be needed to set it up. They test and determine the initial use of the system to ensure it performs as planned. They prepare specifications, work diagrams and structure charts for computer programmers to follow and then work with them to "debug" or eliminate errors from the system. Analysts performing in-depth testing of products may be referred to as software quality assurance analysts. In addition to running tests, these individuals diagnose problems, recommend solutions and determine if program requirements have been met.

5. SYSTEM DEVELOPMENT APPROACHES

Computer information systems serve many different purposes. The factors to be considered in an information systems project are the most appropriate aspect of the computer or communications technology to be applied, the impact of a new system on the people in a firm, and the special features the system should have. These aspects can be determined in a sequential fashion. The experience must be gained through experimentation and staged evolution of a system.

Systems development as a process starts when the management or systems development personnel realize that a particular business system needs improvement. We can represent three distinct approaches to the development of computer information systems:

i. Systems Development Life Cycle Method (SDLC).

ii. Structured Analysis Development Method.

iii. Systems Prototype Method.

We shall discuss each approach, focusing on the characteristics and the conditions under which it is likely to have the highest value to the organization.

5.1 Systems Development Life Cycle

Systems Development Life Cycle (SDLC) is a logical process by which system analysts, software engineers, programmers and end-users build information systems and computer applications to solve business problems and needs. First, it is necessary to determine what the problem is (analysis), then figure out a good approach for solving it (design), and finally put in practice the approach (implementation). It is sometimes called as application development life cycle.

Systems development life cycle means combination of various activities. In other words, we can say that various activities put together are referred to as system development life cycle. System Development Life Cycle (SDLC) is a conceptual model used in project management that describes the stages involved in an information system development project, from an initial feasibility study through maintenance of the completed application.

SDLC comes in two basic types:

- The **waterfall** or linear approach implies that one step follows another in a sequence. Previously, older systems were developed using the waterfall model. A major problem is, it assumes that all the analysis can be done without doing any design or implementation. This is not possible for a complex system.
- The fountain or **iterative** approach implies that you do some analysis, then some design, and then some implementation. Based on what you learn, you cycle back through and do more analysis. This supports human learning a lot better. This approach is followed for projects in present times.

In general, the SDLC methodology follows the following steps:

- The existing system is evaluated to identify deficiencies. This can be done by interviewing users of the system and consulting with support personnel.
- The new system requirements are defined. In particular, the deficiencies in the existing system must be addressed with specific proposals for improvement.
- The proposed system is designed. Plans are laid out concerning the physical construction, hardware, operating systems, programming, communications, and security issues.
- The new system is developed. The new components and programs must be obtained and installed. Users of the system must be trained in its use, and all aspects of performance must be tested. If necessary, adjustments must be made at this stage.
- The system is put into use. This can be done in various ways. The new system can be gradually installed according to application or location, and the old system gradually replaced. In some cases, it may be more cost-effective to shut down the old system and implement the new system all at once.
- Once the new system is up and running for a while, it should be exhaustively evaluated. Maintenance must be kept up rigorously at all times. Users of the system should be kept up-to-date concerning the latest modifications and procedures.

Systems Development Life Cycle Method consists of the below given six activities:

i. Preliminary investigation.

ii. Determination of system requirements.

iii. Designing of system.

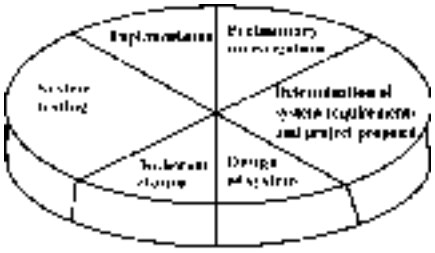
iv. Development of software.

v. Systems testing.

vi. Implementation and evaluation.

The six phases in the system development life cycle can be identified by different names. Also, there are no definite rules regarding what must be included in each of the six phases. The different phase of software development life cycle are shown in figure 6. Let us now describe the different phases and the related activities of systems development life cycle in detail.

Figure 6: Activities in Systems Development Life Cycle



5.1.1 PRELIMINARY INVESTIGATION

Preliminary investigation is the first phase of SDLC. Feasible system requests are evaluated in terms of cost and benefits. The purpose of Preliminary Investigation is not to develop a system, but to gather enough information to determine if the next phase is warranted. This phase is typically very short, usually not more than a day or two for a big project, and in some instances, it can be as little as two hours. This activity has three parts:

i. **Request Clarification:** Requests from employees and users in organizations are not clearly stated. The requests should have the following characteristics:
   - Easy to understand.
   - Include clear instructions.
   - Should have all the supporting documents.
   - Spacious – enough space to enter the data.
   - Make it available on-line (if possible and feasible).

ii. **Feasibility Study:** Feasibility study is basically the test of the proposed system in light of its workability, meeting user's requirements, effective use of resources and of course, the cost effectiveness. The main goal of feasibility study is not to solve the problem but to determine the extent to which it is possible to construct the system. In the process of feasibility study, the cost and benefits are estimated with greater accuracy. It has three types:
   - **Operational Feasibility:** This feasibility addresses questions like: Does the proposed system align with the company's goals? What will be the impact of the System on the employees? Is there enough support/enthusiasm in the company for the proposed System?
   - **Technical Feasibility:** Does the company have enough technical resources, both human/non-human, to run the proposed system? If not, can the company acquire them?
   - **Economic Feasibility**: Will the proposed system increase revenues either directly or indirectly?

   The above study must be conducted before the System Request can be approved/rejected.

   Example: A software engineer working in a software development center comes up with the following proposal: "Buy 10 water filters/purifiers costing Rs.3500 each, and start delivering pure water to homes".

   Looking at this system request, it is evident that this project will have an upfront cost of Rs.35,000 – hence economically feasible.

   Filtering water and then delivering it to homes is a straight forward task, and anyone can do it – hence technically feasible.

Even though supplying pure drinking water is technically feasible without much complexity, the software company should not undertake this project because this does not align with the company's goals, and there will be no support for it from the employees. Therefore, the request should be denied based on its operational unfeasibility. People typically responsible for feasibility assessments are experienced analysts or managers.

iii. **Request Approval:** Some organizations receive so many project requests from employees that only a few of them can be pursued. Those projects that are both feasible and desirable should be put into a schedule. After a project request is approved, its cost, priority, completion time and personnel requirements are estimated and used to determine where to add it to any existing project list.

5.1.2 DETERMINATION OF SYSTEM REQUIREMENTS AND PROJECT PROPOSAL

In this we study the problem, deficiency or new requirement in detail. **Requirement** is a statement of what the system or part of the system should do. Depending upon the size of the project being undertaken, this phase could be as short as the preliminary investigation, or it could take months.

Analysis involves a detailed study of the current system, leading to specifications of a new system. Analysis is a detailed study of various operations performed by a system and their relationships within and outside the system. During analysis, data are collected on the available files, decision points and transactions handled by the present system. Interviews, on-site observations and questionnaires are the tools used for system analysis. Based on the guidelines given below, it becomes easy to draw the exact boundary of the new system under consideration:

- Considering the problems and new requirements.
- Working out the pros and cons including new areas of the system.

All procedures and requirements must be analyzed and documented in the form of detailed Data Flow Diagrams (DFDs), data dictionary, logical data structures and miniature specifications. System analysis also includes sub-dividing complex processes involving the entire system, identification of data store and manual processes.

The main points to be discussed in system analysis are:

- Specification of what the new system is to accomplish based on the user requirements.
- Functional hierarchy showing the functions to be performed by the new system and their relationship with each other.
- Functional network, which is similar to functional hierarchy but highlight the functions which are common to more than one procedure.
- List of attributes of the entities – these are the data items which need to be held about each entity (record).

The Requirements statement should list all of the major details of the program.

Project Proposal

After completing the analysis for the proposed system, the next step is the preparation of project proposal. Depending upon the size and importance of the project, the proposal may be long and detailed or relatively short and focused. The proposal that is presented in the given form and format must highlight the advantages of the project to the organization and its users in clear and precise terms without any ambiguity. It must present the facts which support the proposed project and highlight the goals that would be achieved with the completion of the project. It must also bring to light the positive changes that would happen in the organization once the new system would come into existence with the completion of the project. The proposal must instill confidence in the minds of the readers who

would have the authority to sanction the funds. For this to happen it is necessary that the proposal should emphasize that the proposed project is the best course of action keeping in mind the various risks and alternatives. The contents of the proposal include the following:

- A statement that defines the business problem being solved;
- The chosen solution, explaining why it was chosen and briefly indicating the other alternatives;
- A description of how the new system will work and its impact on external clients and internal users;
- Justification for choosing the preferred alternative and its economic, technical and operational advantages;
- The duties that different people in the organization will have to perform in order to implement the solution; and
- The impact of the proposed project on the way people work, including any new skills that people have to learn and the way of learning additional skills.

5.1.3 DESIGN OF SYSTEM

Based on user requirements and detailed analysis, the new system must be designed. The design of an information system produces the details that state how a system will meet the requirements identified during systems analysis. It is the most crucial phase in the development of a system. Normally, the design proceeds in two stages:

- Preliminary or general design.
- Structure or detailed design.

i. **Preliminary or General Design:** In the preliminary or general design, the features of the new system are specified. The costs of implementing these features and the benefits to be derived are estimated. If the project is still considered to be feasible, we move to the detailed design stage.

ii. **Structure or Detailed Design:** In the detailed design stage, computer oriented work begins in earnest. At this stage, the design of the system becomes more structured. Structure design is a blue print of a computer system solution to a given problem having the same components and inter-relationship among the same components as the original problem. Input, output and processing specifications are drawn up in detail. In the design stage, the programming language and the platform in which the new system will run are also decided.

There are several tools and techniques used for designing. These tools and techniques are:

- Flowchart
- Data Flow Diagram (DFD)
- Data dictionary
- Structured English
- Decision table
- Decision tree.

5.1.4 DEVELOPMENT OF SOFTWARE

Software developers install purchased software or they may write new, custom-designed programs. The choice depends on:

- Cost,
- Availability of time, and
- Availability of programmers.

After designing the new system, the whole system is required to be converted into a language that would be understood by a computer. This is known as coding into an appropriate computer language. It is an important stage where the defined procedures are transformed into control specifications with the help of a computer language. This is also called the **programming phase** in which the programmer converts the program specifications into computer instructions, which we refer to as programs. The programs coordinate the data movements and control the entire process in a system.

It is generally felt that the programs must be modular in structure. This helps in fast development, maintenance and future change, if required.

5.1.5 SYSTEMS TESTING

Before actually implementing the new system into operations, a test run of the system is done removing all the bugs, if any. It is an important phase of a successful system. After codifying the whole programs of the system, a test plan should be developed and run on a given set of test data. The output of the test run should match the expected results. Different test runs are carried out.

When it is ensured that the system is running error-free, the users are called with their own actual data so that the system could be shown running as per their requirements.

5.1.6 IMPLEMENTATION AND EVALUATION

After having the user acceptance of the new system developed, the implementation phase begins. Implementation is the stage of a project during which theory is turned into practice. During this phase, all the programs of the system are loaded onto the user's computer. After loading the system, training of the user starts. The training should help the user to know:

- How to execute the package
- How to enter data
- How to process data (processing details)
- How to take out reports.

After the users are trained about the computerized system, manual working has to shift to computerized working. The following two strategies are followed for running the system:

– **Parallel run:** In parallel run both the systems i.e., computerized and manual, are executed in parallel for a certain defined period. This strategy is helpful because:
  - Manual results can be compared with the results of the computerized system.
  - Failure of the computerized system at the early stage does not affect the working of the organization, because the manual system continues to work as it used to do.

– **Pilot run:** In this type of run, the new system is installed in parts. Some parts of the new system are installed first and executed successfully for considerable time period. When the results are found satisfactory, then only other parts are implemented. This strategy builds the confidence and the errors are traced easily.

When the implementation report is submitted, an evaluation should be made to determine whether the system meets the objectives stated in the general design report. In this phase, users may be able to suggest easy-to-implement improvements. The actual evaluation can occur along any of the following dimensions:

- Operational Evaluation
- Organizational Evaluation
- User Manger Assessment
- Development Performance.

As in the six phase development life cycle, the project can be dropped at any point prior to implementation. A project may be dropped if the benefits derived from the proposed system do not justify commitment of the needed resources or if the cost is higher than expected.

6. STRUCTURED ANALYSIS DEVELOPMENT METHOD

The structured analysis development methods are based on functional decomposition and stepwise refinement which also involves the breaking down of complex systems into single-function tasks and subtasks. These techniques were the first to evolve. The analysis part (data flows) tended to concentrate on the business flow (which was generally manual in the beginning). The data was simple (integers, float/real, and characters), but the processing was typically complex. The early methods were used both for MIS and real-time systems. Extensions of these methods (e.g., Structured Analysis of Real-Time Systems) gave more support for real-time systems and concurrent processing. It consists of elements of both analysis and design.

6.1 Structured Analysis

It is a set of techniques and graphic tools that allow the analyst to develop a new kind of system specification that is easily understandable to the user. Hence, it focuses on specifying what the system is required to do. It does not state how the requirements should be accomplished. It allows seeing logical elements apart from the physical components. It is useful for top down decomposition of the high level system functions. It captures the system structure as perceived by the user.

Elements of structured analysis include:

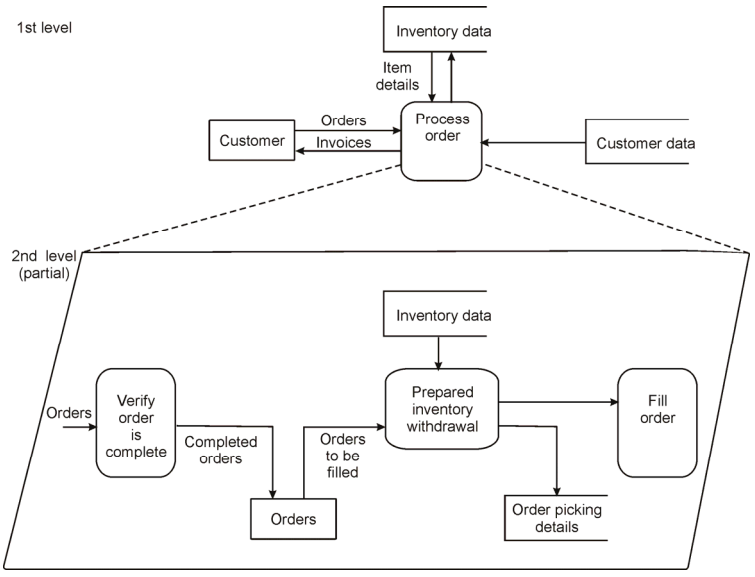
- Graphical Symbols.
- Data Flow Diagrams.
- Data Dictionary.

6.1.1 GRAPHICAL DESCRIPTION

Structured Analysis uses symbols or icons to create a graphical model of the system. Graphical models show details of the system without introducing manual or computer processes, tapes or disk files, or program and operating procedures.

As figure 7 indicates, the icons identify the basic elements of processes, data flows, data stores, data sources and destinations. A line is drawn around the system to indicate the elements included within the system and those outside the boundary.

Figure 7



6.1.2 DATA FLOW DIAGRAMS

Data flow diagrams are the most commonly used methods of documenting the process and current & required systems. As the name suggests, they are a pictorial way of showing the flow of data into, around and out of a system.

Graphical representation of a system's data and how the processes transform the data is known as Data Flow Diagram (or DFD). Unlike flowcharts, DFDs do not give detailed descriptions of modules but graphically describe a system's data and how the data interact with the system. DFDs are constructed using four major components i.e., external entries, data stores, processes and data flows.

We know that structure analysis follows top-down process. Each process can be broken down into a yet more detailed data flow diagram. This may occur repeatedly until sufficient detail is given to allow the analyst to fully understand the portion of the system under investigation.

Figure 7 shows the first and second level for a portion of a system. The focus is on data and processes.

6.1.3 DATA DICTIONARY

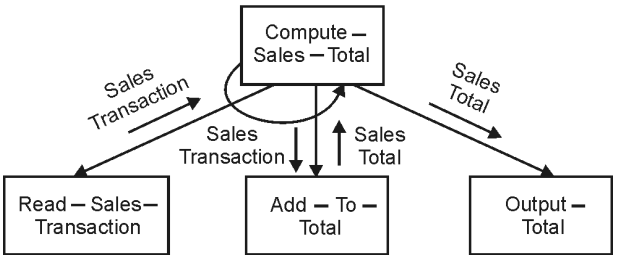
All definitions of elements in the system i.e., data flows, processes and data stores are described in detail in a data dictionary.

6.2 Structured Design

Structured design makes use of graphical description for the development of software specifications. The functions that are identified during structured analysis are mapped on to a modular structure. Top-down decomposition approach is used for successively breaking down the system functions into modules. These modules are then mapped on a structure, which, in turn, is implemented with some programming language. The goal of the structured design is to create programs consisting of independent modules that perform relatively independent of one another.

The fundamental tool of structured design is the structure chart. An example of structure chart is shown in figure 8. Structured charts are graphical representations and avoid specification of hardware or physical details. They describe the interaction between independent modules and the data passing between modules that interact with one another.

Figure 8: An Example of a Structure Chart



In figure 8, the top module is called COMPUTE-SALES-TOTAL. This module calls three lower-level program modules to accomplish its task. READ-SALES-TRANSACTION module is called to read individual sales transaction. ADD-TO-TOTAL model is called to sum the amount in each transaction and OUTPUT-TOTAL module is called to output the sum.

7. SYSTEMS PROTOTYPE METHOD

The systems prototype method involves the user more directly in the analysis and design experience than does the SDLC or Structured Analysis and Structured Design. It is useful only if it is employed at the right time and appropriate manner.

A Prototype is a working model of an actual system that is constructed in order to explore implementation or processing alternatives and evaluate results.

7.1 Reasons for Using Systems Prototype

There are two major problems with building information systems:

- SDLC is a long drawn process, and
- The right system is rarely developed the first time.

Lengthy development effort and tedious methodologies for developing systems frustrates both the users and the system analysts. The reason they often come up with the wrong system is that they expect the users to define their information requirements. It usually turns out that what they want is not what they need.

Although the prototype is a working system, it is designed to be easily changed. Information gained through its uses is applied to a modified design that may again be used as prototype to reveal more valuable design information. The process is repeated as many times as necessary to reveal design requirements.

System analysts find prototypes to be most useful under the following conditions:

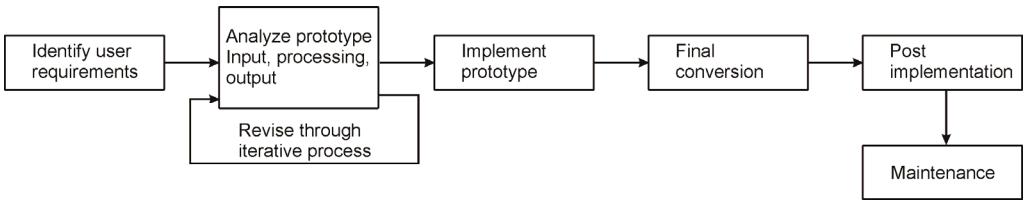
- No system with the characteristics of the one proposed has yet been constructed by the developers.
- The essential features of the system are only partially known; others are not identifiable even after careful analysis of requirements.
- Experience in using the system will significantly add to the list of requirements the system should meet.
- Alternate versions of the system will evolve through experience and additional development and refinement of its features would take place.
- The system user will participate in the development process.
- Systems prototyping is an interactive process. It may begin with only a few functions and be expanded to include others that are identified later.

The steps involved in the Prototyping process are:

- Identify the user's information and operating requirements.
- Develop a working prototype that focuses on only the most important functions using a basic database.
- Allow the user of the prototype to discuss requested changes and implement the most important changes.
- Repeat the next version of the prototype with further changes incorporated until the system fully meets the user's requirements.

The prototyping process is given in figure 9.

Figure 9: Prototyping Process



Prototyping should not be considered a trial-and-error development process. The analyst works with the users to determine the initial or basic requirements for the system. The analyst then quickly builds a prototype. When the prototype is complete, the users work with it and tell the analyst their additional requirements or changes. The analyst uses this feedback to improve the prototype and develops the new version. This iterative process continues until the users are relatively satisfied with the model.

Usually one of the following four alternatives is selected:

i. *The prototype is redeveloped:* This alternative may mean complete reprogramming from scratch.

ii. *The prototype is implemented as the completed system:* Performance efficiency and methods for user interaction may be sufficient to allow the system to be used as it is.

iii. *The prototype is abandoned:* The prototype has provided enough indication that a system cannot be developed to meet the desired objectives within the existing technology or economic or operational guidelines.

iv. *Another prototyping series has begun:* The information gained through current experience may suggest an entirely different approach.

Each alternative is viewed as a successful result of prototyping.

Advantages of Prototyping Technique are:

- It involves the user in analysis and design, and
- It captures requirements in concrete, rather than verbal or abstract form.

8. TOOLS FOR SYSTEMS DEVELOPMENT

A tool is any device that improves the performance of a task, such as the development of a computer information system. The tools can be of three types – Analysis tools, Design tools, and Development tools.

8.1 Analysis Tools

Analysis tools assist systems specialists in documenting an existing system, whether manual or automated, and determining the requirements for a new application. These tools include:

- **Data Collection Tools:** Details should be captured describing current systems and procedures. Processes and decision activities should be documented and used in requirements identification.
- **Charting Tools:** Graphic representations of systems and activities are created. Data Flow Diagrams and icons associated with structured analysis are used in drawing and revision. Flowcharting programs are also included.
- **Dictionary Tools:** Descriptions of system elements such as data items, processes and data stores are recorded and maintained. These tools also provide capabilities to examine inconsistent or incomplete system's descriptions. The capabilities to report where items are used are also included.

Most useful tools in each of these categories are becoming automated, both to improve the efficiency of the analyst and to make the results of the analysis effort more accurate and complete.

8.2 Design Tools

Design tools help in formulating the features of a system that will meet the requirements outlined during the analysis activities. These tools include:

- **Specification Tools:** These tools are used to specify the features that should be included in an application such as input, output, processing, and control specifications. Tools for creating data specifications are also included.
- **Layout Tools:** These tools are used to describe the position of data, messages, and headings on display screens, reports and other input and output media.

Analysts have used tools for the design of systems since the early days of computing. The recent infusion of computer assistance and powerful graphics is giving new meaning to systems design.

8.3 Development Tools

Development tools aid the analyst in translating designs into functioning applications. These tools include:

- **Software Engineering Tools:** These tools are used for formulating software designs, including procedures and controls, as well as documentation for the design are assisted.
- **Code Generators:** These are used to produce source code and working applications from functional specifications that are well articulated.
- **Testing Tools:** These tools are used for evaluating a system or portion of a system against specifications. Evaluations for correct operation, as well as for completeness in comparison with expectations are carried out.

The infusion of computer processing, coupled with sophisticated design practices, is dramatically altering the manner in which design specifications are translated into working information systems.

9. STRUCTURED APPROACH

A Structured approach to systems design not only provides cutting-edge data tools of structured analysis and design, but also presents traditional techniques such as interviewing and forms design. Its goal is to create an integrated methodology by combining the best elements of new and traditional technologies. The tools and techniques of analysis and design are dependent on the way they are used in business applications. Structured Approaches are SDLC, RAD, Spiral Model, etc.

9.1 Information Engineering Approach

The Information Engineering (IE) approach provides a more formal framework for planning top-down systems development, with the development flowing down a number of successive stages. These stages involve identifying the global information needs and then mapping on to these the requirements for gathering, managing, accessing and processing the information.

9.2 Object-Oriented Approach

During object-oriented analysis, the emphasis is on finding and describing the objects – or concepts – in the problem domain. Take the example of library information system, where some of the concepts include Book, Library, and Patron.

During object-oriented design, there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a software object named Book may have a title attribute and a get Chapter method.

10. CASE TOOLS

CASE stands for Computer-Aided Systems Engineering. Many definitions and descriptions of CASE exist. The general definition of CASE is:

"CASE is the use of computer-based support in the software development process." This definition includes all kinds of computer-based support for any of the managerial, administrative or technical aspects of any part of a software project.

Since the early days of writing software, there has been an awareness on the need for automated tools to help the software developer. Initially, the concentration was on program support tools such as translators, compilers, assemblers, macro processors, linkers and loaders. However, as computers became more powerful and the software that was used along with it grew larger and more complex, the range of support tools also began to expand. In particular, the use of interactive time-sharing systems for software development encouraged the development of program editors, debuggers, code analyzers and program-pretty printers.

As computers became more reliable and useful, the need for a broader notion of software development became apparent. Software development came to be viewed as:

- A large-scale activity involving significant effort to establish requirements, design an appropriate solution, implement that solution, test the solution's correctness, and document the functionality of the final system.
- A long-term process producing software that requires enhancement throughout its lifetime. The implications of this are the structure of the software must enable new functionality to be added easily, and detailed records of the requirements, design, implementation and testing of the system must be kept to aid maintainers of the software. In addition, multiple versions of all artifacts produced during a project must be maintained to facilitate group development of software systems.
- A group activity involving interaction among a number of people during each stage of the development process. Groups of people must be able to cooperate in a controlled manner and have consistent views on the state of the project.

Large-scale programming resulted in a wide range of support tools being developed. Initially, the tools were not very sophisticated in their support. However, two important advances had the effect of greatly improving the sophistication of these tools:

- Research in the area of software development processes gave rise to a number of software design methods (e.g., the Jackson Structured Programming and the Yourdon Method) that could be used as the basis for software development. These methods were ideally suited in an environment which provided the support of automated tools which required step-by-step adherence to methods, had graphical notations associated with them, and produced a large number of artifacts (e.g., diagrams, annotations, and documentation) that needed to be recorded and maintained.
- Personal workstations and personal computers have relatively large memory storage capacities, fast processors, and sophisticated bit-mapped graphic displays that are capable of displaying charts, graphical models, and diagrams.

All the above mentioned tools may be referred to as CASE tools and posit the following definition:

“A CASE tool is a computer-based product aimed at supporting one or more software engineering activities within a software development process.”

The most common distinctions between CASE tools are:

- Between those tools that are interactive in nature (such as a design method support tool) and those that are not (such as a compiler). The former classes are sometimes called CASE tools, while the latter classes are called development tools.
- Between those tools that support activities early in the life cycle of a software project (such as requirements and design support tools) and those that are used later in the life cycle (such as compilers and test support tools). The former classes are sometimes called front-end CASE tools and the latter are called back-end CASE tools.
- Between those tools that are specific to a particular life-cycle step or domain (such as a requirements tool or a coding tool) and those that are common across a number of life-cycle steps or domains (such as a documentation tool or a configuration management tool). The former classes are sometimes called vertical CASE tools, while the latter classes are called horizontal CASE tools.

However, there are certain problems associated with these distinctions. In the first case, it is difficult to give a simple and consistent definition of ‘interactive’ that is meaningful. For example, some classes of compilers prompt the user for information. In the second and third cases, there is an assumption about the methods and approaches in use (e.g., object-oriented software development, or prototype-oriented development).

10.1 Case Environment

The first generation of CASE tool developers concentrated to a large extent on the automation of isolated tasks such as document production, version control of source code, and design method support. While successes have been achieved in supporting such specific tasks, the need for these ‘islands of automation’ to be connected has been clearly recognized by many first generation CASE tool users. For example, a typical development scenario requires that designs be closely related to their resultant source code, that they be consistently described in a set of documentation, and that all of these artifacts be under centralized version control. The tools that support the individual tasks of design, coding, documentation, and version control must be integrated if they are to support this kind of scenario effectively. These tools are more often used as components in a much more elaborate software development support infrastructure that is available to software engineers. A typical CASE environment consists of a number of CASE tools operating on a common hardware and software platform. There are different classes of users of a CASE environment. Some users, such as software developers and managers, wish to make use of CASE tools to support them in developing application systems and monitoring the progress of a project. On the other hand, tool integrators are responsible for ensuring that the tools operate on the software and hardware platform available, and the system administrator’s role is to maintain and update the hardware and software platform itself.

It is to be noted that software developers, tool integrators, and system administrators interact with multiple CASE tools and environment components that form the software and hardware platform of the CASE environment. The interactions among the different components of CASE environment and also between the users and these components feature prominently in a CASE environment. In many respects, the approach towards the management, control, and support of these interactions distinguishes one CASE environment from another. We can define a CASE environment by emphasizing the importance of these interactions.

A CASE environment is a collection of CASE tools and other components together with an integration approach that supports most or all of the interactions that occur among the environment components and between the users of the environment and the environment itself.

A critical part of this definition is that the interactions among environment components are supported within the environment. What distinguishes a CASE environment from a random amalgamation of CASE tools is that there is some thing that is provided in the environment that facilitates interaction of those tools. This ‘something’ may be a physical mechanism such as a shared database or a message broadcast system, a conceptual notion such as a shared philosophy on tool architectures or common semantics about the objects the tools manipulate, or a combination of these things.

The range of possible ways of providing the 'glue' that links CASE tools together inevitably leads to a spectrum of approaches to implementing a CASE environment. One of the main points we make in this book is that there are many ways to build a CASE environment. While many people concentrate on the selection of CASE tools and components when assembling a CASE environment, they largely ignore the need to support the interactions among those components. We concentrate less on which components should be chosen and much more on how the selected components can be made to work together effectively. Whether a chosen approach to component interaction is appropriate in a given context will depend on many overlapping factors: the needs of the organization in question, the available resources, and so forth.

The use of CASE tools brings the following benefits:

- Enhancing existing applications.
- Complete, accurate and consistent performance of the system.
- Reducing human effort.
- Integrated development.
- Speed of development.
- Links to object-oriented approaches.

SUMMARY

- Systems analysis and design for businesses is the process of studying a business situation to see how it operates and whether improvement is needed – systems study is conducted to learn the details of the current business situation. Information gathered through the study forms the basis of creating alternative design strategies. The Management selects the strategy to pursue. As managers employees, and other end-users are familiar with computing, they are having a greater role in systems development.
- Systems analysis and design is the application of the systems approach to problem solving, generally using computers. To reconstruct the system, the analyst must consider its elements – inputs, outputs, components, constraints, feedback and environment. Virtually all organizations are systems that interact with their environment through receiving inputs and producing outputs. Systems, which may consist of other smaller systems called subsystems, operate to accomplish specific purposes. However, the purposes or goals are achieved only when control is maintained. In open systems that interact with their environment, performances are evaluated against standards. The results (feedback) are useful in adjusting systems activities to improve performance.
- System analysts play a key role in systems development. They maintain relationship with business users on one hand and technical personnel on the other. Analysts need to develop analytical skills, technical skills, managerial skills and interpersonal skills to succeed.
- Transaction processing systems, also referred to as Online Transaction Processing (OLTP) systems, have as their basic purpose the capturing of data, storing it reliably and securely in a data-base, and retrieving it from this database when requested.

- Management Information System (MIS) is mainly concerned with internal sources of information. MIS usually takes data from the transaction processing systems and summarizes it into a series of management reports. MIS reports tend to be used by the middle management and operational supervisors. Transaction systems are operations-oriented whereas Management Information Systems (MIS) are data oriented. It assists managers in decision-making and problem solving.
- Decision-Support Systems (DSS) are specifically designed to help the management make decisions in situations where there is uncertainty about the possible outcomes of those decisions. A decision is considered unstructured if there are no clear procedures for making the decision and if not all the factors to be considered in the decision can be readily identified in advance. Decision-Support System comprises tools and techniques to help gather relevant information and analyzes the options and alternatives. DSS often involves in data warehouses, and Executive Information Systems (EIS). Decision-support systems are data and decision logic oriented.
- Expert Systems are man-machine systems with specialized problem-solving expertise. The “expertise” consists of knowledge about a particular domain, understanding of problems within that domain, and “skill” at solving some of these problems.
- The components of information systems include hardware, software and data stores in files and databases. Information system applications are the procedures, programs, files, and equipment – all carefully integrated to accomplish specific purposes.
- There are three systems development strategies: the classical systems development life cycle method, structured analysis development method, and the systems prototype method. All the three development strategies are in widespread use in organizations of all types and sizes, and each is effective when properly used. Analysts are responsible for developing information systems that are useful to the management and employees in business systems. The systems development life cycle, includes preliminary investigation, collection of data, determination of requirements, designing of a system, development of software, systems testing, and implementation. Several of these activities may be going on concurrently, since different parts of the system may vary in their degree of completion.
- Structured Analysis is a set of techniques and graphic tools that allow the analyst to develop a new kind of system specification that is easily understandable to the user. DFDs show the flow of data into the system and between processes and data stores.
- Structured Design utilizes graphic description, and focuses on the development of software specifications. It identifies the functions during the structured analysis and mappes on to a modular structure.
- A Prototype is a working system to explore processing alternatives and evaluate results.
- Systems analysts rely on a wide variety of tools to fulfill their responsibilities. When properly applied, important tools of analysis, design, and development can each contribute substantially to the usefulness of a system.
- A CASE environment is a collection of CASE tools and other components together with an integration approach that supports most or all of the interactions that occur among the environment components and between the users of the environment and the environment itself.

Chapter II

Requirements Analysis

After reading this chapter, you will be conversant with:

- Stakeholder
- Software Requirements Analysis
- Requirements Determination
- Fact-Finding Techniques
- Joint Application Design
- Structured Walkthrough
- Analyzing and Documenting Requirements
- Tools for Documenting Procedures and Decisions
- Structured Analysis
- Data Flow Diagram
- Data Dictionary
- Entity-Relationship Diagrams
- Software Requirements Specification

Systems approach is an organized way of dealing with a problem. Systems Analysis and Design mainly deals with software development activities. Systems analysis is a systematic investigation of a real or planned system to determine the functions of the system, the relationship between functions, and the linkages with other systems. It is an explicit formal inquiry carried out to help a decision maker identify a better course of action and make a better decision than he might otherwise have made. Systems analysis is used when complex issues have to be dealt with and where there is an uncertainty with regard to the outcome of any course of action.

Systems analysis consists of the following:

- Identification and re-identification of objectives, constraints and alternative courses of action.
- Examination of the probable consequences of the alternatives in terms of costs, benefits and risks.
- Presentation of the alternative results in a comparative framework so that the decision maker can make an informed choice.

Systems analysis is used to guide decisions on issues such as national or corporate plans and programs, the use of resources and protection policies, research and development in technology, regional and urban development, educational systems, and health and other social services. Interdisciplinary approach is needed to solve these problems.

There are several specific types of systems analysis for different situations:

- A systems analysis related to public decisions is often referred to as a **policy analysis**.
- A systems analysis that concentrates on comparison and ranking of alternatives on the basis of their known characteristics is referred to as **decision analysis**.
- The part or aspect of systems analysis that concentrates on finding out whether an intended course of action violates any constraints is referred to as **feasibility analysis**.
- Cost-effectiveness analysis is a study where for each alternative the time stream of costs and the time stream of benefits (both in monetary units) are discounted.

Systems analysis and design for information systems has its origin in general systems theory. General systems theory is concerned with developing a systematic and theoretical framework for making decisions. It encourages consideration of all the activities of an organization and its external environment. The systems concept has become most practical and necessary in conceptualizing the interrelationships and integration of operations, wherever computers are being used. Thus, we can say that a system is a way of thinking about organizations and the problems facing them.

1. STAKEHOLDER

Stakeholder is a person who has a share or an interest in an enterprise. It can be said that a company is responsible towards all of its stakeholders. Stakeholders are people who cast their influence upon the company and in turn get influenced by the company. This means that a business has to fulfill the aspirations of many different people ranging from the local population and customers to its own employees and owners. In the present competitive world of business, it is perceived that the stakeholder concept improves the image of a firm and makes it less vulnerable to the issues raised by the pressure groups.

In traditional input-output models of a corporation, the firm gets inputs from investors, employees and suppliers, and converts these inputs into outputs. In this process, the firm also makes profit. In this model, firms only address the needs and wishes of four parties: investors, employees, suppliers and customers.

In contrast to the input-output model, stakeholder theory recognizes that there are other parties such as governmental bodies, political groups, trade associations, trade unions, communities, associated corporations etc., that are also involved, or have stake in the working of the firm. This view of the firm is used to define the specific stakeholders of a corporation as well as examine the conditions under which these parties should be treated as stakeholders. These two questions are important in the stakeholder theory.

Stakeholders in a project are those entities within or outside an organization which:

a. Sponsor a project, or

b. Have an interest or make profit when a project is completed successfully.

Business analysts are also needed to identify the business needs of the firms or corporations to help devise appropriate strategies to take advantage of the opportunities and also find solutions to business problems.

2. SOFTWARE REQUIREMENTS ANALYSIS

Software requirements analysis involves obtaining a clear and thorough understanding of the product to be developed. Requirements analysis includes both fact-finding about how the problem is solved in the current practice as well as forecasting how the planned system might work. It is a part of software engineering task that essentially serves as a bridge between the system level requirement engineering and software design. It is generally performed by a software engineer. In case of a complex business application, a system analyst who is trained in the business part of the application domain performs this task.

Requirements analysis starts with the statement of requirements given by the customer. If the project is not for a particular customer but is generally initiated, then a vision statement is created that briefly describes what the proposed system is about and this is followed by a list of features or services that it is going to provide or the tasks that it is going to support. In the initial stage, the system is viewed as a black box, the services that it renders are found out and the unique interaction scenarios are described for each service.

Requirements analysis is an important task because without understanding the requirements of the client or the problem, software may be produced but that which serves another purpose but not the purpose that is originally intended. Therefore, neither the customer (client) is satisfied nor the business problem is solved or simplified with the help of the software. This results in wastage of time, money and effort, and also leads to frustration among the members of the development team, leaving the customers unsatisfied. It is also difficult, expensive and time consuming to make modifications to the wrongly produced final software product.

Requirements, analysis activities spell out the details of the software's operational characteristics (such as function, data and behavior), provide information on software's interface with other system elements and establish objectives that the software must meet. It allows the software engineer (or software analyst) to refine the software allocation and build data and functional models and behavioral domains within which the software is going to function. It also provides the software designer with a mass of information, functions, and behavior that can be translated to data, architectural interface and component-level designs. In addition, requirements specification helps the developer and the customer to access the quality of the software once it is built.

2.1 Areas of Requirements Analysis

The effort expended in software requirements analysis may be divided into five areas: (i) problem recognition, (ii) evaluation and synthesis, (iii) modeling, (iv) specification, and (v) review. Initially, the analyst makes a study of the system specification and software project plan. It is necessary to consider software in the context of a system and to review the scope or area of influence of software in that system. The next goal is to understand the basic problem elements as perceived by the customers or users.

Evaluating the problem and synthesizing the solution is the next task in requirements analysis. The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behavior in the context of events that affect the system, establish system interface characteristics and highlight additional design constraints. An overall approach or solution would emerge to solve the problem at hand for which the software is about to be developed. Finally the developed requirements are revised for consistency and correctness to reflect the requirements that are gathered.

As an example, consider an inventory control system that is required for a major supplier of auto parts. The problems with the current manual system are: (i) inability to obtain rapidly the status of component, (ii) a turn around period of two to three days to update a card file, (iii) multiple reorders to the same vendor because there is no way to associate vendors with components, and so forth. After identifying the causes, the analyst determines as to what information is to be produced by the new system and the type of input data to be provided to the system. For instance, the customer places his requirements and a daily report is generated which indicates the parts for which requisition has been placed from the inventory and the number of parts that remain after the delivery.

The analyst comes up with one or more solutions after evaluating current problems and desired information (input and output). Initially, the data objects, processing functions, and behavior of the system are defined in detail. Once this information has been gathered, basic architectures for implementation are considered. An analyst may think that a client/server approach would be appropriate for implementing the solution. However, it should be decided whether the software to support this implementation would fall within the scope outlined in the software plan, and if a database management system is required, will the user/customer's need be satisfied. The process of evaluation and synthesis continues until both the analyst and the customer feel confident that the software can adequately be specified for subsequent implementation of the inventory information system.

3. REQUIREMENTS DETERMINATION

Before starting the process of actually building an information system (for instance, designing and developing a software system), one must be sure about the type of information the proposed system should deliver, the way in which it is to be delivered, and the receipients of that information; in short, what are the requirements of the system? A requirement is a feature that must be included in a new system. It may include a way of capturing or processing data, producing information, controlling a business activity, or lending the desired support to the management. The determination of requirements entails studying the existing system and collecting details about it to find out what these requirements are.

Upon reviewing a number of information systems, one would find that the reasons for failure upon the introduction of the system or the reason for exceeding the development budgets are that the requirements from the proposed information system had not been clarified properly. This means that certain requirements are still too vague or even incorrect and are therefore misinterpreted by the system designers and builders.

Since system analysts do not work as managers or employees in functional departments (such as marketing, purchasing, manufacturing, or accounting), they are unaware of the basic information as the managers and users working in those areas have.

Therefore, an initial step that should be taken by the analysts is to investigate and understand the situation. Certain types of requirements are basic, common and fundamental in most of the situations. Developing answers to a specific group of questions will help in understanding these basic requirements. There are also special kinds of requirements that arise, depending on whether the system is oriented for transaction purpose or decision making purpose and whether the system covers several departments. For example, the need to inform the manager dealing with inventory when an unusually large order is forthcoming underscores the importance of linking the sales, purchasing and warehouse departments.

Requirements determination can be viewed as consisting of three major activities. They are:

- Requirements Anticipation,
- Requirements Investigation, and
- Requirements Specification.

3.1 Requirements Anticipation

The study conducted by a system analyst is influenced by his/her previous experience in a particular business area or an environment of a system. Based on this experience, the system analyst predicts the likelihood of certain problems or features and also requirements for a new system having similar characteristics. As a result, the features that are investigated for the current system, questions about the current system that are raised, or methods employed depend upon this familiarity.

Anticipating requirements can be a mixed blessing. On the one hand, experience from previous studies can lead to investigation of areas that would otherwise go unnoticed by an inexperienced analyst. It is essential to hire a system analyst having the background to know what to ask or which aspects to investigate in an organization. On the other hand, if a bias is introduced or shortcuts are taken in conducting the investigation, requirements anticipation becomes a problem.

3.2 Requirements Investigation

This activity is the most important in the systems analysis process. Using a variety of tools and skills, analysts study the current system and document its features for further analysis. Requirements investigation relies on the fact-finding techniques and includes methods for documenting and describing system features.

3.3 Requirements Specifications

The data produced during the fact-finding investigation are analyzed to determine requirements specifications and describe the features for a new system. This activity has three interrelated parts:

- **Analysis of Factual Data:** The data collected during the fact-finding study and included in data flow and decision analysis documents are examined to determine the degree of performance of the system and whether it will meet the needs of an organization.
- **Identification of Essential Requirements:** This includes features that must be included in a new system, ranging from operational details to performance criteria.
- **Selection of Requirements Fulfillment Strategies:** The methods that will be used to achieve the stated requirements are selected. These form the basis for systems design, which follows requirements specification.

All three activities are important and must be performed properly. The responsibility of undertaking requirements specification lies with the system analyst and it is he/she who is responsible for quality of the work.

4. FACT-FINDING TECHNIQUES

It is a strenuous task to gather requirements for a new system or to enhance the features of an existing system. Some of the requirements that need to be gathered relate to finding the key users of the system, the genuine business problem, the separation of wants of the key users from their real needs, motivating the people who understand the problem and discuss requirements with them and also implementing the solution.

The specific methods analysts use for collecting data about requirements are called fact-finding techniques. This is also known as **information gathering** or **data collection**. Fact finding methods are used to interact with people in order to know the scope of the project. One can define the scope through interviews or a group meeting. Effective fact-finding techniques are crucial to the development of systems projects. Fact-finding is performed during all phases of the systems development life cycle. To support systems development, the analyst must collect facts about Data, Processes, Interfaces and Geography. Some of the fact-finding tools include:

- Interviews
- Questionnaires
- Written information analysis (on-site reviews), and
- Observation.

Usually these techniques are used in combination to ensure that an accurate and comprehensive study is undertaken. An analyst usually applies several of these techniques during a single system project.

Following are the areas where fact-finding is applied:

- Fact-finding is important in systems planning and systems analysis phases. It is during these phases that the analyst learns about the problems, opportunities, constraints, requirements and priorities of a business and system.
- During systems design, fact-finding becomes technical as the analyst attempts to learn more about the technology selected for the new system.
- During the systems support phase, fact-finding brings to light whether a system has decayed to a point where it needs to be redeveloped.

The fact-finding techniques are briefly described below:

4.1 Interview

Analysts use interviews to collect information from individuals or from groups. The respondents are generally current users of the existing system or potential users of the proposed system. In some instances, the respondents may be managers or employees who provide data for the proposed system or who will be affected by it. Although some analysts prefer interview to other fact-finding techniques, it is not always the best source of application data.

The respondents and analysts interact during an interview. Interviews provide analysts with opportunities for gathering information from respondents since they have knowledge about the system. This method is the best source of qualitative information such as opinion, policies, and subjective descriptions of activities and problems. Other fact-finding methods are likely to be more useful for collecting quantitative data (numbers, frequencies, and quantities).

This method of fact-finding can be especially helpful for gathering information from individuals who do not have effective writing skills or who may not have the time to respond to questionnaires. Interviews allow analysts to discover areas of misunderstanding, unrealistic expectations, and even dislike the proposed system. The information gathered through interviews is very important because it helps in the analysis of the current system and the construction of data flow diagrams.

In general, there are three categories of individuals to be interviewed:

- **Senior Executives:** These interviews serve both to reinforce the project team's understanding of the business and to gain assurance of the top managements commitment to the project.
- **Middle Managers:** These individuals are interviewed to gather information related to the specifics of the problem.
- **Other Company Personnel:** While information is being gathered, it may become clear that persons outside of top and middle management can contribute critical information. These individuals may be identified during interview or through company records.

It is to be noted that the number of people to be interviewed and the category of people to be interviewed will depend upon the business problem under investigation.

Generally, interviews are often conducted in the interviewee's office or at some location considered convenient by the interviewee. Such a setting should result in greater openness during the interview. Let us consider the two types of interviews that can be used:

a. Structured.

b. Unstructured.

4.1.1 STRUCTURED INTERVIEWS

Structured interviews use standardized questions in either an open-response or closed-response format. The former allows respondents to answer in their own words and the latter uses a set of prescribed answers.

Following are the advantages of structured interviews:

- Ensures uniform wording of questions for all respondents.
- Easy to administer and evaluate.
- More objective evaluation of both the respondents and answers to questions.
- Limited training of interviewer needed.
- Results in shorter time period for completion of interviews.

Following are the disadvantages of structured interviews:

- Cost of preparation is high.
- Respondents may not accept high level of structure and mechanical way of putting questions.
- High level of structure may not be suitable for all situations.
- High level of structure reduces respondent spontaneity and the ability of the interviewer to follow up on comments of the interviewee.

4.1.2 UNSTRUCTURED INTERVIEWS

Unstructured interviews, based on question-and-answer format, are appropriate when analysts want to acquire general information about a system. This format encourages respondents to share their feelings, ideas and beliefs.

Following are the advantages of unstructured interviews:

- Interviewer has greater flexibility in wording questions to suit the level of understanding of the respondent (e.g., 'State the time frame aspect of your scheduling commitment' or 'Are you busy at work?')
- Interviewer can give attention to those areas that arise spontaneously during interview.
- May give information on areas that were overlooked or not thought to be important.

Following are the disadvantages of unstructured interviews:

- May be inefficient with respect to the time at disposal of both the respondent and the interviewer.
- Interviewers may introduce biase or distort the questions or reporting results.
- Extraneous information may get included.
- Analysis and interpretation of results may be lengthy.
- This method involves extra time to collect essential facts.

One interview technique that is often used consists of two-person interviewing team. One member acts as the interview leader, and is responsible for asking questions and directing the course of the discussion. The other member takes notes and is responsible for recording pertinent facts elicited during the interview as well as monitoring the momentum and progress of the interview.

There are two aspects or criteria for conducting a successful interview: One is planning and the other is actual conduct of interview.

The planning process involves:

- Setting a schedule of interviews.
- Researching the interview.
- Allocating a fixed time period for the interview.

Conducting an interview involves the following:

- Arriving at the scheduled time.
- Not exceeding the specified time period.
- Conducting the interview in those areas in which the interviewee has adequate knowledge.
- Not attempting to demonstrate personal knowledge.

4.2 Questionnaire

The questionnaire is a method of data collection which can be used by the analyst in situations where it is impossible to interview all desired respondents because either the physical distances or the number of desired respondents to be covered are too large. The nature of the information required lends itself to this form of collection. A questionnaire is defined as a series of questions for obtaining information on a particular subject. This method does not allow analysts to observe the expressions or reactions of respondents.

Questionnaires with sufficient space for answers can be sent to respondents by mail or handed over in person. The interviewer meeting the respondent has the advantage of allowing explanation of questions and an increase in the likelihood of accurate responses. Various questionnaires are given periodically in order to understand the operations of the organization and also to collect information and opinions from the respondents.

There are two types of questionnaires:

i. Open-ended questionnaires.

ii. Closed questionnaires.

Analysts often use **open-ended questionnaires** to learn about feelings, opinions, and general experiences or to explore a process or problem. It is necessary to ensure that an open-ended question can be answered quickly, briefly, accurately and in common terminology.

Closed questionnaires control the frame of reference by presenting respondents with specific responses to choose from. This format is appropriate for eliciting factual information.

Regardless of the type of questions used, the questionnaire should conform to the general rules:

- It should not be too long. The longer the questionnaire, the less likely people are to complete it.
- Confidentiality of responses (where necessary) should be ensured and communicated to the respondents.
- The use and purpose of the questionnaire should be explained to the respondents.

The way the potential respondents are selected is critical to the value of the results. Clearly, when the sample population is small compared with the total, that sample must be as representative as possible.

Following are the advantages of questionnaires:

- Questionnaires churn out high volume of responses.
- They are inexpensive.
- There is standardized wording in the preparation of questionnarires and most of the time answers are given in a standardized form.
- When compared to the interview method of data collection, the respondents of a questionnaire remain anonymous.
- It is a faster method of data collection.

Following are the disadvantages of questionnaires:

- Limitations on the types of questions (e.g., not suitable for probing questions).
- There is a possibility of misinterpretation of facts and figures.
- The response to questionnaires is generally low.

4.3 Record Review

In order to know the details of the current operation of a business system, the project team should attempt to gather facts from the available documents. In record reviews, analysts examine information that has been recorded about the system and users. Record inspection can be preformed at the beginning of the study, as an introduction, or later in the study, as a basis for comparing actual operations with the recorded version of the information.

Available pertinent documentation should be reviewed. This might include:

Business Documentation:

- Annual reports
- Business plans and forecasts
- Organization charts and manuals
- Company handbooks and manuals
- Literature on advertisements.

Current Systems Environment Documentation:

- Systems descriptions
- Data administration guidelines

- System architecture documentation
- System Flowcharts and database specifications
- Information systems organization charts
- User manuals.

Current Technical Environment Documentation:

- Hardware distribution lists
- Capacity planning documents
- System Software lists
- Data network documentation
- Performance statistics
- Hardware and software acquisition plans.

In short, all the available written material that describes the business and information environments should be considered. They can help analysts understand the system by familiarizing them with the type of operations that must be supported and with formal relations within the organization. In addition, the documents that provide important information should be catalogued for future reference.

4.4 Observation

Interviews are useful in gaining such information from users that is normally not shared with the analyst. Much knowledge is tacit, i.e. it is never verbalized because the user assumes the analyst is already commonplace familiar with the facts. Unfortunately, facts that the user thinks are already in the pocket of the analyst may well be vital for the analyst's understanding. One way to tackle the tacit knowledge problem is to observe things. The analyst observes the users in their workplace, without the users knowing his/her presence.

Observation is very helpful to understand how each employee works and interacts with the current system. For instance, an analyst may observe that the employees are doing paper work and other administrative tasks. It allows analysts to gain information they cannot obtain by means of any other fact-finding method.

Observation entails watching the departmental staff carrying out their various tasks. It is a time-consuming activity and therefore should be undertaken in a scientific manner with a definite purpose.

5. JOINT APPLICATION DESIGN

To determine user requirements in present times, it is necessary to adopt more effective techniques that recognize both the differences in communication styles of employees in the company and the differences in application requirements of the business functions that are undertaken. One such technique is Joint Application Design (JAD).

Joint Application Design is a management process – a people process – which allows Information Systems (IS) to work more effectively with users in a shorter time frame. Since the late seventies, JAD has proven to be an effective technique for building user commitment to the success of application systems through their active participation in the analysis of requirements and the specification of the system design. JAD sessions are conducted in a location away from where the people involved normally work. They are usually held in special-purpose rooms where participants sit around horseshoe-shaped tables.

This is to keep participants away from as many distractions as possible so that they can concentrate on systems analysis.

The following is a list of typical JAD participants:

- **JAD Session Leader –** The JAD leader organizes and runs the JAD. He is a trained individual who plans and leads JAD sessions. He sets the agenda and works towards achievement of objectives of the agenda.
- **Users –** Users are vital participants in a JAD. Users are those individuals who clearly understand purposeful use of the system on a daily basis.
- **Managers –** Managers of the work groups who use the system in question provide insights into new organizational directions, motivations for and organizational impacts of systems, and support for requirements determined during the JAD.
- **Sponsor –** A JAD must be sponsored by someone at a relatively high level in the company such as the vice president or the chief executive officer. If the sponsor attends any sessions, it is usually only at the very beginning or the end.
- **Systems Analysts –** Members of the system analysis team attend the JAD although their actual participation may be limited.
- **Scribe –** The person who makes detailed notes of the happenings at the JAD session.
- **IS Staff –** Besides system analysts, other IS staff such as programmers, database analysts, IS planners, and data center personnel, may attend the session. Their purpose is to learn from the discussion and possibly to contribute their ideas on the technical feasibility of proposed ideas or on the technical limitations of current systems.

The end result of a completed JAD is a set of documents that detail the working of the current system and the features of a replacement system. Depending on the exact purpose of the JAD, analysts may gain detailed information on what is desired of the replacement system.

6. STRUCTURED WALKTHROUGH

Before the phase of the SDLC can begin, the users, management, and development group must review and approve the Baseline Project Plan (BPP). This review takes place before the BPP is submitted or presented to some project approval body, such as an IS steering committee or the person who must fund the project. The objective of this review is to ensure that the proposed system conforms to organizational standards and also make sure that all relevant parties understand and agree with the information contained in the BPP. A common method for performing this review is called walk through or structured walkthrough. Structured walkthrough has proved effective in ensuring quality of an information system.

Although walkthroughs are not rigidly formal or exceedingly long in durations, they have a specific agenda that highlights the areas that need to be covered and the expected completion time. Individuals attending the meeting have specific roles. Which are listed below:

- **Coordinator:** Coordinator plans the meeting and facilitates discussion. He/she may be the project leader responsible for the current life cycle step.
- **Presenter:** Presenter describes the work product to the group. The presenter is an analyst who has usually analysts who have done all or some of the work being presented.
- **User:** User (or group) makes sure that the product meets the needs of the project's customers. Users do not belong to the project team.
- **Secretary:** Secretary takes notes and records decisions or recommendations made by the group. He/she may be a clerk assigned to the project team or one of the analysts working with the team.

- **Standard-bearer:** Standard-bearer ensures that the work product adheres to organization's technical standards. Many larger organizations have staff within the unit responsible for establishing standard procedures, methods, and documentation formats. For example, within Microsoft, user interface standards are developed and rigorously enforced on all development projects. As a result, all systems have the same look and feel to users. These standard bearers validate the work so that it can be used by others in the organization.
- **Maintenance Oracle:** Maintenance oracle person reviews the work product in terms of future maintenance activities. The goal is to make the system and its documentation easy to maintain.

Walkthrough meetings are a common occurrence in most systems development groups. In addition to reviewing the BPP, these meetings can be used for the following activities:

- System specifications.
- Logical and physical designs.
- Code or program segments.
- Test procedures and results.
- Manuals and documentation.

One of the key advantages of using a structured review process is to ensure that formal review points occur during the project. At each subsequent phase of the project, a formal review should be conducted to make sure that all aspects of the project are satisfactorily accomplished before assigning additional resources to the project. This conservative approach of reviewing each major project activity contingent upon successful completion of the prior phase is called incremental commitment. It is much easier to stop or redirect a project at any point when using this approach.

7. ANALYZING AND DOCUMENTING REQUIREMENTS

After requirements are gathered, they are analyzed and documented using an iterative approach. As each requirement is analyzed, it generally leads to further questions. This requires the analyst to probe further till all relevant issues are cleared.

The business analyst must ensure that the requirements are documented in a standard and consistent manner that is easily and clearly understood by all members of the team involved in finding a solution. To do this, the analyst uses text, diagrams or a combination of both.

To manage and communicate requirements more easily, they are categorized into:

- Business Requirements,
- Functional Requirements, and
- Technical Requirements.

A business analyst needs to have the following skills for carrying out analysis:

- Analytical skills.
- Understanding of system development methodologies.
- Modeling techniques.
- Prototyping techniques.
- Documentation.

8. TOOLS FOR DOCUMENTING PROCEDURES AND DECISIONS

Making decisions and following procedures are an integral part of conducting business. For instance, deciding, when to reorder supplies, are less complex, involve fewer people and are guided by step-by-step procedures. Both procedures and decisions are important to systems analysts who are part of the investigation process.

A tool is any device, object, or operation used to accomplish a specific task. Systems analysts rely on tools just as other people do their everyday activities. For example, a mechanic uses wrenches and screwdrivers when working on an automobile, a carpenter uses hammers and saws to prepare furniture and so on. For making furniture, it is not only necessary to know the availability of various relevant tools but also their proper usage.

Tools help analysts to put together information gathered through the data collection methods in a relevant context. But, like all tools, the ones analysts use to document procedures and decisions must be used properly. The three tools used for documenting procedures are:

- Decision Concepts,
- Decision Trees,
- Decision Tables, and
- Structured English.

8.1 Decision Concepts

When analyzing procedures and decisions, the analysts must start identifying conditions, concepts and actions, common to all activities.

8.1.1 CONDITIONS AND DECISION VARIABLES

Conditions vary, which is why analysts may refer to them as decision variables. For example, in business, the handling of an invoice is based on a condition that constitutes a decision variable. In some organizations it is a norm that all invoices be signed before payment can be authorized. The same invoice could also be described by other alternate conditions: authorized or unauthorized, correctly priced or incorrectly priced.

To create documentation, the investigator must identify both the relevant and the permissible conditions that can occur in a situation. Only those conditions that are relevant to the study should be included. The fact that the invoice is signed or unsigned is a relevant variable.

8.1.2 ACTIONS

When all possible conditions are known, the analyst determines the next course of action when certain conditions occur. Actions are alternatives – the steps, activities, or procedures that an individual may decide to take when confronted with a set of conditions. The actions can be quite simple in some cases and extensive in others.

Actions can be related to quantitative conditions. For example, businesses often give discounts on purchased merchandise, depending on the size of the order. A company might give discounts based on the condition, SIZE OF ORDER: over Rs.15,000, Rs.10,000 to Rs.15,000, and less than Rs.10,000. The three actions for the given three conditions respectively are: 3% discount, 2% discount, and no discount, that is, full payment of invoice amount.

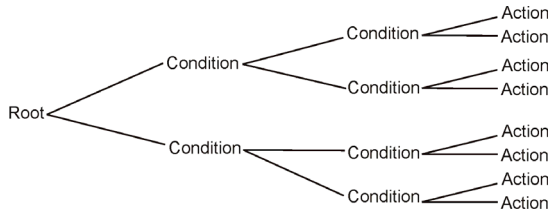
8.2 Decision Trees

Analysts need to organize information collected about decision-making in a standardized manner.

Decision tree is a method for describing and communicating decisions in an unambigious manner. Decision Trees are excellent tools that help in choosing between several courses of action. They provide a highly effective structure within which options can be laid out and possible outcomes can be investigated for choosing those options. They also help in forming a balanced picture of the risks and rewards associated with each possible course of action.

A decision tree is a diagram that presents conditions and actions sequentially and thus shows which conditions to consider first, which second and so on. It is also a method of showing the relationship of each condition and its permissible actions. The decision tree defines the conditions as a sequence of left to right tests. A decision tree helps to show the paths that are possible in a design following an action or decision by the user. Decision trees represent a series of IF … THEN type rules which are linked together and can be used to predict properties for our observations based upon the values of various features. Figure 1 below illustrates the branches of a tree.

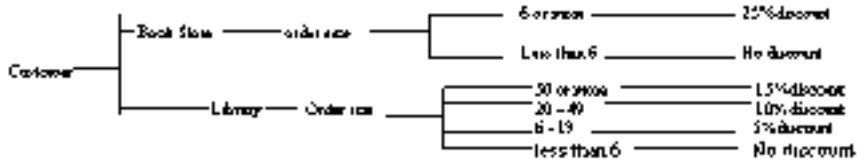
Figure 1: Format of a Decision Tree



8.2.1 DRAWING A DECISION TREE

Consider an example of constructing a decision tree as shown in figure 2. The construction of a decision tree begins with a decision that one has to make i.e., we start at the topmost point in the tree and ask the question “Type of the Customer?”. This point, and all other points in which a question is based upon the data given is called a node, with the first decision node referred to as the root of the tree. From this, draw out lines towards the right for each possible solution, and write that solution. The answer to the question determines the path we take through the tree. This final destination which has no other paths leading away from it is called a leaf, and each leaf has a classification attached to it. The example of a decision tree is given below.

Figure 2: Example of a Decision Tree



After completion, the decision tree is reviewed. It is necessary to keenly observe each and every line to see if there are any solutions or outcomes that have been missed and those are again indicated in the decision tree. After completing the decision tree, the range of possible outcomes will be highlighted.

8.3 Decision Tables

Decision tables and trees were developed long before the advent of computers. They not only isolate many conditions and possible actions but also help ensure that nothing has been overlooked. Decision tables are used to lay out in a tabular form all possible situations which a business decision may encounter and to specify the action to be taken in each of these situations.

A decision table is a matrix of rows and columns, rather than a tree that shows conditions and actions. Decision rules, included in a decision table, state what procedure to follow when certain conditions exist. This method has been used since the mid-1950s, when it was developed at General Electric for the analysis of business functions such as inventory control, sales analysis, credit analysis, and transportation control and routing.

A decision table is a table composed of rows and columns, separated into four separate quadrants as shown below:

Conditions	Condition Alternatives
Actions	Action Entries

The upper left quadrant contains the conditions that will affect the decision or policy. The upper right quadrant contains the condition rules for alternatives. The lower left quadrant contains the actions to be taken and the lower right quadrant contains the action rules. Rules describe which actions are to be taken under a specific combination of conditions. They are specified by first inserting different combinations of condition attribute values and then putting X's in the appropriate columns of the action section of the table.

8.3.1 DEVELOPING DECISION TABLES

In order to build decision tables, it is necessary to determine the maximum size of the table, eliminate any impossible situations, inconsistencies, or redundancies, and simplify the table as much as possible. The following are the guidelines for developing decision tables:

i. Determine the number of conditions that may affect the decision. Combine rows that overlap, for example, conditions that are mutually exclusive. The number of conditions becomes the number of rows in the top half of the decision table.

ii. Determine the number of possible actions that can be taken. This becomes the number of rows in the lower half of the decision table.

iii. Determine the number of condition alternatives for each condition. In the simplest form of decision table, there would be two alternatives (Y or N) for each condition. In an extended-entry table, there may be many alternatives for each condition.

iv. Calculate the maximum number of columns in the decision table by multiplying the number of alternatives for each condition. If there were four conditions and two alternatives (Y or N) for each of the conditions, there would be sixteen possibilities as follows:

Condition 1	:	x	2 alternatives
Condition 2	:	x	2 alternatives
Condition 3	:	x	2 alternatives
Condition 4	:	x	2 alternatives
			16 possibilities

The maximum number of possibilities are 2^N, where, N is the number of conditions.

Fill in the condition alternatives. Start with the first condition and divide the number of columns by the number of alternatives for that condition. In the foregoing example, there are sixteen columns and two alternatives (Y and N), so sixteen divided by two is eight. Then, choose one of the alternatives and write Y in all of the eight columns. Finish by writing N in the remaining eight columns as follows:

Condition 1-YYYYYYYYNNNNNNNN

Repeat this for each condition using a subset of the table:

Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Y	Y	Y	Y	N	N	N	N								
Y	Y	N	N												
Y	N														

Continue the pattern for each condition:

Condition 1	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Condition 2	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Condition 3	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Condition 4	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N

1. Complete the table by inserting an X where rules suggest certain actions.
2. Combine rules where it is apparent that an alternative does not make a difference in the outcome; for example:

Condition 1	Y Y
Condition 2	Y N
Action 1	X X

Can be expressed as:

Condition 1	Y
Condition 2	–
Action 1	X

The dash (–) signifies that condition 2 can be either Y or N and action will still be taken.

1. Check the table for any impossible situations, contradictions, or redundancies.
2. Rearrange the conditions and actions (or even rules) to make the decision table more understandable.

Example: A store wishes to program a decision on non-cash receipts for goods.

The conditions to check are as follows:

1. Transaction under Rs.50.
2. Pays by cheque (guarantee Rs.50).
3. Pays by credit card.

The possible actions that a cashier could take are as follows:

1. Ring up sale.
2. Check credit card from local database.
3. Call a supervisor.
4. Automatic check of credit card company database.

Using the above given rules, we construct a decision table showing all possible combinations of alternatives.

The condition rules are Yes (Y) or No (N)

Under £50	Y	Y	Y	Y	N	N	N	N
Pays by cheque	Y	Y	N	N	Y	Y	N	N
Pays by credit card	Y	N	Y	N	Y	N	Y	N
Ring up sale								
Check from local database								
Call Supervisor								
Check credit card database								

We can see that some of the combination of conditions given in the condition rules are invalid; For instance, the customer cannot pay by cheque AND pay by credit card or not pay by either method. We have decided that these conditions are mutually exclusive. This decision table can be reduced to four condition rules.

Under Rs.50	Y	Y	N	N
Pays by cheque	Y	N	Y	N
Pays by credit card	N	Y	N	Y
Ring up sale				
Check from local database				
Call Supervisor				
Check credit card database				

Indicate the actions.

Under £50	Y	Y	N	N
Pays by cheque	Y	N	Y	N
Pays by credit card	N	Y	N	Y
Ring up sale	X			
Check from local database		X		
Call Supervisor			X	
Check credit card database				X

After constructing a table, analysts verify it for correctness and completeness to ensure that the table includes all the conditions along with the decision rules that relate them to the actions. Analysts should examine the table for redundancy and contradictions.

8.4 Structured English

Structured English is a modified form of the English language used to specify the logic of information system processes. Although there is no single standard, structured English typically relies on action verbs and noun phrases and contains no adjectives or adverbs.

Structured English is used during the analysis stage of a project to identify business processes. For example, “If hours greater than 40 pay fixed rate plus actual – 40 times rate”. Closely related to structured English is pseudocode which is closer to actually writing the program and is written in a form that can be easily converted into programming statements. Pseudocode enables the programmer to concentrate on the algorithm, without caring for the peculiarities of the programming language.

While being characterized as formally-styled natural language, neither structured English or Pseudocode are defined in terms of notation. There is an underlying structure, the use of six specific structured programming constructs: SEQUENCE, WHILE, IF-THEN-ELSE, REPEAT-UNTIL, FOR, and CASE.

Structured English is an additional method to overcome problems of ambiguous language in stating conditions and actions in decisions and procedures. This method uses narrative statements to describe a procedure. It does not show decision rules, it states them. No special symbols or formats are used. Entire procedures can be stated quickly, since only English-like statements are used.

8.4.1 BENEFITS OF STRUCTURED ENGLISH

Structured English is a useful way of specifying procedural rules with a natural order. In situations where the rules are non-procedural, without a natural order, a decision table is more suitable.

Structured English can be useful to describe conditions and actions. When examining a business setting, analysts can use structured English to state decision rules as they are being applied. If analysts cannot state what action to take when a decision is made, it means they need to acquire more information to describe it. On the other hand, after activities have been described in this structured fashion, analysts can take help from other persons to review the narrative and quickly determine whether mistakes or omissions have been made in stating the decision processes.

8.4.2 DEVELOPING STRUCTURED STATEMENTS

The purpose of Structured English is to describe a process in unambiguous terms that can be easily read and understood by a non-technical business user.

Some good style points for Structured English are:

- Use uppercase for keywords and lower case for everything else. This ensures that the logical structure of the process is easily visible.
- Use indentation to reinforce the logical structure of the process.
- Keep the description high-level.

Structured English uses three basic types of statements to describe a process:

i. Sequence Structures.

ii. Decision Structures.

iii. Iteration Structures.

i. **Sequence Structures:** A sequence structure is a single step or action included in a process i.e., a block of steps that should be executed in sequence. It does not depend on the existence of NOT condition, and, when encountered, it is always taken. Several sequence instructions are used together to describe process. The general form is as follows:

BEGIN

Process step 1

Process step 2

...

Process step n

END

For example, in a payroll system, the sequence structure would be as follows:

- Accept employee-ID.
- Accept employee-type.
- Pay base salary.

ii. **Decision Structures:** The action sequences are included within decision structures that identify conditions. Decision structures thus occur when two or more actions can be taken, depending on the value for a specific condition i.e., choosing between alternative courses of action based on multiple conditions. One must assess the condition and then take the decision to take the stated actions or set of actions for that condition. Once the determination of the condition is made, the actions are unconditional. The decision structure, through the use of IF/THEN/OTHERWISE phrases, points out alternatives in the decision process quite clearly. The general form will be as follows:

IF condition 1 THEN

Block of steps 1

(ELSE IF condition n THEN

Block of steps n)

[ELSE

Block of steps otherwise]

END IF

The conditions are examined in turn and the block of statements are executed following the first true condition. There can be zero or more "ELSE IF" parts which are evaluated in the order in which they appear. The "ELSE" part can only occur a maximum of once – it provides an alternative if none of the conditions in the "IF" or "ELSE IF" part are true. Decision structures are not limited to two condition-action combinations. There can be many conditions.

An example in Decision Structure is expressed as follows:

BEGIN

BEGIN IF

IF Employee-Type is Salary

THEN PAY base salary

END IF

BEGIN IF

IF Employee-Type is Hourly

AND Hours-Worked is <40

THEN CALCULATE hourly wage AND PRODUCE Absence Report

END IF

BEGIN IF

IF Employee-Type is Hourly

AND Hours-Worked is 40

THEN CALCULATE hourly wage

END IF

BEGIN IF

IF Employee-Type is Hourly

AND Hours-Worked is >40

THEN CALCULATE hourly wage AND CALCULATE overtime

END IF

END

iii. **Iteration Structures:** Certain activities are repeated while a certain condition exists or until a condition occurs. Iteration instructions permit analysts to describe these cases. The general form is as follows:

SELECT expression

CASE range of values 1

Block of steps 1

(CASE range of values n

Block of steps n)

[ELSE

Block of steps otherwise]

END SELECT

The expression following the "SELECT" keyword should evaluate to a single value. This value is then matched to each of the range of values associated with each "CASE" keyword in the order in which they appear. The block of steps following the first matching case is executed. If no matching "CASE" is found then the block of steps following the "ELSE" is executed (if any are present).

Repetition of a block of steps conditionally:

LOOP [UNTIL condition]

Step 1

Step 2

...

[EXIT WHEN condition]

...

Step n

END LOOP [UNTIL condition]

OR

FOR EACH expression

Step 1

Step 2

...

Step n

END FOR

The same above example in iteration structures will be follows:

DO until all employees are processed

BEGIN IF

IF Employee-Type is Salary

THEN PAY base salary

END IF

BEGIN IF

IF Employee-Type is Hourly

AND Hours-Worked is <40

THEN CALCULATE hourly wage AND PRODUCE Absence Report

END IF

BEGIN IF

IF Employee-Type is Hourly

AND Hours-Worked is 40

THEN CALCULATE hourly wage

END IF

BEGIN IF

IF Employee-Type is Hourly

AND Hours-Worked is >40

THEN CALCULATE hourly wage AND CALCULATE overtime

END IF

END DO

9. STRUCTURED ANALYSIS

Structured Analysis is a widely used system modeling technique for understanding real world systems before they are built. It is based on functional decomposition of the problem domain. However, its inherent weakness is in identification and partitioning of a system's functionality.

Structured analysis is a set of techniques and graphical tools that allow the analyst to develop new kind of system specifications that are easily understandable to the user. Most of them have no tools. The traditional approach focuses on cost/benefit and feasibility analysis, project management, hardware and software selection and personnel considerations. In contrast, structured analysis considers new goals and structured tools for analysis. The new goals specify the following:

a. Graphics should be used wherever possible to help communicate better with the user.

b. Logical and physical systems should be differentiated.

c. A logical system model to familiarize the user with system characteristics and interrelationships before implementation should be built.

The structured tools focus on the data flow diagram, data dictionary, structured English, decision trees and decision tables. The objective is to build a new document called system specification document. This document provides the basis for design and implementation.

In structured analysis there are three views:

- The functional view, made up of data flow diagrams, is the primary view of the system. It defines what is done, the flow of data between things that are done and provides the primary structure of the solution. Changes in functionality result in changes in the software structure.
- The data view, made up of entity relationship diagrams, is a record of what is in the system, or what is outside the system that is being monitored. It is the static structural view.
- The dynamic view, made up of state transition diagrams, defines when things happen and the conditions under which they happen.

Structured analysis has the following attributes:

i. It is graphic. The DFD for example, presents a picture of what is being specified and is conceptually easy to understand the presentation of the application.

ii. The process is partitioned so that one has clear picture of the progression from general to specific in the system flow.

iii. It is logical rather than physical. The elements of system do not depend on vendor or hardware. They specify in a precise, concise and highly readable manner the workings of the system and its implementation.

iv. It is based on the rigoros study of the user area, a commitment that is often taken lightly in the traditional approach to systems analysis.

v. Certain tasks that are normally carried out late in the system development life cycle are moved to the analysis phase. For example, user procedures are documented during analysis rather than later in implementation.

10. DATA FLOW DIAGRAM

Data Flow Diagram (DFD) is "a structured, diagrammatic technique for showing the functions performed by a system and the data flowing into, out of, and within it". It was first developed by Larry Constantine as a way of expressing system requirements in a graphical form which led to modular design.

DFDs show the flow of data from external entities into the system, the movement of data from one process to another, as well as its logical storage. It consists of data flows, processes, sources, destinations, and stores – all described through the use of easily understood symbols. An entire system can be described from the viewpoint of the data it processes with only four symbols. At the same time, data flow diagrams are powerful enough to show parallel activities. Essentially, DFDs describe the information flows within a system. Data flow diagrams are of atwo types. They are:

- Physical Data flow diagrams.
- Logical Data flow diagrams.

Physical data flow diagrams are implementation-dependent. They show the actual devices, department, people, etc., involved in the current system. They can be verified by the users. A physical DFD is a good starting point in developing a logical DFD.

Logical data flow diagrams, in contrast, describe the system independently of how it is actually implemented; that is, they show what takes place, rather than how an activity is accomplished.

Both types of data flow diagrams support a top-down approach to systems analysis, whereby analysts begin by developing a general understanding of the system and gradually explode components in greater detail. Additional details that are added inclued information about control although upper-level general diagrams are drawn without showing specific control issues to ensure focus is on data and processes.

DFDs show the following:

- The processes within the system.
- The data stores (files) supporting the system's operation.
- The information flows within the system.
- The system boundary.
- Interactions with external entities.

10.1 DFD Principles

Following are the principles of DFDs:

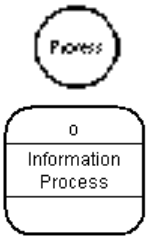
- The general principle in Data Flow Diagramming is that a system can be decomposed into subsystems and subsystems can be decomposed into lower level subsystems and so on.
- Each subsystem represents a process or activity in which data is processed. At the lowest level, processes can no longer be decomposed.
- Each 'process' i.e., a subsystem and an activity in a DFD has the characteristics of a system. All processes must have at least one data flow in and one data flow out. All processes should modify the incoming data, producing new forms of outgoing data. A data flow must be attached to at least one process.

- Just as an operational system must have input and output, a process must also have input and output.
- Data enters the system from the environment; data flows between processes within the system and is produced as output from the system. Each data store must be involved with at least one data flow. Each external entity must be involved with at least one data flow.

10.2 Data Flow Diagram Notations

There are four symbols used in data flow diagrams:

- **Squares representing *external entities*, which are sources or destinations of data:** External Entities, also known as External sources/recipients, are things (For example, people, machines, organizations etc.) which contribute data or information to the system or which receive data/information from it. External entities are represented by rectangles and are out side the system, such as vendors or customers with whom the system interacts. When modeling complex systems, each external entity in a DFD will be given a unique identifier. It is common practice to have duplicates of external entities in order to avoid crossing lines, or just to make a diagram more readable.
- **Rounded rectangles representing *processes*, which take data as input, do something to it, and output it:** A Process represents activities in which data is manipulated by being stored or retrieved or transferred in some way. In other words we can say that process transforms the input data into output data. Circles stand for a process that converts data into information. Processes can be represented either by circles or by rounded rectangles.



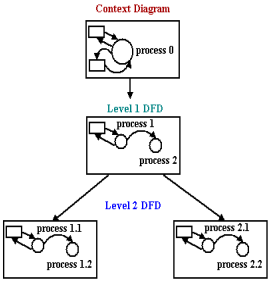
- **Arrows representing the *data flows*, which can either be electronic data or physical items:** Data Flows are represented by a line with an arrow, ⟶

  The arrow shows the direction of flow of data. It depicts data/information flowing to or from a process. The arrows must either start and/or end at a process box. It is impossible for data to flow from data store to data store except via a process, and external entities are not allowed to access data stores directly. Arrows must be named. Double ended arrows may be used with care.
- **Open-ended rectangles representing *data stores*:** These also include electronic stores such as databases or XML files and physical stores such as filing cabinets or stacks of paper. Data stores are repositories of data in the system. A data stored is depicted by two parallel lines or open-ended rectangles. Here, data are to be stored or referenced by a process in the system. The data store may represent computerized or non-computerized devices.

10.3 Data Flow Diagram Layers

DFDs can be drawn in several nested layers. A single process node on a high level diagram can be expanded to show a more detailed data flow diagram. The context diagram is drawn first, followed by various layers of data flow diagrams. The nesting of DFDs is shown in figure 3.

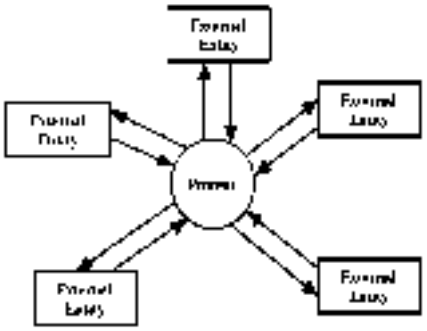
Figure 3: The Nesting of Data Flow Diagrams



10.4 Context Diagrams

A context diagram is a top level (also known as Level 0) data flow diagram. It only contains one process node (process 0) that generalizes the function of the entire system in relationship to external entities. The context diagram is shown in figure 4.

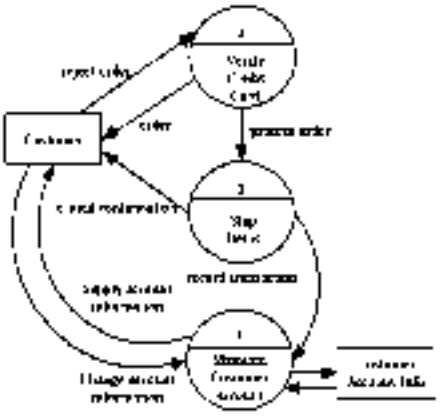
Figure 4: Context Diagram



10.5 DFD levels

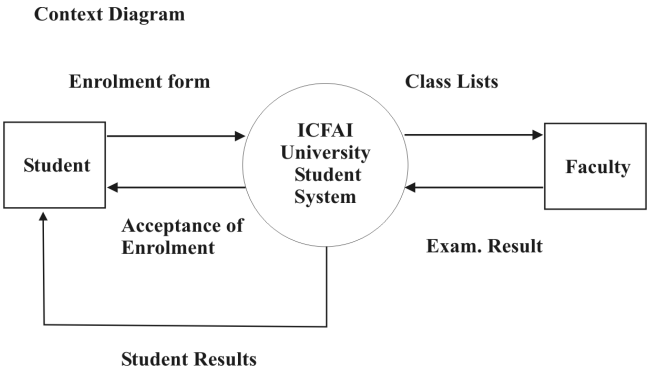
The first level DFD shows the main processes within the system. Each of these processes can be broken into further processes until you reach pseudocode. The first level DFD is shown in figure 5.

Figure 5: Levels of DFD

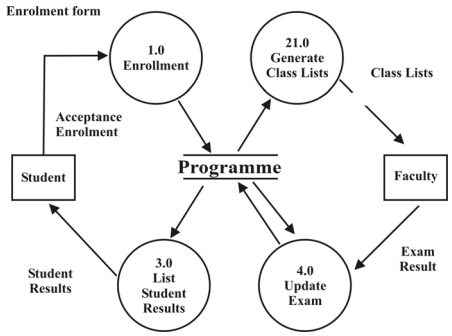


For example, students use an enrolment form to enroll in the program (say MCA) at the ICFAI University. When the form is received, an acceptance-of-enrolment is sent to the students. Later in the term, faculties receive class lists of enrolled students for each subject (say Operating System) that they teach. At the end of the term, the faculties return these lists, having marked each student's exam result on it. When all the listed are returned, students results for all their subjects are posted. This process is depicted in figure 6.

Figure 6
(a) Context Diagram



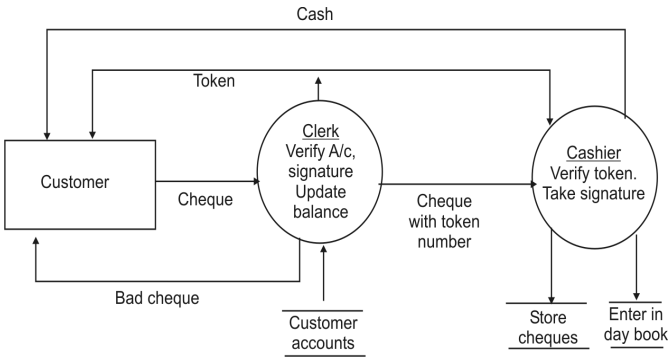
(b) Level 1 Diagram



Consider another example. A customer presents a cheque to a clerk. The clerk checks a ledger containing all account numbers and makes sure that the given account number in the cheque is valid, that adequate balance is there in the account to pay the cheque, and that the signature is authentic. Having done these, the clerk gives the customer a token. The clerk also debits the customer's account by the amount specified on the cheque. If cash cannot be paid due to an error in the cheque, the cheque is returned. The token number is written on the top of the cheque and it is passed on to the cashier. The cashier calls out the token number and the customer goes to the cash counter with the token. The cashier checks the token number, takes the customer's signature, pays cash, enters cash paid in a ledger called day book, and files the cheque.

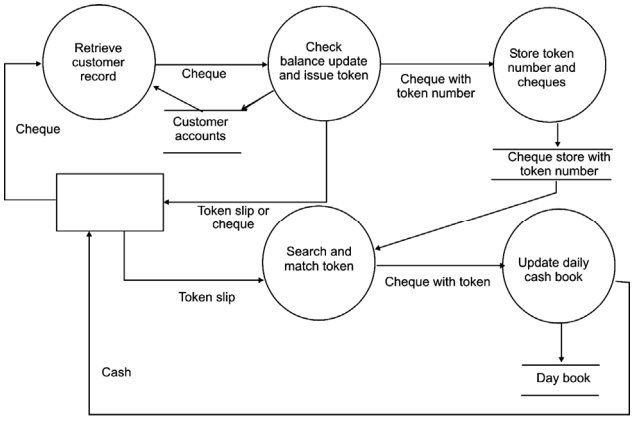
The physical DFD for the process of getting a cheque enchased in a bank is shown in figure 7.

Figure 7: Physical DFD for Encashing Cheque



The same diagram may be converted to a logical DFD as shown in figure 8. In this conversion, one takes the functions performed at each step as same. Each process has well-defined operation. In this diagram one does not include details such as clerks/cashiers performing operations.

Figure 8: Logical DFD for Encashing Cheque



DFDs are used for representing logical processing of data. It is useful to evolve a logical DFD after first developing a physical DFD which shows the persons performing various operations and how data flows between persons performing operations.

11. DATA DICTIONARY

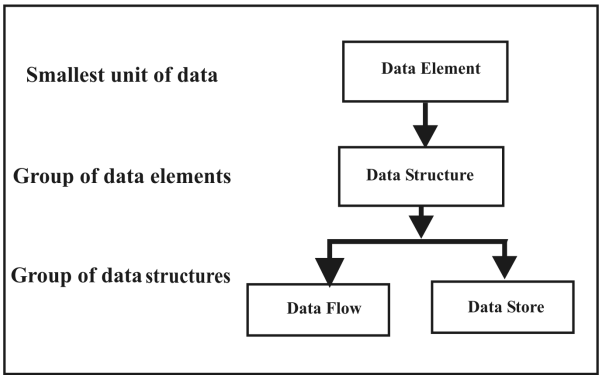
A data dictionary is a catalog or a structured repository that describes the different elements in a system. In other words, it is a structured repository of data about data. It is a rigorous definitions of all DFD data elements and data structures. In a data dictionary a list of all the elements comprising the data flowing through a system is found. The major elements are data flows, data stores, and processes. The data dictionary stores details and descriptions of these elements.

There are three classes of items as given below:

- Data Element.
- Data Structure.
- Data Flows and Data Store.

The three levels that make up the hierarchy of data are shown in figure 9.

Figure 9: Data Hierarchy



11.1 Describing Data Elements

Data Element is the smallest unit of data that provides for no further decomposition. For example, “date” consists of day, month, and year. The description of a data element should include the name, description, and an alias (synonym).

For example,

AUTHOR-NAME	–	first WHISKY	–	name
	–	Middle	–	distiller
	–	Last	–	vintage
	–	Alias		

The description should be a summary of the data element. A data element should have the following:

1. **A Different Name:** For example, a PURCHASE ORDER may exist as PUR. ORDER, PURCHASE ORD, or P. O. All these may be recorded in the data dictionary and included under the PURCHASE ORDER definition and separately with entries of their own. One example is "P.O. alias of PURCHASE ORDER."
2. Usage characteristics, such as range of values or the frequency of use or both. A value is a code that represents a meaning. There are two types of data elements:
    a. **Those that take a value within a range:** For example, a payroll that checks amount between $1 and $10,000 is called a continuous value.
    b. **Those that have a specific value:** For example, departments in a firm may be coded 100 (accounting), 110 (personnel, etc.).

100 means "Accounting Department", 101 means "Accounts Receivable Section", 102 means "Accounts Payable Section".

In either type, values are codes that represent a meaning:

1. Control information such as the source, date of origin, users, or access authorization.
2. Physical location in terms of a record, file or database.

11.2 Describing Data Structures

It is a group of data elements handled as a unit. For example, "phone" is a data structure consisting of four data elements, i.e., area code – exchange – number – extension- for example, 919 – 845 – 1254 – 267. "BOOK DETAILS" is a data structure consisting of the data elements: author name, title, ISBN (International Standard Book Number), LOCN (Library of Congress Number), publisher's name, and quantity. Any data structure is described by specifying the name of each data structure and the elements it represents, provided they are defined elsewhere in the data dictionary. Some elements are mandatory whereas others are optional. The example given below depicts the data structure and data elements.

		Mandatory	Optional
Data structure	Book-Details		
Data elements	Author-Name	X	
	Title of Book	X	
	Edition	X	
	ISBN (International Standard Book Number)		X
	LOCN (Library of Congress Number)		X
	Publisher-Name	X	
	Quality Ordered	X	

11.3 Describing Data Flows and Data Stores

The contents of a data flow may be described by the names of the data structures that pass along it. Using the above BOOK-ORDER example, data flows may be described as given below:

Data Flow	Comments
Book-Details	From Newcomb Hall Bookstore (source)
Author-Name	
Title of Book	
Edition	Recent edition required
Quantity	Minimum 40 copies

Data Store is a location where data structures are temporarily located. The above example BOOK STORE-ORDER may be described as follows:

Data Flow	Comments
Order	
Order-Number	Data flow/data structure feeding data store
Customer-Details	Content of data store
Book-Detail	Data flow/data structure extracted from data store

In constructing a data dictionary, the analyst should consider the following points:

1. Each unique data flow in the DFD must have one data dictionary entry. There is also a data dictionary entry for each data store and process.
2. Definition must be readily by name.
3. There should be no redundancy definitions in the data definition.
4. The procedure for writing definitions should be straight forward but specific.

Data dictionaries are an integral component of structured analysis. If analysts want to know how many characters are in a data item, by what other names it is referenced in the system, or where it is used in the system, they should be able to find the answers in a properly developed data dictionary.

The dictionary is developed during data flow analysis and assists the analysts involved in determining system requirements.

Analysts use data dictionaries for five important reasons. They are as follows:

- To manage the detail in large systems.
- To communicate a common meaning for all system elements.
- To document the features of the system.
- To facilitate analysis of the details in order to evaluate characteristics and determine where system changes should be made.
- To locate errors and omissions in the system.

Manage Detail

Large systems have huge volumes of data flowing through them in the form of documents, reports and even conversations. Similarly, many different activities take place that use existing data or create new details. All systems are ongoing all the time. Those who try invariably make mistakes or forget important elements. The best analysts don't even try – they record the information. Some write the

descriptions on paper and others use index cards. Many enter the details into a word processor running on a personal computer. The best-organized and most effective analysts use automated data dictionaries designed specifically for system analysis and design.

Communicate Meaning

Data dictionaries assist in ensuring common meaning for system elements and activities. Data dictionaries record additional details about the data flow in a system so that all persons involved can quickly look up the description of data flows, data stores or processes.

Document System Features

Documenting the features of an information system is the third reason for using data dictionary systems. Features include the parts or components and the characteristics that distinguish each. Once the features have been articulated and recorded, all participants in the project will have a common source for information about the system.

Facilitate Analysis

The fourth reason for using data dictionaries is to determine whether new features are needed in a system or whether changes of any type are required. For example, one is working with a university that is considering allowing its students to register for courses by dialing into an online registration system over touch-tone telephones. What questions would one ask and what information would one want to have available for examination? For any situation, systems analysts will typically focus on the following system characteristics:

Nature of Transactions

The business activities that will be carried on while using the system, including the data needed to accept, authenticate, and process each activity are said to be the nature of a transaction.

Example: Will the system permit the processing of course registration transactions where payment is by credit card? What additional features are needed to permit registration by touch-tone telephone. How will payments be received if students do not choose to pay by credit card?

Inquiries

Requests for retrieval of information or processing to generate a specific response are called inquiries.

Example: Student and course descriptive data are in two separate files and are currently not linked together. How can one make the data jointly available for advisors who wish to assist students in program planning and course scheduling?

Output and Report Generation

It is the result of system processing (out put) presented to users in an acceptable form (report generation).

Example: How can one identify those students who will register for courses over touch-tone telephones so that they can be listed on a separate report? How does one provide these same students with a signed registration record as one now does for those registering on-site?

Files and Databases

Details of transactions and master records of concern to the organization are called files and databases.

Example: What data must be captured to verify the accuracy and authenticity of transactions arriving over telephones?

System Capacity

The ability of the system to accept, process and store transactions and data are termed as system capacity.

Example: How many students can register simultaneously over touch-tone telephones? What are the current and anticipated numbers of students that can be registered in one hour?

Locate Errors and Omissions

A dictionary which stores lot of information about a student requstration system such as transactions, inquiries, type of data and capacity would help us to know a lot about the purpose of the system and help in evaluating the system. The information itself should be complete and accurate. Therefore, dictionaries are used for a fifth reason: to evaluate and locate errors in the system description.

12. ENTITY-RELATIONSHIP DIAGRAMS

Entity-Relationship (E-R) model is a conceptual data model that views the real world as entities and relationships. The E-R diagram is a model of entities in the business environment, the relationships among the entities, and the attributes or properties of both the entities and their relationships. A basic component of the model is the Entity-Relationship diagram which is used to visually represent data objects.

The overall logical structure of a database can be expressed graphically by an E-R diagram:

Rectangles: Represent entity sets.

Ellipses: Represent attributes.

Diamonds: Represent relationship sets among entity sets.

Lines: Link attributes to entity sets and entity sets to relationship sets.

Double ellipse: Represent multivalued attributes.

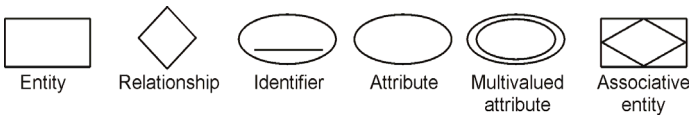
Dashed lines: Represent derived attributes.

Double lines: Indicate total participation of an entity in a relationship.

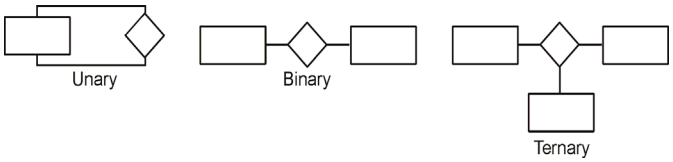
Double rectangle: Represent weak entity sets.

The basic symbols used in E-R diagrams are shown below.

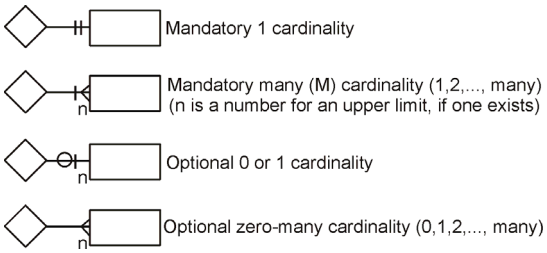
Figure 10: Basic Symbols of E-R Diagrams



The following Relationship symbols are used in E-R diagrams:



The following symbols show the relationship cardinality in E-R diagrams:



An **entity** is anything that can be distinguished from any other thing. An entity is an abstract or concrete thing in the real world that we want to model in a database. It is described using a set of attributes. Every attribute is connected to one and only one entity.

To identify an entity the best approach is defining a noun from the list of attributes.

For example, in Customer Name, Customer is the entity and in payment mode, payment is entity etc.

Entity type is a collection of entities that share common properties or characteristics. An **entity instance** is a single occurrence of an entity type. An entity type is described just once in a data model, whereas many instances of that entity type may be represented by data stored in the database.

Attribute is a named property of an entity that is of interest to the organization.

For example, when Student is an entity, student-id, student-name, and student-address will be the attributes of the Student entity.

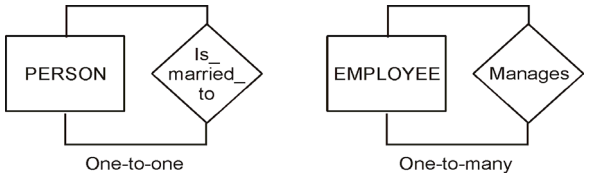
Single valued attributes can take only one value for a given entity instance. For example, Loan-number attribute for a specific loan entity refers only one number. **Multivalued attributes** may take more than one value for a given entity instance. For example, Phone-number attribute for a specific employee entity set may have more than one numbers. **Derived attributes** are those whose values can be calculated from related attribute values. If employee entity set has a date of birth of employee, then we calculate the age by subtracting the value of date of birth from today's date.

A **Relationship** represents an association between two or more entities. In a relational database, all entities have bonds between them, expressed as relationships. A relationship is a link between two entities, and it tells us something about which relationships exist between the entities. The **degree of a relationship** is the number of entities associated with the relationship. The three most common relationships in E-R diagrams are:

1. Unary Relationship.
2. Binary Relationship.
3. Ternary Relationship.

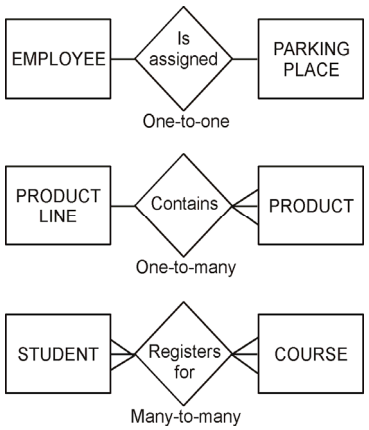
Unary Relationship is also called as recursive relationship. It is a relationship between the instances of one entity type. Figure 11 shows the example of unary relationship in E-R Diagrams:

Figure 11: Unary Relationship in E-R Diagrams



Binary relationship is a relationship between instances of two entity types. Figure 12 shows the binary relationship in E-R Diagrams:

Figure 12: Binary Relationship in E-R Diagrams



Ternary relationship is a simultaneous relationship among instances of three entity types. Figure 13 shows the ternary relationship in E-R Diagrams.

Figure 13: Ternary Relationship in E-R Diagrams

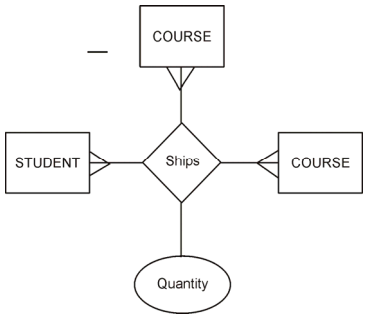
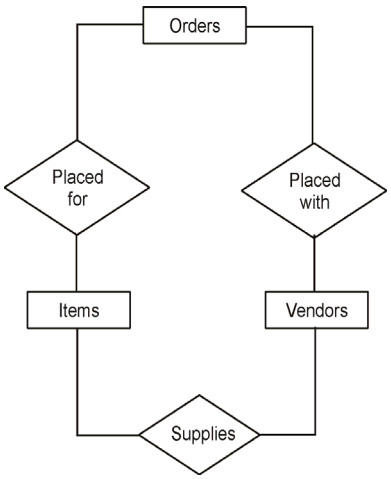


Figure 14 shows an E-R diagram for orders placed with vendors for supply of items.

Figure 14: An Example of E-R Diagram



13. SOFTWARE REQUIREMENTS SPECIFICATION

Once the analysis is complete, the requirements must be written or specified. The final output is the Software Requirements Specification (SRS) document. For smaller problems or problems that can easily be comprehended, the specification activity might come after the entire analysis is complete. However, it is more likely that problem analysis and specification are done concurrently. An analyst typically will analyze some parts of the problem and then write the requirements for that part.

The quality assurance goal of SRS makes it necessary to generate the requirements document that provides the technical specifications for the design and development of the software. This document enhances the system's quality by formalizing communication between the system developer and the user and provides proper information for accurate documentation.

The software design document defines the overall architecture of the software that provides the functions and features described in the software requirements document. It addresses the question, how it will be done? The document describes the logical subsystems and their respective physical modules. It ensures that all conditions are covered.

The quality assurance goal of the testing phase is to ensure completeness and accuracy of the system and minimize the retesting process. In the implementation phase, the goal is to provide a logical order for the creation of the modules and, in turn, the creation of the system.

Maintenance and support is necessary to ensure that the system continues to comply with the original specifications. The quality assurance goal is to develop a procedure for correcting errors and enhancing software. This procedure improves quality assurance by encouraging complete reporting and logging of problems, ensuring that reported problems are promptly forwarded to the appropriate group for resolution, and reducing redundant effort by making known problem reports available to any department that handles complaints.

13.1 Characteristics of an SRS

To properly satisfy the basic goals, an SRS should have certain properties and should contain different types of requirements. In this section, we discuss some of the desirable characteristics of an SRS. We will also discuss the different components of an SRS. A good SRS should have the following attributes:

- Understandability
- Unambiguousness
- Completeness
- Verifiability
- Consistency
- Modifiability
- Traceablity.

An SRS should be **understandable**, as one of the goals of the requirements phase is to produce a document upon which the client, the users and the developers can agree. Since multiple parties need to understand and approve the SRS, it is of utmost importance that the SRS should be understandable.

An SRS is **unambiguous** if and only if every requirement stated has one and only one interpretation. Requirements are often written in natural language, which are inherently ambiguous. If the requirements are specified in a natural language, the SRS writer has to be most careful to ensure that there are no ambiguities. One way to avoid ambiguities is to use some formal requirements specification language. The major disadvantage of using formal languages is the large effort required to write an SRS and the high cost of doing so.

An SRS is **complete** if everything the software is supposed to do is in the SRS. A complete SRS defines the responses of the software to all classes of input data. For specifying all the requirements, the requirements relating to functionality performance, design constraints, attributes and external interfaces must be specified. In addition, the responses to both valid and invalid input values must also be specified.

A requirement is **verifiable** if there exists some cost-effective process that can check if the final software meets that requirement. An SRS is verifiable if and only if every stated requirement is verifiable. This implies that the requirements should have as little subjectivity as possible because subjective requirements are difficult to verify. Unambiguity is essential for verifiability.

An SRS is **consistent** if there is no requirement that conflicts with another. Terminology can cause inconsistencies; for example, different requirements may use different terms to refer to the same object. There may be logical or temporal conflict between requirements causing inconsistencies. This occurs if the SRS contains two or more requirements whose logical or temporal characteristics cannot be satisfied together by any software system. For example, suppose a requirement states that an event ‘e’ is to occur before another event ‘f’. But then another set of requirements state (directly or indirectly by transitivity) that event ‘s’ should occur before event e. inconsistencies in an SRS can be a reflection of some other major problems.

Writing an SRS is an iterative process. Even when the requirements of a system are specified, they are later modified as the needs of the client change with time. Hence, an SRS should be easy to modify. An SRS is **modifiable** if its structure and style is such that any necessary change can be made easily, while preserving the completeness and consistency. Presence of redundancy is a major hindrance to modifiability, as it can easily lead to errors. For instance, assume that a requirement is stated in two places, and that at a later time the requirement needs to be changed. If only one occurrence of the requirement is modified, the resulting SRS will be inconsistent.

An SRS is **traceable** if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development. Forward traceability means that each requirement should be traceable to some design and code elements. Backward traceability requires that it should be possible to trace design and code elements to the requirements they support. Traceability aids verification and validation.

Of all these characteristics, **completeness** is perhaps the most important. The most common problem in requirements specification is when some of the requirements of the client are not specified. This necessitates additions and modifications to the requirements later in the development cycle, which are often expensive to incorporate. Incompleteness is also a major source of disagreement between the client and the supplier. The importance of having complete requirements cannot be over-emphasized.

13.2 Components of an SRS

Completeness of specifications is difficult to achieve and even more difficult to verify. Having guidelines about what different thins an SRS should specify will help in completely specifying the requirements. Here, we will discuss some of the system properties that an SRS should specify. The basic issues which an SRS must address are:

1. Functionality.
2. Performance.
3. Deign constraints imposed on an implementation.
4. External interfaces.

Functional Requirements

Functional requirements specify which outputs should be produced for the given inputs. They describe the relationship between the input and output of the system. For each functional requirement, a detailed description of all the data inputs and their source, the units of measure, and the range of valid inputs must be specified.

All the operations to be performed on the input data to obtain the output should be specified. This includes specifying the validity checks on the input and output data, parameters affected by the operation, and equations or other logical operations which must be used to transform the inputs in to corresponding outputs. For example, if there is a formula for computing the output, it should be specified. Care must be taken not to specify any algorithms that are not part of the system, but which may be needed to implement the system. These decisions should be left for the designer.

An important part of the specification is the system behavior in abnormal situations, like invalid input (which can occur in many ways) or error during computation. The functional requirement must clearly state what the system should do if such situations occur. Specifically it should specify the action that needs to be taken in those situations where the input is valid but it is not possible to carry out normal operations. An example of this situation is an airline reservation system, where a reservation cannot be made even for valid passengers, if the airplane is

fully booked. In short, the system behavior for all foreseen inputs and for all foreseen system states should be specified. These special conditions are often likely to be overlooked, resulting in a system that is not robust.

Performance Requirements

This part of an SRS specifies the performance constraints on the software system. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements:

1. Static Requirements.
2. Dynamic Requirements.

Static requirements are those that do not impose constraints on the execution characteristics of the system. These include requirements such as the number of terminals to be supported, the number of simultaneous users to be supported, the number of files and their sizes that the system has to process. These are also called *capacity* requirements of the system.

Dynamic requirements specify constraints on the execution behavior of the system. These typically include response time and throughput constraints on the system. Response time is the expected time for the completion of an operation under specified circumstances. Throughput is the expected number of operations that can be performed in a unit time. For example, the SRS may specify the number of transactions that must be processed per unit time, or what the response time for a particular command should be. Acceptable ranges of the different performance parameters should be specified as well as acceptable performance for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms. So, requirements like “response time should be good”, or the system must be able to “process all the transactions quickly”, are not desirable as they are imprecise and are not appropriately quantifiable. Instead, statements like “in 90% of all cases, the response time of command should be less than one second”, or “in 98% of all cases, a transaction should be processed in less than one second” should be used to declare performance specification.

Design Constraints

There are a number of factors present in the client’s environment that may restrict the choices of a designer. Such factors include: standards that must be followed, resource limits, operating environment, reliability and security requirements, and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints:

Standard Compliance: This specifies the requirements for the standards that the system must follow. The standards may include the report format and accounting procedures. There may be audit tracing requirements, which require certain kinds of changes or operations that must be recorded in an audit file.

Hardware Limitations: The software may have to operate on some existing or pre-determined hardware, thus imposing restrictions on the design. Hardware limitations can include the type of machines to be used, operating system available on the system, languages supported, and limits on primary and secondary storage.

Reliability and Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex and expensive. Requirements about system behavior in the face of certain kinds of faults are specified. Recovery requirements often form integral part of satisfying certain properties that describe what the system should do if some failure occurs in order to ensure certain properties. Reliability requirements are very important for critical applications.

Security: Security requirements are particularly significant in defense and many other database systems. Security requirements place restrictions on the use of certain commands, control access to data, provide different kinds of access requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system.

External Interface Requirements

All the possible interactions of the software with people, hardware, and other software should be clearly specified. For the user interface, the characteristics of each user interface of the software product should be specified. User interface is increasingly becoming more important and must be given proper attention. A preliminary user manual should be created with all user commands, screen formats, an explanation of how the system will appear to the user, and feedback and error messages. Like other specifications, these requirements should also be precise and verifiable. So, a statement like "the system should be user friendly", should be avoided in preference to statements like "commands should be no longer than 6 characters", "command names should reflect the function they perform" etc.

For hardware interface requirement, SRS should specify the logical characteristics of each interface between the software product and the hardware components. If the software is to execute on existing hardware or on pre-determined hardware, all the characteristics of the hardware, including memory restrictions, should be specified. In addition, the current use and load characteristics of the hardware should be given.

The interface requirement should specify the interface with other software which the system will use or which will use the system. This includes the interface with the operating system and other applications. The message content and format of each interface should be specified.

13.3 Structure of Requirements Document

All the requirements for the system have to be included in a document. The requirements document should be clear and concise. For this, it may be necessary to organize the requirements document into sections and subsections. There can be many ways to structure a requirements document. The outline document will be as follows:

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, Acronyms and Abbreviations
   - 1.4 References
   - 1.5 Overview
2. General Description
   - 2.1 Product Perspective
   - 2.2 Product functions
   - 2.3 User Characteristics
   - 2.4 General Constraints
   - 2.5 Assumptions and Dependencies

3. Functional Requirements
    - 3.1 Functional Requirement 1
    - 3.1.1 Introduction
    - 3.1.2 Inputs
    - 3.1.3 Processing
    - 3.1.4 Outputs
    - 3.2 Functional Requirement 2
4. External Interface Requirements
    - 4.1 User Interfaces
    - 4.2 Hardware Interfaces
    - 4.3 Software Interfaces
5. Performance Requirements
6. Design Constraints
    - 6.1 Standards Compliance
    - 6.2 Hardware Limitations
7. Other Requirements

The introduction section contains the purpose, scope and overview of the requirements document. Section 2 describes the general factors that affect the product and its requirements. Product prospective is the relationship of the product to other products. Section 3, 4, 5 and 6 specify the particular requirements. If there are any other requirements that have not been described they are specified in Section 7.

There are other ways of organizing a requirements document. The key concern is that once requirements have been identified, the requirements document should be organized in such a manner that its validation and system design are carried out in an easy way.

SUMMARY

- System analysis is a systematic investigation of a real or planned system to determine the functions of the system and how they relate to each other and to any other system.
- The determination of requirements involves studying the existing system and collecting details about it to find out what these requirements are. Requirements *determination* consists of the three activities of requirements anticipation, requirements investigation, and requirements specification.
- Fact-finding methods such as interview, questionnaire, on-site record review, and observation assist analysts if used properly. Each has particular advantages and disadvantages; none is adequate by itself. Cross-checking the information obtained during fact finding is very important.
- JAD is a structured process in which users, managers, and analysts work together for several days in a series of intensive meetings to specify or review system requirements.
- When initiating a study, analysts want to know why and how certain activities are performed and what data are used in the work. Timing, frequency and volume of activities are also important facts to collect.

- Decision trees are presentations of decision variables that are graphic and sequential, showing which conditions to consider first, which second, and so on. The root of a decision tree is the starting point for analyzing a specific situation, and the branches indicate the sequence of decisions leading up to the proper action to be taken.
- An alternative tool, decision tables, relates conditions and actions through decision rules. A *decision rule* states the conditions that must be satisfied for a particular set of actions to be taken. The decision rule incorporates all the conditions that must be true at one time, not just one condition.
- Structured English is used to state decision rules. The three types of statements are termed sequence structures, decision structures, and iteration structures. These statements show unconditional actions, repetitive actions, and actions that occur only when certain conditions occur.
- Structured English offers a concise way of summarizing a procedure, where decisions must be made and actions taken. It can be reviewed by other persons quite easily so that possible misunderstanding and mistakes can be detected and corrected.
- Structured Analysis is a widely used system modeling technique for understanding real world systems before they are built. It is based on functional decomposition of the problem domain.
- Data flow analysis consists of four tools: data flow diagrams, data dictionaries, data structure diagrams, structure diagrams, and structure charts. The first two are developed during requirements determination while the later two are most useful during systems design.
- A data flow diagram is a graphic description of a system or portion of a system. It consists of data flows, processes, sources, destinations, and stores all described through the use of easily understood symbols.
- Physical data flow diagrams are implementation-dependent. They show the actual devices, departments, and people etc., involved in the current system. Logical data flow diagrams, in contrast, describe the system independently of how it is actually implemented. That is, they show what takes place, rather than how an activity is accomplished. Both types of data flow diagrams support a top-down approach to systems analysis, whereby analysts begin by developing a general understanding of the system and gradually explore components in greater detail. As details are added, information about control can also be included, although upper-level general diagrams are drawn without showing specific control issues to ensure focus on data and processes.
- The data dictionary stores descriptions of data items and structures, as well as systems processes. It is intended to be used to understand the system by analysts, who retrieve the details and descriptions it stores, and during systems design, when information about such concerns as data length, alternate names (aliases), and data use in specific processes must be available. The data dictionary also stores validation information to guide the analysts in specifying controls for the system's acceptance of data.
- The dictionary also contains definitions of data flows, data stores, and processes.

- Data dictionaries can be developed manually or using automated systems. Automated systems offer the advantage of automatically producing data elements, data structures, and process listings. They also perform cross-reference checking and error detection, important advantages when working on large systems that must be error free. Automated dictionary systems are becoming the norm in the development of computer systems.
- The E-R diagram is a model of entities in the business environment, the relationships among the entities, and the attributes or properties of both the entities and their relationships.
- Once the analysis is complete, the requirements must be written or specified. The final output is the Software Requirements Specification Document (SRS). Each and every SRS contains some characteristics and also some components like functionality, performance and design.

Chapter III

System Design

After reading this chapter, you will be conversant with:

- Design Objectives
- Designing an Information System
- Design Specifications
- System Flowcharts
- Structured Flowcharts
- Database Design
- File Organization
- Design of Computer Output
- Design of Input
- User Interface Design
- Designing Interfaces and Dialogues
- Coupling and Cohesion

When a problem is encountered, a solution must be designed. Thus, the design phase is the first step in proceeding from "problem domain" towards "solution domain". In finding a solution, requirements are translated into ways of meeting them. The deign phase focuses on the detailed implementation of the system recommended in the feasibility study. Emphasis is on translating performance specifications into design specifications. The design phase is a transition from a user-oriented document (system proposal) to a document oriented to the programmers or data base personnel. The objective of the software design phase is to transform the contents of Software Requirement Specification (SRS) into design that is implemented using some programming language i.e., plan a solution for the requirements specified in the SRS document. It is an iterative process which transforms the requirements into a "blue print" for construction. The design will determine the success of the system. Through design, system analysts can influence the effectiveness of a user, whether for the processing of transactions or for managing the activities of the organization. Output of design phase is called the System Design Specification (SDS) Document.

1. DESIGN OBJECTIVES

System Design is a bridge between the requirements specifications and the final solution. Systems design goes through two phases of development:

- Logical design.
- Physical design.

1.1 Logical Design

A Data Flow Diagram (DFD) shows the logical flow of a system and defines the boundaries of the system. It describes the inputs (source), outputs (destination), databases (data stores), and procedures (data flows), all in a form that meets the user's requirements. When analysts prepare the logical system design, they specify the user needs at a level of detail that virtually determines the information flow into the system and the required data resources. The design covers the following:

i. Reviews the current physical system, its data flows, file contents, volumes, frequencies, etc.

ii. Prepares *output* specifications that is, it determines the format, content, and frequency of reports, including terminal specifications and locations.

iii. Prepares *input* specifications that is, it determines format, content and most of the input functions. This includes determining the flow of the document from the input data source to the actual input location.

iv. Prepares security and control specifications. This includes specifying the rules for edit correction backup procedures and the controls that ensure processing and file integrity.

v. Specifies the implementation plan.

vi. Prepares a logical design walkthrough of the information flow, output and input, controls, and implementation plan.

vii. Reviews benefits, costs, target dates and system constraints.

1.2 Physical Design

Physical design follows logical design. It produces the working system by defining the design specifications that specify the tasks that the system must carry out. In the design stage, the programming language and the platform in which the new

system will run are also decided. Specifically, physical system design consists of the following steps:

i. Design the physical system:
   - Specify input/output media.
   - Design the database and specify back up procedures.
   - Design physical information flow through the system and a physical design walkthrough.

ii. Plan system implementation:
   - Prepare a time-table for conversion and a target date.
   - Determine training procedure, courses and timetable.

iii. Devise a test and implementations plan and specify any additional hardware/ software.

iv. Make available benefits, costs, conversion date, and system constraints (legal, financial, hardware, etc).

2. DESIGNING AN INFORMATION SYSTEM

The objectives in designing an information system are as follows:

- Specify the Logical Design Elements
- Support Business Activities
- Meet User Requirements
- Making available a system that is convenient and easy to use.
- Making available Software Development Specifications
- Conform to Design Standards.

Now, we shall discuss these objectives in detail:

2.1 Specify the Logical Design Elements

In Systems designing, the first step is logical design and this is followed by physical construction of the system. When analysts formulate a logical design, they write the detailed specifications for the new system; they highlight its features: the outputs, inputs, files and databases, and procedures consistent with project requirements. The statement of these features is termed as the design specifications of the system.

The logical design of an information system is like the engineering blueprint of an automobile: it shows the major features (such as the engine, transmission, and passenger areas) and their relationship to one another. The reports and outputs of the analyst are like the components designed by an engineer. Data and procedures are linked together to produce a working system.

In designing an inventory system, for example, the system specifications include reports and output screen definitions describing stock on hand, stock additions and withdrawals, and summarizing transactions that occur throughout, say, in a month of operation. The logical design also specifies input forms and screen layouts for all transactions and files for maintaining stock data, transaction details, and supplier data. Procedure specifications describe methods to enter data, produce reports, copy files, and discover problems if any.

Physical construction activity, which follows logical design, produces program software, files and a working system. Design specifications instruct programmers about the activities of the system. The programmers in turn write the programs that accept input from users, process data, produce reports, and store data in the files.

2.2 Support Business Activities

A fundamental objective in the design of an information system is to ensure that it supports the business activity for which it is developed. The computer and communications technology specified in the design should always be secondary to the results the system is intended to produce.

For example, if it is essential for an organization to transfer information very quickly, along its various units to remain competitive, then the design specifications of the information system must be based around this essential business requirement. In this environment, a system that is not efficient in transferring information at the requisite speed will hamper business. For example, an airline reservation system that does not tell users quickly whether seats are available on a specific flight or an automated teller system in a bank that is operationally inefficient for customers to instantly find out their current account balance are not of any use. These activities are essential to their respective businesses and the information system must support them.

The design must fit the way a firm does business. If a sales system is designed to work best for orders that are paid in cash, when the firm offers a liberal credit facility, neither management nor customers will be very happy. Even if the information system functions well in technical terms, it will not fulfill the business objective.

These examples demonstrate the importance of fitting the system to the needs of the business, an objective that should guide virtually all systems design decisions.

2.3 Meet User Requirements

User requirements are translated into system characteristics during design. An information system satisfies user needs if it accomplishes the following:

- Satisfies the right procedures properly.
- Presents information and instructions in an acceptable and effective manner.
- Produces accurate results.
- Provides an acceptable interface and method of interaction.
- Is perceived by users as a reliable system.

In systems analysis, the priority is to obtain the right system along with getting the system right. The objective of systems design is to achieve both of these objectives.

2.4 Convenient and Easy to Use

Many technical features of an information system such as reliability, accuracy, and processing speed are secondary to the human aspects of a system design. Therefore, analysts strive to design the system that can be easily engineered for the convenience of the people by including desirable ergonomic features.

2.4.1 HUMAN ENGINEERING

After the information system is installed, managers and staff members begin interacting with the system on an ongoing basis. After the use of installed system becomes routine, end-users scrutinize and test its features. It is in this context that human engineering features often exceed technical features in importance. If information systems are not designed for people, they may fail.

The analyst should strive to design a system that:

- Incorporates system features that are easy to understand and use.
- Does not allow user errors or carelessness.
- Prevents failures or improper procedures that will give incorrect results or be detrimental to the organization.
- Provides enough flexibility to accommodate different user needs of individuals.
- Is convenient with regard to user's familiarity with the system.
- Generally functions in a manner that seems natural to the user.

2.4.2 ERGONOMIC DESIGN

Ergonomics deals with the physical factors of an information system that affect the performance, comfort and satisfaction of direct users. The design of terminals, chairs, and other equipment influences the amount of fatigue and strain that accompanies the usage of these items. These factors in turn result in the introduction of errors during data entry, user efficiency and even absenteeism.

Ergonomics plays a vital role in the selection of equipment and in the design of work areas. Ergonomic factors must also be considered when selecting colors for the presentation of information, location of command keys, or methods of interaction with the system.

2.5 Making Available Detailed Software Development Specifications

Systems design includes formulating software specifications. The specifications state input, output, and the processing functions and algorithms used to perform them. Software modules and routines that perform specific functions and the procedures for constructing them are specified as well. Selection of programming languages, software packages, and software utilities occurs during the logical design process and the recommendations are included in the software specifications.

2.6 Conform to Design Standards

The objectives of systems design are broad and affect many aspects of both the application and the organization in which the system will be used. Hence systems development standards are essential. Systems design specifications are established within these standards. The examples of design standards are:

- ***Data Standards:*** Guidelines for data item names, length, and type specifications that are used for all applications developed by the information systems group.
- ***Coding Standards:*** Formal abbreviations and designations to describe activities and entities within an organization (e.g., customer categories and transaction types).
- ***Structural Standards:*** Guidelines on structuring the system, reuse of software modules and software. Policies on software modularization, structured coding and the interrelation of system components.
- ***Documentation Standards:*** Descriptions of systems design features, interrelation of components, and operating characteristics that can be reviewed to learn the details of the application.

Many organizations have a quality control department that is responsible for reviewing all information system design specifications as well as the completed system itself to ensure that the application conforms to standards.

3. DESIGN SPECIFICATIONS

Design specification describes the features and components of the system. It specifies the features a system analyst must design. The system elements must be addressed in the formal design specifications.

3.1 Elements of the Design

The components of an information system described during requirements analysis are the focal points in systems design. Analysts must design the following elements:

- Data Flows
- Data Stores
- Processes
- Procedures
- Controls
- Roles.

Data Flows: The movement of data into, around, and out of the system.

Data Stores: Temporary or permanent collection of data.

Process: Activities to accept, manipulate, and deliver data and information. May be manual or computer-based.

Procedures: Methods and routines for using the information system to achieve the intended results.

Controls: Standards and guidelines for determining whether activities occur in the anticipated or accepted manner, that is, "under control". It also specifies actions to take when problems or unexpected circumstances are detected. It may include the reporting of exceptions or procedures for correcting problems.

Roles: The responsibilities of all persons involved with the new system, including end-users, computer operators, and support personnel are defined. The full spectrum of system components, including input of data to distribution of output or results are covered. Roles are often stated in the form of procedures.

3.2 Design of Output

Output refers to the results and information that are generated by the system. For many end-users, output determines the utility for developing the system and the basis on which they will evaluate the usefulness of the application. Most end-users will not actually operate the information system or enter data through workstations, but they will use the output generated by the system.

When designing output, system analysts must accomplish the following:

- Determine what information to present.
- Decide whether to display, print, or present the output information in audio form and select the medium accordingly.
- Arrange the presentation of information in an acceptable format.
- Decide the way of disturbing the output to intended recipients.

The arrangement of information on a display or printed document is termed as *layout*.

Accomplishing the general activities listed above will require specific decisions, such as whether to use preprinted forms when preparing reports and documents, how many lines to plan on a printed page, or whether to use graphics and color.

The output design is specified on layout forms, sheets that describe the location characteristics (such as length and type), and format of the column headings and page layout.

The goal of designing efficient and intelligible output design is to improve the system's relationship with the user and help in decision-making. A major form of output is a hard copy from the printer. Printouts should be designed around the output requirements of the user. The output devices depend on factors such as compatibility of the device with the system, response time requirements, expected print quality, and number of copies needed. The following media devices are available for providing computer-based output:

1. MICA readers.
2. Line, matrix, and daisy wheel printers.
3. Computer Output Microfilm (COM).

3.3 Design of Files

The designing files includes decisions about the nature and content of the file itself such as whether it is to be used for storing transaction details, historical data, or reference information. Some of the decisions made during file design are as following:

- The data items to be included in a record format within the file.
- Length of each record based on the characteristics of the data items on which it is based.
- The sequencing or arrangement of records within the file (the storage structure, such as sequential, indexed, or relative).

Not all new information system applications require the designing of all files used by them. For example, some master files may already exist because they are used in other existing applications. A new application may need to reference only the existing master file. In this instance, the details of the file are included in the application design specifications, but the file itself is not designed.

3.4 Design of Database Interactions

Many information systems interact with databases that span multiple applications. Because of the importance of databases to many systems, their design is established and monitored by a database administrator, who has the responsibility for developing and maintaining the database. In these instances, the system analyst does not design a database but rather consults the database administrator to determine the most appropriate way of interacting with the database.

The analyst provides the database administrator with the descriptions of

- Data needed from the database, and
- Actions that will affect the database (for example, retrieve data only, change the data values, or enter new data into the database).

The responsibilities of the database administrator are as follows:

a. Evaluate the appropriateness of the analysts' request.
b. Describe the methods for interaction with the database.
c. Ensure that the application cannot damage the database or adversely affect the needs of other information system applications in any way.

3.5 Design of Input

Systems analysts decide the following input design details:

- What data to input.
- What medium to use.
- How the data should be arranged or coded.
- The dialogue to guide users in providing input.
- Data items and transactions needing validation to detect errors.
- Methods for performing input validation and steps to follow when errors occur.

The design decisions for handling input specify how data are accepted for computer processing. Analysts decide whether the data are entered directly, perhaps through a workstation, or by using source documents such as sales slips, bank checks, or invoices where the data are transferred into the computer for processing.

The design of input also includes specifying the means by which end-users and system operators direct the system with regard to acceptance of input, production of a report, or end processing.

Online systems include a *dialogue* or *conversation* between the user and the system. Through dialogue, users request system services and tell the system when to perform a certain function. The nature of online conversation often makes the difference between a successful and unacceptable design. For instance, a blank display screen would reflect poor design and result in ambiguity regarding the follow-up action and also confuse a user about the action that is to be taken next.

The arrangement of messages and comments in *online conversations*, as well as the placement of data, headings, and titles on display screens or source documents, is also part of input design. Sketches of each are generally prepared to communicate the arrangement to users for their review, and to programmers and other members of the systems design team.

The goal of designing input data is to make data entry as easy, logical, and free from errors as possible. In entering data, operators need to know the following:

- The allocated space for each field.
- Field sequence, which must match the field sequence in the source document.
- The format in which data fields are entered; for example, entry of data in the date field is in the specified format say mm/dd/yy.

3.6 Design of Control

The systems analyst must assume that mistakes will be made in entering data or in requesting the performance of certain functions. Some mistakes are very minor and inconsequential, but others can be so serious that they could result in destruction of data or improper use of the system. Even if there is only a slight chance that a serious error will occur, a good information system design will offer the means for detecting and handling the error.

Input controls provide ways to:

1. Ensure that only authorized users access the system.
2. Guarantee that transactions are acceptable.
3. Validate the data for accuracy.
4. Determine whether any necessary data have been omitted.

3.7 Design of Procedures

Procedures specify what tasks must be performed in using the system and who is responsible for carrying them out. Important procedures include:

Data Entry Procedures: Methods for capturing transaction data and entering it into the information system (for example, sequence for entering data items recorded on source documents).

Run-time Procedures: Steps and actions taken by system operators and, in some cases, end-users who are interacting with the system to achieve the desired results (for example: mounting disk packs or loading printers with preprinted forms).

Error-Handling Procedures: Actions to be taken when unexpected results occur (for example, an error occurs when the system attempts to read data from a file or the printer jams partway through a long print run).

Security and Backup Procedures: Actions to protect the system and its resources against damage (for example, when and how to make duplicate copies of master files or segments of a database).

These procedures will be written and formally described as part of the documentation for the system.

3.8 Design of Program Specifications

Program specifications also constitute a part of design activity. They describe how to transform the system design specifications – for output, input, files, processing, and so on – into computer software.

Designing computer software is important to ensure that,

- The actual programs produced perform all tasks and do so in the manner intended.
- The division of the software into modules permits testing and validation to make it sure that the procedures are correct.
- Future modifications can be made in an efficient manner and with a minimum disruption to the design of the system.

A particular software system will be designed just once, but it will be used repeatedly and will undergo changes as the needs of users change. The methods for developing the design and for specifying the details will vary depending on the practices followed in a particular organization.

4. SYSTEM FLOWCHARTS

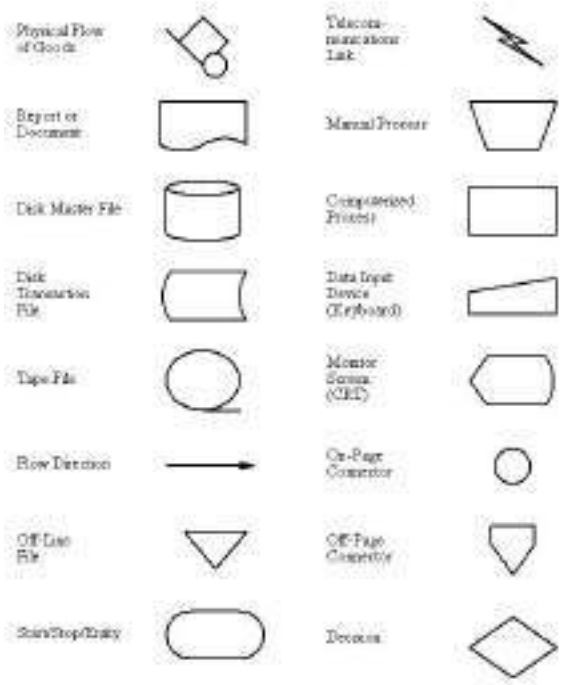
A system flowchart describes the data flow for a data processing system. It provides a logical diagram of the way the system operates. It represents the flow of documents and the operations performed in data processing systems. It also reflects the relationship between input and processing of data and generation of output. Following are the features of system flowcharts:

- The sources from which data is generated and the accessories used for generating data.
- Various processing steps involved.
- The intermediate and final output prepared and the devices used for their storage.

A systems flowchart is commonly used in analysis and design. Flowlines represent the sequences of processes, and other symbols represent the inputs and outputs to a process.

Below given are the Symbols used in System Flow Chart.

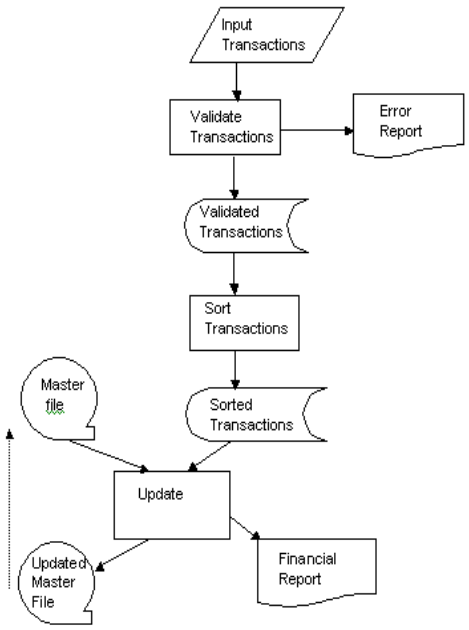
Figure 1: System Flowchart Symbols



The symbols are linked with directed lines (lines with arrows) showing the flow of data through the system.

An example of a system flowchart is shown in figure 2.

Figure 2: Example of a System Flowchart

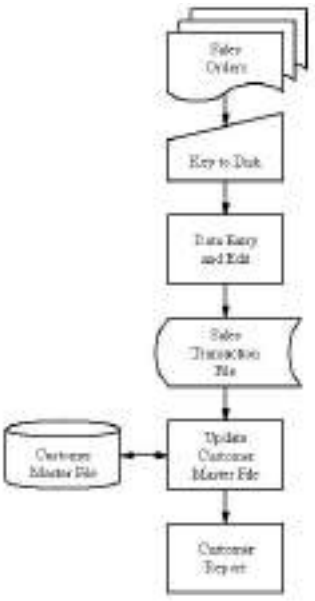


Transactions are input, validated and sorted, and then used to update a master file.

Note that the arrows show the flow of data through the system. The dotted line shows that the Updated master file is then used as input for the next Update process.

Another example of a system flowchart that represents Sales Transaction Processing is shown in figure 3.

Figure 3: An Example of System Flowchart



5. STRUCTURED FLOW CHARTS

Structured flow diagrams allow graphical representation of structured programs. Statement sequences are specified vertically, selection is specified by horizontal choices, and repetition is specified by a repeat indicator. Program embedding is specified by chart reference, diagonal layout, or block interiors. Flow diagrams are often limited to one page with a single entrance and exit. Machine readable flow charts are simplified if control is specified separately from statement definition.

Structured Flow Chart is used as a tool for Software Design and Documentation. Nassi and Shneiderman published a new flowcharting language with a structure closely similar to that of structured code. They provided a structure that can be retained by programmers who develop the application software. They are also called as Nassi – Shneiderman charts. The advantages of these charts are as follows:

i. The scope of iteration is well-defined and visible.

ii. The scope of IF THEN ELSE clauses is well-defined and visible; moreover, the conditions or process boxes embedded within compound conditions can be seen easily from the diagram.

iii. The scope of local and global variables is immediately obvious.

iv. Arbitrary transfers of control are impossible.

v. Complete thought structures can and should fit on no more than one page (i.e., no off-page connectors).

vi. Recursion has a trivial representation.

vii. These charts are adaptable to the peculiarities of the system or language they are used with.

By combining and nesting the basic structures, all of which are rectangular, a programmer can design a structured branch-free program.

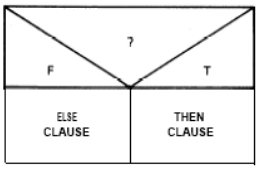
5.1 Basic Elements

There are three basic elements that are used in developing structured charts. They are:

i. **Process:** The basic process symbol is a rectangle representing assignments, calls, input/output statements, or any other sequential operations. In addition, a process symbol may contain other symbols nested within it. A name or brief description written in the box states the purpose of the process.

ii. **Decision:** The decision symbol represents alternative conditions that are possible and that program must handle them in a particular manner. They show an equivalent of the IF-THEN-ELSE structures in the Structured English. It may show actions for more than two alternatives at the same time. The symbol used to represent a decision is given below.

Figure 4: Decision symbol



This IF-THEN-ELSE symbol contains the test or decision in the upper triangle and the possible outcomes of the test in the lower triangles. “Yes” and “No” may be substituted for “True” and “False” and there is no particular objection to switching them right and left, although consistency is desirable. The rectangles contain the functions to be executed for each of the outcomes. Notice that the ELSE and THEN clause boxes are actually process symbols and may contain any valid process statements or nested structures.

iii. **Iteration:** Repeating processes i.e., *while* a certain condition exits or *until* a condition exits, are represented by an iteration symbol. It shows the scope of the iteration, including all processes and decisions that are contained within the loop. One of the three symbols may be used depending on whether loop termination is at the beginning or at the end of the loop.

Figure 5 shows a DO-WHILE symbol used for loops that test a condition at the beginning.

Figure 5: DO-WHILE Symbol

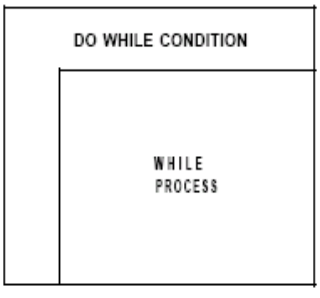


Figure 6 shows a DO-UNTIL symbol used for loops that test condition for the termination of the loop at the end.

Figure 6: DO-UNTIL Symbol

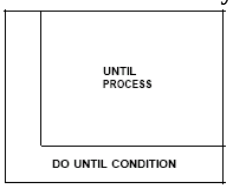
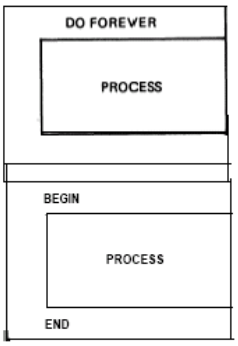


Figure 7 is a combination for loops with compound tests and may also be used for special constructs such as DO-FOREVER or for setting off BEGIN/END blocks.

Figure 7: DO-FOREVER Symbol



The CASE structure is represented by the symbol shown in figure 8. This form of CASE requires the setting of a variable to an integer value, and the choice of path is based on the value of that variable.

Figure 8: CASE Structure Symbol

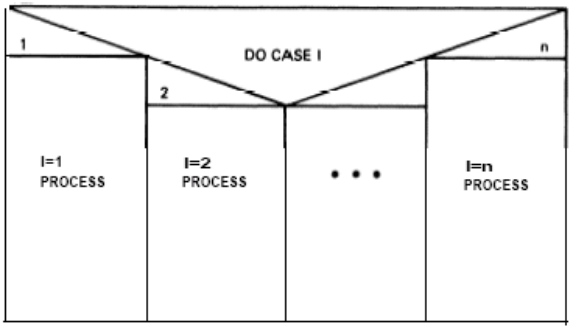
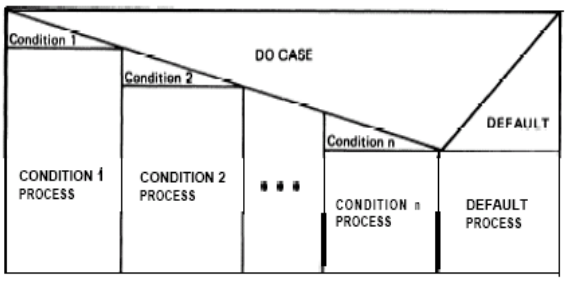


Figure 9 depicts a more powerful form of CASE, but one that requires the designer to be certain that the conditions chosen are mutually exclusive and cover all necessary condition testing.

Figure 9



Nesting of structures to create programs should now be an obvious extension of the use of basic symbols.

Figure 10 shows a structured flow chart to calculate and print an FICA report in a style useful to designers.

Figure 10: An Example of Structured Flow Chart

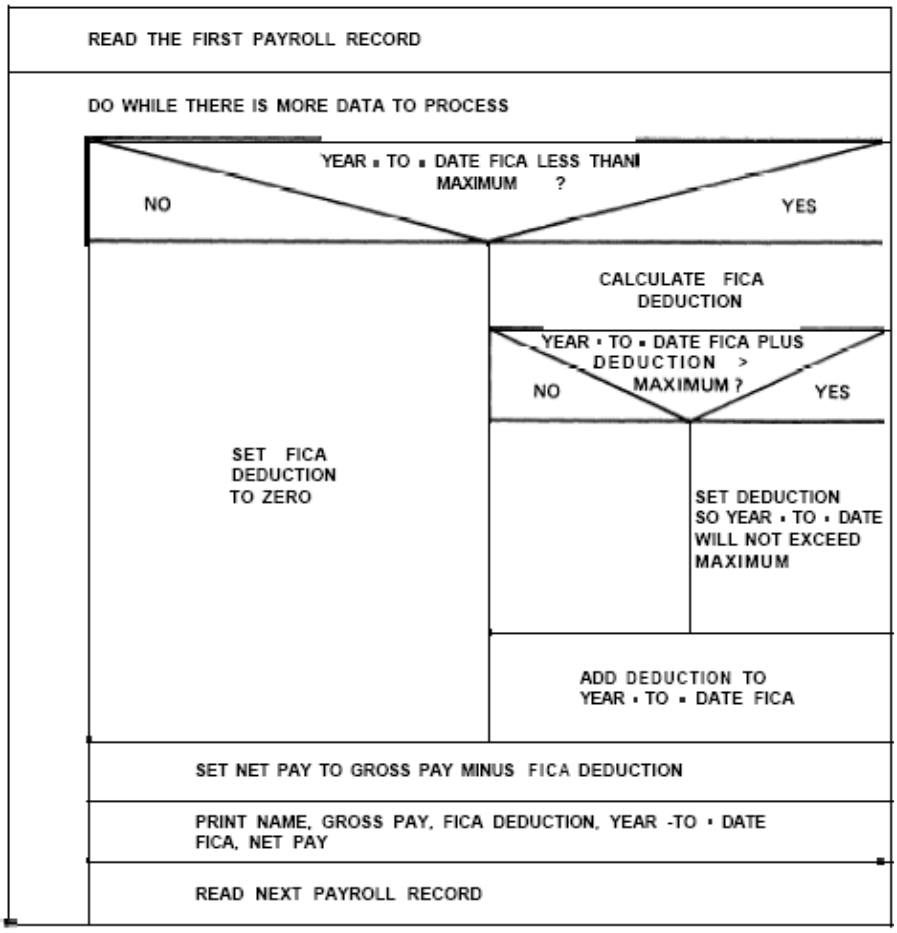
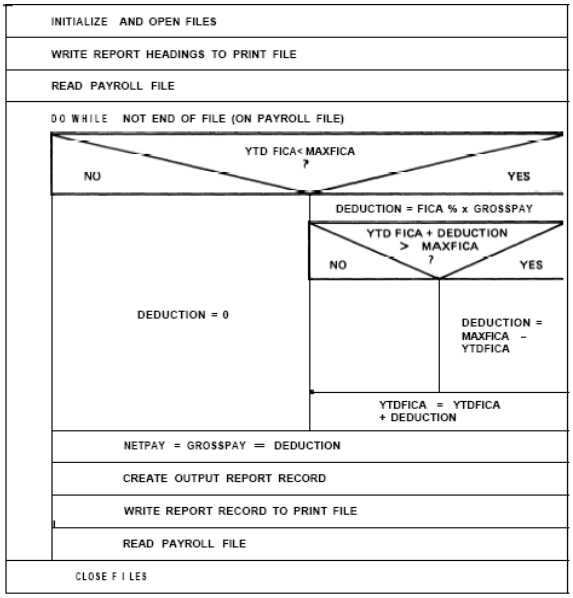


Figure 11 shows the same chart written in a style closer to the programming language, that is used by the programmers.

Figure 11



6. DATABASE DESIGN

File and database design occur in two steps. Logical and physical design activities are undertaken in parallel along with other systems design activities. We collect the detailed specifications of data necessary for logical database design. This design is driven not only from the previously developed E-R data model for the applications but also from form and report layouts. The designs for logical databases and system inputs and outputs are then used in physical design activities to specify computer programmers, database administrators, and others the way of implementing new information system.

6.1 Process of Database Design

In this section, we will discuss methods of developing logical and physical database designs during the system design phase.

6.1.1 LOGICAL DESIGN

It is based on the conceptual data model. Four key steps that are used in logical database design are:

i. Develop a logical data model for each known user interface for the application using normalization principles.

ii. Combine normalized data requirements from all user interfaces into one consolidated logical database model.

iii. Translate the conceptual E-R data model for the application into normalized data requirements.

iv. Compare the consolidated logical database design with the translated E-R model and produce one final logical database model for the application.

6.1.2 PHYSICAL DESIGN

It is based on the results of logical database design. The key decisions inclded:

a. Choosing storage format for each attribute from the logical database model.

b. Grouping attributes from the logical database model so as to form physical records.

c. Arranging related records in secondary memory (hard disks and magnetic tapes) so that records can be stored, retrieved and updated rapidly.

d. Selecting media and structures for storing data to make access more efficient.

6.1.3 DELIVERABLES AND OUTCOME

- Logical database design must account for every data element, system input or output.
- Normalized relations are the primary deliverables.
- Physical database design results in converting relations into files.

6.2 Relational Database Model

In this model, data is represented as a set of related tables or relations. Relation is a named, two-dimensional table of data. Each relation consists of a set of named columns and an arbitrary number of unnamed rows. The following table shows an example of relation named EMP. This relation contains the attributes: Emp-id, Name, Dept, Salary. There are five rows corresponding to a record that contains data values for an entity.

Emp-id	Name	Dept	Salary
10	Elias M.Awad	Info.systems	15000
25	James A.Senn	Accounting	21090
28	Valacich	Marketing	21000
35	Venu Gopal	Finance	34500
12	Rama Krishnan	Info systems	34200

We can express the structure of the relation by a shorthand notation in which the name of the relation is followed by the names of the attributes in the relation i.e.

EMP (Emp-id, Name, Dept, Salary)

Not all the tables are relations. Relations have several properties that distinguish them from non-relational tables:

- Entries in cells are simple.
- Entries in columns are from the same set of values.
- Each row is unique.
- The sequence of columns can be interchanged without changing the meaning or use of the relation.
- The rows may be interchanged or stored in any sequence.

6.3 Well-Structured Relation

A well-structured relation is that which contains a minimum amount of redundancy and allows users to insert, modify and delete the rows without errors or inconsistencies. The above defined relation (EMP) is a well-structured relation. Each row of the table contains data describing one employee, and any modification to an employee's data is confined to one row of the table.

6.4 Normalization

Normalization is the process of converting complex data structures into simple and stable data structures. It is based on well-accepted principles and rules. It results in the formation of tables that are related to each other. This group of tables constitute a database. There are actually four levels of normalization in Relational Database Management System (RDBMS). Each level reduces the complexity of the previous level and also the redundancy of data occurrence. We shall describe the First normal form, the Second normal form, the Third normal form and the Boyce-codd normal form with examples.

Suppose we have an entity customer and attributes like cusno, name and address are required. We can perceive that for a particular cusno, only one name and address are possible. Hence, the attributes name and address are said to be functionally dependent on the attribute cusno.

i. **First Normal Form:** A relational scheme is said to be in first normal form if only one value is associated with each attribute and the value is not a set of values or a list of values. A database scheme is in first normal form if every relational scheme included in the database scheme is in first normal form.

Example: Consider a table which is unnormalized as given below:

Cuscode	Shopname	Itemcode	Quantity
100	Sh1	112 17 18	2 4 3
121	Sh2	18 111	2 6
150	Sh3	112	1

In the first normal form, the above table appears as:

Cuscode	Shopname	Itemcode	Quantity
100	Sh1	112	2
100	Sh1	17	4
100	Sh1	18	3
121	Sh2	18	2
121	Sh2	111	6
150	Sh3	112	1

ii. **Second Normal Form:** A relation scheme **R**<**S, F**> is in second normal form if it is in the first normal form and if all non-prime attributes are fully functionally dependent on the relation key(s). A database scheme is in second normal form if every relation scheme included in the database scheme is in second normal form. For a table to be in the second normal form, it shall also be in first normal form and every attribute should functionally be dependant on the whole key.

Example: Consider the table given in the first normal form. In the second normal form, the redundancy of data observed in the table of first normal form can be overcome to some extent by constructing the following two tables.

Cuscode	Shopname
100	Sh1
121	Sh2
150	Sh3

Cuscode	Itemcode	Quantity
100	112	2
100	17	4
100	18	3
121	18	2
121	111	6
150	112	1

iii. **Third Normal Form:** A relation scheme **R**<**S, F**> is in third normal form if for all non-trivial functional dependencies in F of the form X→A, either X contains a key (i.e., X is a superkey) or A is a prime attribute. A database scheme is in third normal form if every relation scheme included in the database scheme is in the third normal form. For a table to be in the third normal form, it should be in the second normal form and each non-key attribute should functionally be dependent only on the primary key. In the third normal form relation, every non-prime attribute is non-transitively and fully dependent on the candidate key.

Example: Let us consider the table shown below:

Cuscode	Shopname	areacode
100	Sh1	A1
121	Sh2	A2
150	Sh3	A3
171	Sh4	A4
151	Sh5	A5
190	Sh6	A6

In the above table, the primary key is cuscode. The customer attribute is dependent on shopname. The attribute areacode is dependent on the shopname which shows an indirect dependence on the primary key. To prevent the problems faced during insertion, updation or deletion, the table given in the above example is split up into two tables given below:

Cuscode	Shopname
100	Sh1
121	Sh2
150	Sh3
170	Sh4
180	Sh5
190	Sh4

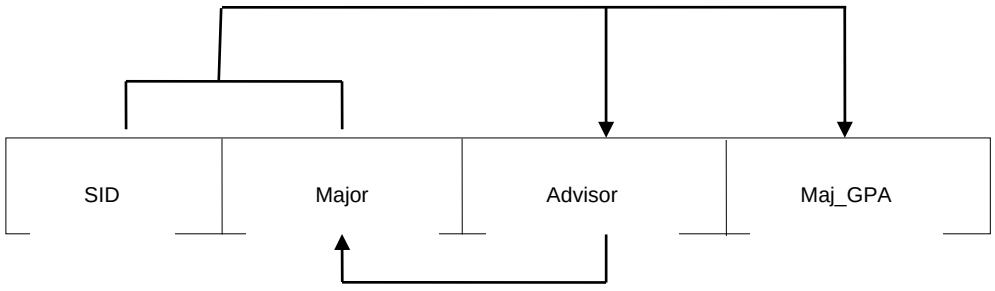
Shopname	Areacode
Sh1	A1
Sh2	A2
Sh3	A3
Sh4	A4
Sh5	A5

iv. **Boyce-Codd Normal Form:** A relation is in Boyce-Codd Normal Form (BCNF) if and only if every determinant in the relation is a candidate key. A determinant is an attribute, or a group on which some other attribute is fully functionally dependent. Boyce-Codd normal form is based on functional dependencies that take into account all candidate keys in a relation. To test whether a relation is in BCNF, we identify all the determinants and make sure that they are candidate keys. For example, consider the relation, STUDENT_ADVISOR as shown in Table given below:

STUDENT_ADVISOR

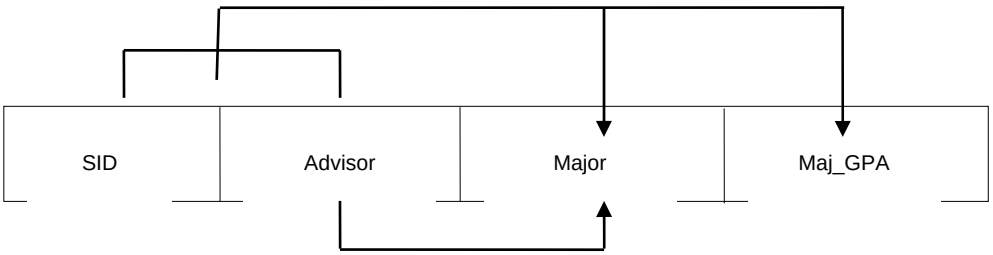
SID	Major	Advisor	Maj_GPA
123	Physics	Hawking	4.0
123	Music	Mahler	3.3
456	Literature	Michener	3.2
789	Music	Bach	3.7
678	Physics	Hawking	3.5

Functional Dependencies in STUDENT_ADVISOR



The above relation is in third normal form but not in BCNF. It can be converted into BCNF in two steps.

In the first-step, the relation is modified so that the determinant in the relation that is not a candidate key becomes a component of the primary key of the revised relation. The attribute that is functionally dependent on that determinant becomes a non-key attribute. This is a legitimate restructuring of the original relation because of the functional dependency. The result of applying this rule to STUDENT_ADVISOR is shown below:



The second step of the conversion process is to decompose the relation to eliminate the partial functional dependency. This results in two relations. The two relations (i.e. STUDENT and ADVISOR) with sample data are shown in the tables given below:

STUDENT

SID	Advisor	Maj_GPA
123	Hawking	4.0
123	Mahler	3.3
456	Michener	3.2
789	Bach	3.7
678	Hawking	3.5

Advisor

Advisor	Maj_GPA
Hawking	Physics
Mahler	Music
Michener	Literature
Bach	Music
Hawking	Physics

The result of normalization is that every non-primary key attribute depends upon the whole primary key.

6.5 Physical File and Database Design

The following information is required for designing physical files and databases:

- Normalized relations including volume estimates.
- Definitions of each attribute.
- Descriptions of where and when data are used, entered, retrieved, deleted and updated (including frequencies).
- Expectations or requirements for response time and data integrity.
- Descriptions of the technologies used for implementing the files and database.

Normalized relations are the result of logical database design. Statistics on the number of rows in each table as well as other information have to be collected during requirements determination in systems analysis.

Let us consider various aspects of database design.

6.5.1 DESIGNING FIELDS

A Field is the smallest unit of named application data recognized by system software. Each attribute from each relation will be represented as one or more fields. For example, student name attribute in a normalized student relation might be expressed as first name, middle name and last name.

6.5.2 CHOOSING DATA TYPES

Data Type is a coding scheme recognized by system software for representing organizational data. Four objectives of the data type are:

1. Minimize storage space.
2. Represent all possible values for the field.
3. Improve data integrity for the field.
4. Support all data manipulations desired on the field.

6.5.3 METHODS OF CONTROLLING DATA INTEGRITY

- ***Default Value:*** A value that a field will assume unless an explicit value is entered for that field.
- ***Picture Control (or Template):*** A pattern of codes that restricts the width and possible values for each position of a field.
- ***Range Control:*** Limits range of values which can be entered into field.
- ***Referential Integrity:*** An integrity constraint specifying that the value (or existence) of an attribute in one relation depends on the value (or existence) of the same attribute in another relation.
- ***Null Value:*** A special field value, distinct from 0, blank or any other value, which indicates that the value for the field is missing or otherwise unknown.

6.6 Designing Physical Tables

Relational database is a set of related tables. In logical database design, those attributes are grouped into a relation that concern some unifying, normalized business concept. In contrast, a **physical table** is a named set of rows and columns that specifies the fields in each row of the table.

6.7 Design Goals

i. Efficient use of secondary storage (disk space):
   - Disks are divided into units that can be read in one machine operation.
   - Space is used most efficiently when the physical length of a table row is compatible with storage unit.

ii. Efficient data processing:
   - Data are most efficiently processed when stored next to each other in secondary memory.

6.8 Denormalization

It is the process of splitting or combining normalized relations into physical tables based on affinity of use of rows and fields. It optimizes certain operations at the expense of others. Three common situations where denormalization may be used:

i. Two entities with a one-to-one relationship.

ii. A many-to-many relationship with nonkey attributes.

iii. Reference data.

6.8.1 ARRANGING TABLE ROWS

The result of the Denormalization is the definition of one or more physical files. A **physical file** is a named set of table rows stored in a contiguous section of secondary memory. Each table may represent a physical file or one file may represent a whole database. This depends on the database management software that is used.

7. FILE ORGANIZATION

File Organization is a technique for physically arranging the records of a file. Some objectives of file organization are:

- Fast data retrieval.
- High throughput for processing transactions.
- Efficient use of storage space.
- Protection from failures or data loss.
- Minimizing the effort for reorganization.
- Accommodating growth.
- Security from unauthorized use.

7.1 Types of File Organization

There are three commonly used file organization types. They are:

i. **Sequential File Organization:** In sequential file organization, the rows in the file are stored in sequence according to a primary key value. Updating and adding records may require rewriting the file. Deleting records results in wasted space. Sequential files are very fast when used to process rows sequentially, but they are essentially impractical for retrieving rows randomly. Only one sequence can be maintained without duplicating the rows.

ii. **Indexed File Organization:** In indexed file organization, the rows are stored either sequentially or non-sequentially and an index is created that allows software to locate individual rows. **Index** is a table used to determine the location of rows in a file that satisfy some condition. **Secondary Index** is an **i**ndex based upon a combination of fields for which more than one row may have same combination of values. Following are the guidelines for choosing indexes:

   - Specify a unique index for the primary key of each file.
   - Specify an index for foreign keys.
   - Specify an index for nonkey fields that are referenced in qualification, sorting and grouping commands for the purpose of retrieving data.

iii. **Hashed File Organization:** In hashed file organization, the address for each row is determined using an algorithm, which converts a primary key value into a row address. There are several variants of hashed files; the rows are located non-sequentially as dictated by the hashing algorithm. The retrieval of random rows is very fast.

7.2 Designing Controls for Files

The goals of the physical table are achieved primarily by implementing controls on each file. Two other important types of controls are file backup and security.

Backup Techniques include: Periodic backup of files, Transaction log or audit trail and Change log.

Data Security Techniques include: Coding or encrypting, User account management and Prohibiting users from working directly with the data. Users work with a copy which updates the files only after validation checks.

8. DESIGN OF COMPUTER OUTPUT

The term *output* implies any information produced by an information system in visual or printed form.

One of the most important aspects of an information system for users is the output it produces. Without quality output, the entire system may appear to be unnecessary and the users will avoid using it, possibly causing it to fail. In this section we will discuss designing of computer output and how to present of effective information.

When analysts design computer output, they,

- **Identify** the specific output that is needed to meet the information requirements.
- **Select** methods for presenting information.
- **Create** document, report or other formats that contain information produced by the system.

The methods of output vary across systems.

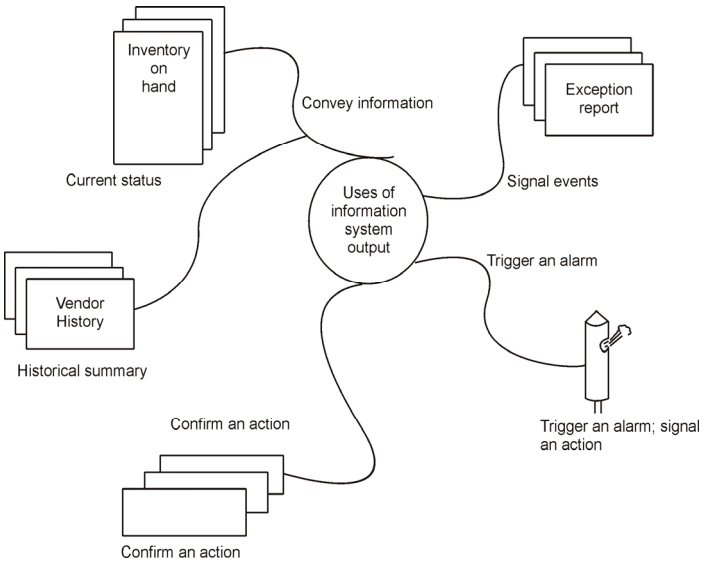
8.1 Output Objectives

The output from any system should accomplish one or more of the following objectives:

- **Convey** information about past activities, current status, or projections for the future.
- **Signal** important events, opportunities, problems, or warnings.
- **Trigger** an action.
- **Confirm** an action.

The objectives of using computer output are shown in figure 12.

Figure 12: Different Uses of Computer Output



The design itself begins when the systems analyst identifies the output the system must produce. In the structured analysis development method, DFDs prepared earlier in the development process determine the nature of the output needed. Each data flow carries information that is either used elsewhere in the system or that leaves the system as external output.

8.2 Types of Output

System output may be one or more of the following:

- A report.
- A document.
- A message.

Depending on the circumstances and the contents, the output may be displayed or printed. Output originates from these sources:

i. Retrieval from a data store.

ii. Transmission from a process or system activity.

iii. Directly from an input source.

The following media devices are available for providing computer based output:

i. MICR readers

ii. Line, matrix and daisy wheel printers

iii. COM (computer output microfilm)

iv. CRT screen display

v. Graph plotters

vi. Audio responses.

A system analyst not only selects the output devices but also prepares the format for editing and generating the final prinout. The standards for printed output are as follows:

i. Give each output a specific name or title.

ii. Provide a sample of the output layout, including areas where printing may appear and the location of each field.

iii. State whether each output field is to include significant zeros, spaces between fields, alphabets or any other data.

iv. Specify the procedures for proving the accuracy of output data.

In online applications, information is displayed on the screen. The layout sheet for displayed output is similar to the layout chart used for designing input.

9. DESIGN OF INPUT

Inaccurate input data are the most common cause of errors in data processing. Errors entered by data entry operators can be controlled by input design. Input design is the process of converting user-originated inputs to a computer-based format. To decide about the inputs required for a system, several questions need to be addressed. Some of them are:

- What data needs to be entered into the computer system?
- How much data needs to be input and how often?
- Where does the data come from?
- How will the data be entered into the system?

9.1 Input Data

The goal of designing input data is to make data entry as easy, logical and free from errors as possible. While entering data, operators need to know the following:

a. The allocated space for each field.

b. Field sequence, which must match that in the source document.

c. The format in which data fields are entered.

In input design, we design the source documents that capture the data and then Select the media to send input data into the computer.

9.2 Source Document

Source data are captured initially on a paper or as a source document. Source documents may be entered into the system from punched cards, from diskettes or even directly through the keyboard. A source document may or may not be retained in the candidate system. A source document should be logical and easy to understand. Each area in the form should be clearly identified and should specify the user what to write and where to write it.

9.3 Input Media and Devices

Source data are input into the system in a variety of ways:

- *Punch Cards* are either 80 or 90 columns wide. Data are arranged in a sequential and logical order. Operators use a keypunch to copy data from source documents onto cards. This means that the source document and card design must be considered simultaneously.
- *Key-to-diskette* is modeled after the keypunch process. A diskette replaces the card and stores up to 325000 characters of data (it is equivalent to the data stored in 4500 cards). Like cards, data on diskettes are stored in sequences and in batches. The approach to source document and diskette design is similar to that of the punched card.
- Magnetic Ink Character Recognition (MICR) translates the special fonts printed in magnetic ink on cheques into direct computer input. Magnetic printing is used so that the characters can be reliably read into a system.
- *Mark-sensing* readers automatically convert marks in predetermined locations on a card to punched holes on the same card.
- *Optical Character Recognition (OCR)* readers are similar to MICR readers, except that they recognize pencil, ink or characters by their configuration rather than their magnetic pattern.
- *Optical bar* readers detect a combination of marks that represents data.
- *Cathode-Ray Tube (CRT)* screens are used for online data entry. CRT screens display 20, 40 or 80 characters simultaneously on a television-like screen. They show as many as 24 lines of data.

In addition to determining record media, the analyst must decide on the method of input and the speed of capture and entry into the system. Processing may be,

1. Sequential.
2. Batched.
3. Random.

The following table shows the type of processing carried out by different input devices.

Input Device	Type of Processing
Key punch/punch card	Batch, sequential
Key-to-diskette	Batch, sequential (or random)
MICR reader	Batch, sequential (or random)
Mark-sensing reader	Batch, sequential (or random)
OCR reader	Batch, sequential (or random)
Optical bar code	Batch, sequential (or random)
Online data entry	Online, sequential (or random)

9.4 Online Data Entry

Online data entry makes use of a processor that accepts commands and data from the operator through a keyboard or a device such as a touch-sensitive screen or voice input. The input received is analyzed by the processor. It is then accepted or rejected, or further input is requested. The request for input is in the form of a message displayed on the screen or conveyed in audio form.

There are three major approaches for entering data into the computer:

a. Menus.

b. Formatted Forms.

c. Prompts.

Let us study about these approaches in detail:

9.4.1 MENU

A menu is a selection list that simplifies computer data access or entry. Instead of remembering what to enter, the user chooses a list of options and types the option letter associated with it. The example given below shows the list of matrix operations in menu form. A cursor blinking in the space reserved for ( ) ENTER CHOICE requests the user to type the number that represents the option.

```
        Matrix Operations

    (1) Addition

    (2) Subtraction

    (3) Multiplication

    (4) Transpose

ENTER CHOICE  (  )
```

A menu limits a user's choice of response but reduces the chances of error in data entry.

9.4.2 FORMATTED FORM

A formatted form is a preprinted form or a template that requests the user to enter data in appropriate locations. It is a fill-in-the-blank type form. The form is displayed on the screen as a unit. The cursor is usually positioned at the first blank. The user enters his response one line after another until the form is completed. During this routine, the user may move the cursor up, down, right or left to various locations for making changes in the response. The following example shows the formatted form for entering student details:

Student Details

Students Name ----------

Father's Name ----------

ICFAI CET Rank ----------

Degree with Branch Name ----------

Marks Secured ----------

Contact Number 1. ----------

2. ----------

Address ----------

9.4.3 PROMPT

In this approach, the system displays one enquiry at a time, asking the user for a response. The following dialogue represents a typical interaction between the system and the user at an ATM.

System	:	Please Insert ATM Card
User	:	inserts
System	:	Enter PIN
User	:	**** (User types password

Most systems edit the data entered by the user. If password is not matching, the system responds with a message like "INVALID NUMBER". The user has three chances to enter the correct password after which the system locks up.

The prompt method also allows the user to input questions and receive response of the system.

The main limitation with many of the available menus or prompts is that they require only one item to be entered at a time rather than a string of data items simultaneously.

9.5 CRT Screen Design

Many online data entry devices are CRT screens that provide instant visual verification of input data and a means of prompting the operator. The operator can make any changes desired before the data moves to the system for processing. A CRT screen is actually a display station that has a buffer for storing data. A common size display is 24 rows of 80 characters each for 1920 character.

10. USER INTERFACE DESIGN

User interface acts as a means of communication between a user (a human being) and a computer. A software engineer is responsible for the design of user interface by applying a set of iterative processes based on pre-defined design principles.

A good design of user interface is critical to the success of a software system. A good user interface allows people to work with the application without the need for receiving training or studying the manuals. Unless the software is easy to use, nobody will make an effort to use it. In the absence of good user interface, a user commits mistakes, gets frustrated and his efforts to accomplish the goal go in vain. Users will simply refuse to use the software system if the interface is difficult to use.

The design of user interface begins with the identification of user, task, and environmental requirements. After the identification of the user and tasks is over, user scenarios are created and analyzed in order to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphically designed icons and their placement, definition of descriptive screen text, specification and title for windows, and specification of major and minor menu items.

User interfaces can be textual or Graphical User Interface (GUI) based. GUIs are relatively easy to learn and use. Minimal amount of training is required since multiple screens are presented before the user and he can switch from one screen to another without losing sight of the original task he started with. User interface design is a iterative process. Various steps involved are:

- Analyzing the user Developing requirements and activities that are to be supported by the system.
- Developing screen-based designs that simulate user interaction.
- Evaluating the prototype with the help of the user.
- Feedback from the users is used to refine the user interface design.

A series of these steps finally result in the implementation of user interface.

10.1 User Interaction

Designers of the user interface should consider two issues – how information from the user is fed to the computer and how can information from the computer system be presented to the user. User interface must integrate user interaction and information presentation. Different forms of interaction are:

- **Direct Manipulation:** The user interacts with the objects on the screen. These interfaces have fast and intuitive interaction. They are quite easy to learn but hard to implement.
- **Menu Selection:** A command is selected from a list of possibilities. Little typing is required. But this interaction is a slow process for experienced users.
- **Form Fill-in:** Simple data entry interface. A user fills in the fields of a form. Though this interface is easy to learn, it takes up a lot of screen space.
- **Command Language:** Special commands with associated parameters are used to access the services provided by the system. This type of interface is powerful and flexible. But it is hard to learn.
- **Natural Language:** The user issues a command in natural language. Such interfaces can be accessible to casual users. But systems that relay on natural language are unreliable.

The above-mentioned interaction styles may be combined and an application may include several such styles. For example, Microsoft Windows supports direct manipulation of icons, menu-based command selection and also form-based interfaces.

10.2 Information Presentation

Input information may be directly presented to the user or it may be presented graphically. The software requirement for information presentation will be separated from the information itself. The representation on the user's screen can be changed without affecting the information or the underlying computational system. This can be achieved by separating the presentation system from the data.

In model-view-controller approach, users can interact with each presentation using a style. The information is encapsulated into a model object. Each model view object has a number of separate view objects associated with it. The controller object handles user input and device interaction. A model represents numeric data, in a number of different views such as a histogram or a table. Information that does not change should be distinguished from dynamic information by using different presentation styles. If information does not change fast then textual representation is suitable. If data changes quickly then graphical representation should be used. When precise alphanumeric information is presented, graphics can be used to pick up the information from its background. Graphics can attract user's attention. Large amounts of information can be presented using abstract visualizations. Three-dimensional presentations are particularly effective in product visualizations.

10.3 User Support

User interface should always provide online help. It is the documentation provided with the system. User guidance covers three areas: the messages generated by the system in response to user actions, the online help system for ready reference, and the documentation provided with the system. The designing of messages should involve professional writers and graphic artists. Various factors that are considered during the designing of error messages or help text are:

- **Context:** The help system should be in the current context.
- **Experience:** Messages should be tailored to the users' skills as well as their experience.

- **Style:** Messages should be addressed in an active rather than passive mode.
- **Error Messages:** The background and experience of the users should be anticipated while designing error messages. When users commit some mistake, they need to have an understanding of that error while using error messages.

10.4 Help System Design

Help system helps the users to understand the error messages. Help systems have a complex network structure where each frame of information may refer to several other information frames. The text in the help system should be prepared with the help of application specialists.

10.5 User Documentation

System manuals provide detailed information about the system's use and should be designed such that different classes of system end-users can use it. Different types of manuals that cater to the needs of different levels of expertise of the users are:

- **Functional Description:** Briefly describes the services provided by the system.
- **Installation Document:** Describes the installation process of the system. It includes installation instructions and details of how to set up configuration-dependent files.
- **Introductory Manual:** Describes the normal usage of the system and also how to get started and the common system facilities used by the end-users.
- **Reference Manual:** Describes system facilities, a list of error messages, causes of these errors and recovery methods.
- **Administrator Manual:** It describes the repair mechanisms for hardware failure and methods of connecting new peripherals. It also gives information about the messages generated when the system interacts with other systems and the procedures to react to these messages.

10.6 The Golden Rules

There are three important principles (golden rules) given by Theo Mandel that guide the designing of effective user interfaces. These rules are: (i) Place the user in control, (ii) reduce the user's memory load, and (iii) make the interface consistent. In order to develop such an interface which stands true to these principles, an organized design process must be conducted.

i. **Place the User in Control:** A user basically needs a system that obeys his commands to the maximum possible extent by satisfying all his needs. The user wants to have complete mastery over the computer. The system should understand the needs of the user and satisfy those needs and make the task easier. The constraints and limitations introduced by the designer should simplify the implementation of the interface. A number of design principles, given by Mandel that allow the user to maintain control are given in this regard:

   - **Define interaction modes in a way that does not force a user into unnecessary or undesired actions.** The current state of the interface is an interaction mode because through an interface the system takes commands from the user. For instance, if the user wants to check spellings of the words typed in a word document, the word processing software enters the spell check mode. When the user finishes his task of checking the spellings the software should allow him to exit from the spell check mode to another mode as desired without much effort or loss of time.

- **Provide for flexible interaction.** There should be adequate choices provided to different users for interacting with the system based on the nature of work they wish to perform. For instance, the interaction with the software might be carried out using keyboard commands, mouse movements, a digitizer pen, or voice recognition commands. However, a given action can be performed only using a particular mode of interaction. For instance, a complex figure cannot be drawn using keyboard commands.
- **Allow user interaction to be interruptible and undoable.** Even when a user is performing a sequence of actions, the interface system should be flexible enough to allow him to do something interrupting the current task at hand. The user should also be able to undo any action previously performed.
- **Streamline interaction as skill levels advance and allow the interaction to be customized.** While interacting with the system, it is common for the users to do the same set of tasks repeatedly. Hence, the design of a "macro" mechanism enables the users to carry out a set of tasks that repeat themselves time and again. For instance, in MS-Word, a macro command can be assigned to a combination of keys on the keyboard and when this combination is used, the predefined task is performed.
- **Hide technical internals from the casual user.** The user interface should be able to hide technical details regarding operating systems, or file management functions. It should provide the user, the facility to carry out his tasks or applications without knowing any technical fundamentals or the internal mechanism of the machine.
- **Design for direct interaction with objects that appear on the screen.** The user gets a feeling of control over the system when he is able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the objects were a physical thing.

ii. **Reduce the User's Memory Load:** In a well-designed user interface, a user is required to remember very less (for instance, the commands) while interacting with the system. This will in turn reduce the number of errors committed by the user. In order to reduce the user's memory load, Mandel defines certain design principles which are given below:

- **Reduce demand on short-term memory.** User's involvement with complex tasks increases the burden on short-term memory. The interface should be designed to reduce the need to remember past actions and results. Visual cues enable users to recognize past actions, without making strenuous effort to recall them.
- **Establish meaningful defaults.** The user interface provides a number of facilities by default for the user to interact with the system. However, a user also has his own set of individual preferences and those should be given due recognition while designing the user interface. A reset option provided to the user would enable him to redefine the original default values.
- **Define shortcuts that are intuitive.** When mnemonics are used to accomplish a system function (for instance, in MS-Word, Ctrl + S is used to save the document), the mnemonic should convey the action in a way which is easy to remember. For instance, in Ctrl + S, S stands for SAVE.
- **The visual layout of the interface should be based on a real world metaphor.** The user interface is helpful if it uses metaphors to guide the user. For instance, an E-commerce site uses a trolley wherein users deposit the articles that they have selected to purchase. This enables the user to rely on well-understood visual cues, rather than memorizing a long sequence.

- **Disclose information in a progressive fashion.** Hierarchical organization of interface would help the user to navigate through the system in an organized manner. Information about a task, an object, or some behavior should be presented first at a high level of abstraction. The details should be presented after the user indicates his interest by clicking the hyperlinks with a mouse.

iii. **Make the Interface Consistent:** The presentation of information to the user and acquisition of requisite information for further processing by the interface should be in a consistent fashion. This implies that (i) all visual information is organized according to a design standard that is maintained throughout all screen displays, (ii) input mechanisms are constrained to a limited set that are used consistently throughout the application, and (iii) mechanisms for navigating from one to another are consistently defined and implemented.

Following are the set of design principles defined by Mandel to make the interface consistent:

- **Allow the user to put the current task into a meaningful context.** User interfaces need to provide indicators to the users in the form of window titles, graphic icons, consistent color coding etc., when users are implementing complex tasks with a number of screen images. These indicators help the users in knowing the context of the work at hand. In addition, indicators should also serve the additional purpose of helping the users in keeping track of their past and future course of action.
- **Maintain consistency across a family of applications.** A set of all the applications should implement the same design rules so that a consistency is maintained in all interactions.
- **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.** Once a particular standard of interaction with the system has been well understood and used by a large number of users, then the users want the same standard to be implemented in new user interfaces. For instance, Ctrl+P is the standard to print the word document. Thus, users do not need to learn and remember new ways for printing the document. A change of standards will cause confusion which should be avoided.

11. DESIGNING INTERFACES AND DIALOGUES

An Interface is the common boundary between the user and the computer system application. Dialogue is the user's way of interacting with the computer system and application. Interface and dialogue design focuses on the way information is provided to and captured from users. It also includes defining the manner in which humans and computers interact with one another for finding any information. A good human-computer interface provides a unifying structure for finding, viewing and invoking the different components of a system.

11.1 Purpose of Interface

Designing an interface will accomplish the following purposes:

- Tell the system what actions to take.
- Facilitate use of the system.
- Avoid user errors.

Systems analysts consider the interface as a window to the system, a view portion of the entire system's features. Users tend to view the interface window as the entire system.

11.2 Characteristics of Interface

The characteristics of an interface in online systems include the devices used to enter and retrieve data, the dialogue which prompts and directs users and the methods and patterns followed in the display of information:

Entry: The information provided by users to request an action that initiates a response from the system.

Response: A message, prompt, or processing activity that results when an entry is provided by the user.

Dialogue: Dialogue guides the interaction between the system and the user. The dialogue strategy determines what information is entered and how responses are made.

Common interface devices in online systems are keyboard, mouse, light pen, scanner, touch screen or voice.

11.3 The Process of Designing Interfaces and Dialogues

Designing of interfaces and dialogues is done primarily keeping in mind the needs of the user. Prototyping methodology is followed for collecting information, constructing a prototype, assessing usability and making refinements. This process takes place parallely with the form and report design process.

11.3.1 DELIVERABLES AND OUTCOMES

Creation of design specifications is the result of system interface and dialogue design. Design specifications contain four sections:

- Narrative.
- Sample Design.
- Testing and usability assessment.
- Dialogue Sequence.

11.4 Designing Interfaces

In this section we will discuss the designing of interface layouts, providing guidelines for structuring and controlling data entry fields, providing feedback, and designing online help.

11.4.1 DESIGNING LAYOUTS

Standard formats similar to paper-based forms and reports should be used for designing computerized forms, used for reporting information Screen navigation on data entry screens should be left-to-right, and top-to-bottom as on paper forms.

When designing layouts, flexibility and consistency are primary design goals. Users should be able to move freely between fields. Data should not be permanently saved until the user explicitly requests for it. Each key and command should be assigned to one function.

11.4.2 STRUCTURING DATA ENTRY

We should follow several guidelines when structuring data entry fields on a form. Some of them are:

- *Entry:* Never require data that are already online or that can be computed.
- *Defaults:* Always provide default values when appropriate.
- *Units:* Make clear the type of data units requested for entry.
- *Replacement:* Use character replacement when appropriate.
- *Captioning:* Always place a caption adjacent to fields.
- *Format:* Provide formatting examples.
- *Justify:* Automatically justify data entries.
- *Help:* Provide context-sensitive help when appropriate.

11.4.3 CONTROLLING DATA INPUT

One objective of interface design is to reduce data entry errors. The role of a systems analyst is to anticipate user errors and design features in the system's interfaces to avoid, detect, and correct data entry mistakes.

Some types of data entry errors are:

- *Appending* – adding additional characters to a field.
- *Truncating* – losing characters from a field.
- *Transcripting* – entering invalid data onto a field.
- *Transposing* – reversing the sequence of one or more characters in a field.

The following techniques are used by system designers to detect errors:

- *Class or Composition* – test to ensure that data are of proper tpe.
- *Combinations* – test to see if the value combinations of two or more data fields are appropriate.
- *Expected Values* – test to see if the data are as expected.
- *Missing Data* – test for existence of data items in all fields of a record.
- *Pictures* – test to assure that data conform to a standard format.
- *Range* – test to assure data are within a proper range of values.
- *Reasonableness* – test to assure data are reasonable for situation.
- *Self-checking Digits* – test to see where an extra digit is added to a numeric field after its value is derived using a standard formula.
- *Size* – test for too few or too many characters.
- *Values* – test to make sure values come from a set of standard values.

11.4.4 PROVIDING FEEDBACK

While interacting with friends one expects feedback (response) from them in the form of say a nod and response to questions and comments. In the similar way when designing system interfaces, providing appropriate feedback is an easy way to make a user's interaction more enjoyable. System feedback consists of three types:

a. *Status Information:* Keeps users informed of what is going on in system. Displaying status information is especially important if the operation takes longer than a second or two.

b. *Prompting Cues:* Best to keep as specific as possible.

c. *Error and Warning Messages:* Messages should be specific and free of error codes and jargon. The user should be guided towards a result rather than scolded. Use terms familiar to the user. Consistency in format and placement of messages would be essential.

11.5 Designing Dialogues

Dialogue is the sequence in which information is displayed to and obtained from a user. One Primary design guideline is consistency in the sequence of actions, keystrokes, and terminology. Designing dialogues is a three-step process:

- Designing the dialogue sequence.
- Building a prototype.
- Assessing usability.

11.5.1 DESIGNING THE DIALOGUE SEQUENCE

The first step is defining the sequence. In other words, it is necessary to have a clear understanding of the user, task, and technological and environmental characteristics. After defining the sequence, transform the sequence into a formal dialogue specification. The method for designing and representing dialogues is **dialogue diagramming**. Dialogue diagrams have only one symbol, a box with three sections; each box represents one display within a dialogue. The three sections of the box are as follows:

1. **Top:** Unique display reference number used by other displays for referencing dialogue.
2. **Middle:** Contains the name or description of the display.
3. **Bottom:** Contains display reference numbers that can be accessed from the current display.

Figure 13: General Dialogue Diagram

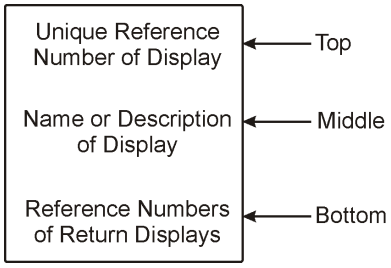
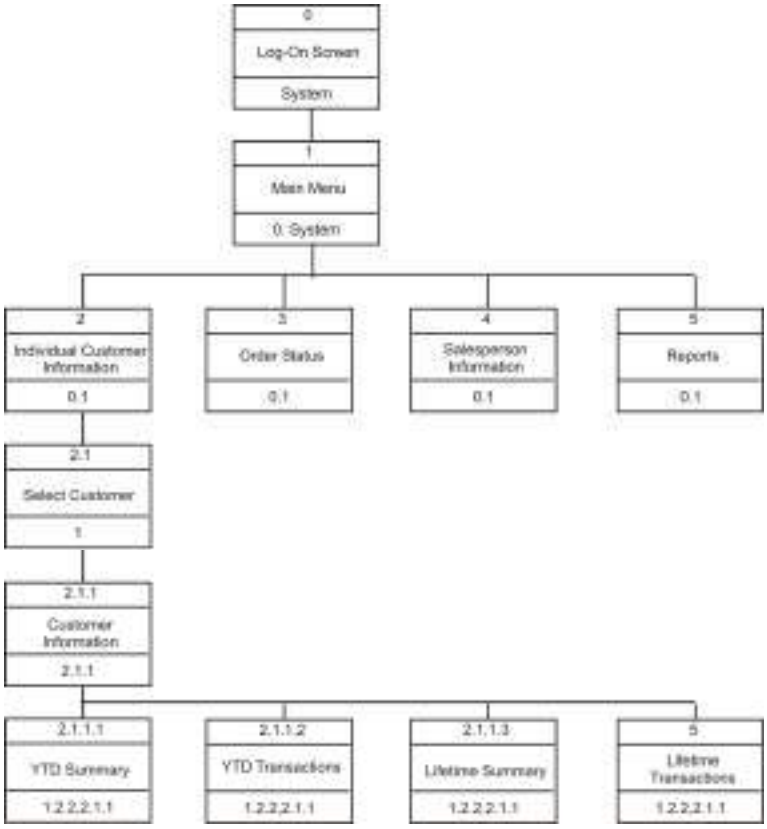


Figure 14: Dialogue Diagram for Customer Information System



12. COUPLING AND COHESION

Coupling refers to the degree of dependency among subsystems. It implies that the dependence between two subsystems is such that modifying one will have a strong impact on the other or one uses the services of the other or classes in one subsystem make use of objectives defined in the other.

Subsystems are **strongly coupled** to one another if there is a high degree of dependence between them. Subsystems should be **loosely coupled** if the impact of errors or changes in one subsystem is localized and has minimal effect on the rest of the system.

Cohesion refers to the strength of dependencies within a subsystem. Subsystems should be **highly cohesive**. All classes within the subsystem should have a single purpose.

The main goal of design (from a low-level perspective) is to minimize coupling and maximize cohesion. Coupling is the level of interdependency between a method and the environment (other methods, objects and classes), while cohesion is the level of uniformity of the method goal. While coupling needs a low-level perspective, cohesion needs a higher point of view. All patterns tend to reach these two objectives.

Coupling measures the strength of the relationships between modules. The modules should be loosely coupled i.e., independent, and interacting as little as possible. The looser the coupling, the easier it is to adapt the design.

Cohesion means each module should have one role. All parts of a module should contribute to this one role. It should not be made up of unrelated operations.

12.1 Types of Coupling

Following are the different types of couplings:

1. **Normal Coupling:** One module calls another with no parameter passing nor return values (i.e., no data communication), for example, clear Screen.
2. **Data Coupling:** Data is passed between modules. It can be achieved via parameter passing and/or return values.
3. **Stamp Coupling:** Unnecessary data passed between modules. For example, whole personnel record sent to calc-age module when only the date of birth is needed. Makes the called module do more than it needs to. Makes it less reusable in programs with different record structures.
4. **Control Coupling:** One module passes a piece of information intended to control the internal logic of another which may be data or a flag.
5. **Common (Global) Coupling:** Very undesirable. Communication would be via shared or global data. Suppose modules A, B and C each access some global data. Module A reads it and then invokes B, which alters it incorrectly. Later, C reads it, attempts to process it, fails, and the program crashes. Apparent cause is module C, actual cause is module B.
6. **Content (Pathological) Coupling:** The highest and worst degree of coupling, occurring when either one module makes use of data or control information held within another module, or one module branches into the middle of another.

12.2 Types of Cohesion

1. **Functional Cohesion:** This type of cohesion is considered the best. All the module's elements are necessary for the single, specific task Hence, a module contains all elements required for the task.
2. **Sequential Cohesion:** The elements are related by sequence. The output from one is the input for some other.
3. **Communicational Cohesion:** All of the elements in a module operate on the same input data or produce the same output data.
4. **Procedural Cohesion:** The elements make up a single control sequence; usually occurs if flowcharting has been used as a design technique.
5. **Temporal Cohesion:** The elements are all executed at the same time (for example, initialization or close down).
6. **Logical Cohesion:** The elements perform a set of logically related tasks (for example, different types of error handling).
7. **Coincidental Cohesion:** The weakest type of cohesion. No significant relationship between the elements of a module; they are simply bundled together by coincidence producing "scatterbrained" modules.

12.3 Features of a Good Design

Well-designed systems have:

- Modularity.
- Loose coupling between modules.
- High cohesion within modules.

In good systems, a module may be more easily:

- Tested, maintained, understood, and documented.
- Re-used in other systems/programs.
- Replaced by a more efficient implementation.

In addition, different modules can be developed, and tested in parallel with other modules by a team of software engineers, thus saving development time.

SUMMARY

- The objectives of information systems design are to provide specifications of the system blueprint, that is, the features of the system, which can then be translated into software for use in the organization. These specifications, called the logical system design, include the details of output, input files, database interaction, controls and procedures. Physical construction, which follows logical design, produces software, files and a working system.
- Other design objectives include ensuring that the system supports the activities of the business, meeting end-user requirements, engineering the system to be user-friendly, and providing detailed software specifications. All design features should conform to information systems standards established for the organization.
- When specifying for institutional systems, analysts design data flows, data stores, processes, procedures and controls. They also describe the roles of all the people who will be involved with the new system, such as end-users, computer operators and support personnel. System procedures often explain these individuals' roles. Analysts are also responsible for designing output, input and database interactions.

- Both the end-users and systems analysts share responsibility under the end-user method of development. To reduce risks to the organization, it is important to download data, avoid user data entry, follow design standards, document the system, and review all design specifications.
- System flowchart describes the data flow for a data processing system. It provides a logical diagram of how the system operates. It represents the flow of documents, and the operations performed in data processing system. It also reflects the relationship between inputs, processing and outputs.
- Structured flow diagrams allow graphical representations of structured programs. Statement sequences are specified vertically, selection is specified by horizontal choices, and repetition is specified by a repeat indicator. Structured Flow Chart is used as tool for Software Design and Documentation.
- Normalization is the process of converting complex data structures into simple, stable data structures. Denormalization is the process of splitting or combining normalized relations into physical tables based on affinity of use of rows and fields. It optimizes certain operations at the expense of others.
- File organization is a technique for physically arranging the records of a file. Three families of file organization are used in most file management environments. Those families are sequential file organization, indexed file organization and hashed file organization.
- In sequential file organization, the rows in the file are stored in sequence according to a primary key value. In indexed file organization, the rows are stored either sequentially or non-sequentially and an index is created that allows software to locate individual rows. Index is a table used to determine the location of rows in a file that satisfy some condition. In the hashed file organization, the address for each row is determined using an algorithm, which converts a primary key value into a row address.

Chapter IV

System Testing and Implementation

After reading this chapter, you will be conversant with:

- System Verification
- System Validation
- Software Testing
- Installing a System
- Training and Training Methods
- Conversions
- Post-Implementation Review
- System Audit

It is necessary that a software system meets specifications and also runs correctly. This would satisfy the processes of verification and validation. **Verification** is the process of determining if a system meets the conditions set forth at the beginning. **Validation** is the process of evaluating a system to determine whether it satisfies the specified requirements.

Today, there is an unprecedented growth in the development and use of automated tools and software aids for testing. One such tool is the functional tester, which determines whether the hardware is operating up to a minimal standard. It is a computer program that controls the complete hardware configuration and verifies that it is functional. For example, it can test computer memory by performing read/write tests, and it tests each of the peripheral device individually.

The functional tester is of great value when minute hardware problems are disguised as software bugs. For example, hardware faults are usually repeatable, whereas software bugs are generally erratic. Problems arise when the delicate interaction between hardware and software cause a hardware problem to appear as an erratic software bug. A functional tester determines immediately that the problem is in the hardware. This saves considerable time during testing.

Another software aid is the debug monitor. It is a computer program that regulates and modifies the application software that is being tested. It can also control the execution of functional tests and automatically patch or modify the application program being tested. The use of these tools will increase as systems grow in size and complexity and as the verification and insurance of reliable software become increasingly important.

1. SYSTEM VERIFICATION

There are three prominent means of minimizing the risks associated with the use of technology. They are: system verification, testing and maintenance. Every aspect of a computer system – hardware, communications and software, should be verified and thoroughly tested before the system is used for accomplishing an event. After successful testing, systems will need regular maintenance to ensure that they will perform effectively and also to enhance their working lives with addition of enhanced features.

The importance of technology in the working of a system will determine the degree of rigor applied to verifying, testing and maintaining the technology. For instance, a system to be used for a crucial electoral function, such as an electronic voting system, needs high degree of rigor. And for such a system, it is appropriate to employ an independent testing authority to perform system verification tests. For less important systems, system verification could be conducted in-house. System verification tests could include:

- Testing of hardware under conditions simulating expected real-life conditions.
- Testing of software to ensure that appropriate standards are followed and that the software performs its intended functions, including audits of code.
- Ensuring system documentation is adequate and complete.
- Ensuring data communications systems conform to appropriate standards and perform effectively.
- Verifying that systems are capable of performing under expected normal conditions and possible abnormal conditions.
- Ensuring appropriate security measures are in place and that they conform to appropriate standards.
- Ensuring that appropriate quality assurance measures are in place.

2. SYSTEM VALIDATION

Validation refers to the process of using software in a real-time environment in order to find errors. Validation of a computerized system means a documented verification that a specific computerized system performs according to its specifications. It means finding out whether a system performs its expected duties, and then this performance is documented.

There are many benefits of system validation which are given below:

- Compliance with regulatory requirements.
- Minimized risk of malfunction.
- Reduction in the cost of continuous operation.
- Increased knowledge of processes through improved system knowledge.
- Greater trust in the computerized system.

The feedback from the validation phase is analyzed. Changes are made in the software accordingly to deal with errors and failures that are uncovered. Validation may continue for several months. During this phase, the failures that have occurred are noted and the software is changed accordingly.

3. SOFTWARE TESTING

The testing of large and complex systems is a highly difficult and expensive activity whose importance in software development and maintenance cannot be underestimated.

Software testing is taken up after writing the source code. In software testing, a system or application is operated under controlled conditions and results are evaluated. The controlled conditions should include both normal and abnormal conditions. Even though experienced programmers take great care to write the source code which logically finds solution to the problem at hand, there is no guarantee that the source code, when executed by the system gives the correct result. Therefore, it is necessary to test the source code with sample data so as to be satisfied that the stated objectives have been met. Software testing is the process of executing a software system to determine whether it matches its specifications and executes in its intended environment. It can also be stated as the process of identifying defects, where a defect is any variance between actual and expected results. Testing commonly means executing software and finding errors. The process of software testing is used to identify the correctness, completeness and quality of developed computer software. There are a number of different testing approaches used to accomplish this, and they range from the most informal *ad hoc* testing to formally specified and controlled methods such as automated testing. Software testing is a very important step before the software project is released to the customer. Testing uncovers a number of errors are (logical errors, semantic errors, and run time errors). The uncovered errors are corrected before the product is delivered to the customer. When errors are discovered, they are eliminated through a process called debugging.

Testing performs a very critical role in ensuring quality assurance and thereby the reliability of software. Testing is the phase where the errors from all the previous phases must be detected. During testing, the program to be tested is executed with a set of test cases and the output of the program is evaluated to determine if the program is performing on expected lines. Customer satisfaction is very important to increase the customer base and win their faith. Therefore, programmers working for a software development company must execute the program before it is delivered to the customer with the sole intention of unravelling and removing all the errors. If the task of software testing is conducted systematically with the help of test cases designed using disciplined techniques, then there is a high probability of finding a large number of errors. It is necessary for software engineers to condition their minds psychologically before undertaking the testing phase.

This is because software engineers, however experienced they may be, should not have preconceived notions regarding the 'correctness' of software that they have built through analytical thinking and logical mind.

Testing rules as given by Glen Myers can very well serve as testing objectives; they are given below:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet-undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

The above mentioned objectives when applied successfully, will uncover errors in the software. In addition, testing also brings to light the fact that the software functions appear to be working according to specification. It is necessary that to meet the above objectives, data collected must be reliable.

Software testing is classified according to the manner in which testers perform the first two phases of the testing process. The scope of the first phase, modeling the software's environment, determines whether the tester is doing unit, integration, or system testing. The process of system testing has several steps to validate and prepare a system for final implementation. The different types of testing are:

i. **Unit Testing:** This tests individual software components or a collection of components. Testers define the input domain for the units under consideration and put aside the rest of the system. Unit testing sometimes requires the construction of throwaway driver code and stubs and is often performed in a debugger.

ii. **Integration Testing:** This tests multiple components that have each received prior and separate unit testing. In general, the focus is on the subset of the domain that represents communication between the components.

iii. **System Testing:** This tests a collection of components that constitutes a deliverable product. Usually, the entire domain must be considered to satisfy the criteria for a system test.

iv. **Positive Testing:** This is making sure that the new programs do process certain transactions according to specification.

v. **Acceptance Testing:** This is running the system with live data by the actual user.

Now, we shall study these tests in detail:

3.1 Unit Testing

Unit Testing is just one of the levels of testing in the entire gamut of testing a system. It complements integration and system level testing. It should also complement code reviews and walkthroughs. Unit testing is generally seen as a "white box" test class. That is, it is biased at looking and evaluating the code *as implemented*, rather than evaluating conformance to some set of *requirements*.

3.1.1 PURPOSE OF UNIT TESTING

Units are the smallest building blocks of software. In a language like C, individual functions make up the units. Unit testing is the process of validating such small building blocks of a complex system much before testing an integrated large module or the system as a whole. Some of the major benefits are:

- Ability to test parts of a project without waiting for the other parts to be available.
- Achieve parallelism in testing by being able to test and fix problems simultaneously by many engineers.
- Ability to detect and remove defects at a much less cost compared to testing at later stages.

- Ability to take advantage of a number of formal testing techniques available for unit testing.
- Simplify debugging by limiting to a small unit the possible code areas in which to search for bugs.
- Ability to test internal conditions that are not easily reached by external inputs in the larger integrated systems (for example, exception conditions not easily reached in normal operation).
- Ability to achieve a high level of structural coverage of the code.
- Avoid lengthy compile-build-debug cycles when debugging difficult problems.

3.2 Integration Testing

Integration testing can proceed in a number of different ways, which can be broadly characterized as ***top down*** or ***bottom up***. In ***top down integration testing,*** the high level control routines are tested first, possibly with the middle level control structures present only as ***stubs***. Subprogram ***stubs*** are incomplete subprograms which are only present to allow the higher level control routines to be tested. Thus, a menu driven program may have the major menu options initially only present as stubs, which merely announce that they have been successfully called, in order to allow the high level menu driver to be tested.

Top down testing can proceed in a ***depth-first*** or a ***breadth-first*** manner. For depth-first integration, each module is tested in increasing detail, replacing more and more levels of detail with actual code rather than stubs. Alternatively breadth-first would proceed by refining all the modules at the same level of control throughout the application. In practice a combination of the two techniques would be used. At the initial stages all the modules might be only partly functional, possibly being implemented only to deal with non-erroneous data. These would be tested in breadth-first manner, but over a period of time each would be replaced with successive refinements which were closer to the full functionality. This allows depth-first testing of a module to be performed simultaneously with breadth-first testing of all the modules.

The other major category of integration testing is ***bottom up integration testing*** where an individual module is tested from a group of test modules connected together. Once the individual modules have been tested, they are then combined into a collection of modules, known as ***builds***, which are then tested by a second test harness. This process can continue until the build consists of the entire application.

In practice, a combination of top-down and bottom-up testing is used. In a large software project being developed by a number of sub-teams, or a smaller project where different modules were being built by individuals, the sub-teams or individuals would conduct bottom-up testing of the modules which they were constructing before releasing them to an integration team which would assemble them together for top-down testing.

3.3 System Testing

The term System Testing can be used in a number of ways. In a general sense, the term ‘system testing’ refers to the testing of the system in artificial conditions to ensure that it performs as expected and as required. System testing is the type of black-box testing that is based on overall requirements specifications. It does not require the knowledge of the inner design of the code or logic. It attempts to discover defects that are properties of the entire system rather than of its individual components. System testing is conducted by taking into account the complete and integrated system so as to evaluate whether the system is compliant with its prestated requirements. It covers all the parts of a system.

System testing is used specifically to test the behaviors and bugs that manifest the entire system as distinct from properties attributable to components. System testing examines performance, throughput, security, recovery, resource loss, transaction

synchronization etc. Since this testing takes the integrated view of the entire system, it is performed after the system successfully completes integration testing and also after the integration of software with any hardware applications.

System testing can be used to assess the following characteristics of a system:

- Assignment and proper handling of interrupt priorities.
- Correct handling of the processing of each interrupt.
- Testing whether the performance of each interrupt-handling procedure is based on requirements.
- Testing whether the arrival of high volume of interrupts at critical times creates problems in function or performance.

In the system testing stage, the functional and the performance requirements of the system are checked. **System test** is the final validation step, and is performed once the whole system is developed. Before undertaking comprehensive system testing, we test the system in parts, according to a defined integration test strategy. The objective of a systematic test strategy during development is to detect design failures as early as possible so as to arrive at the system test phase with a mature, well-functioning system.

Therefore, the individual modules are first tested in isolation in order to detect any defects in their implementations at the earliest. This is the unit test stage. Many white-box and black-box methods exist for the selection of unit tests. Knowing that the single modules work well in isolation is not enough because, we want to test the interactions between them: we merge modules into progressively larger subsystems and select tests to detect possible problems in their interfaces and communications. In this process, there are two key considerations: how to select an effective set of test cases, and how to progressively combine modules in subsystems. Integration testing is mostly addressed with reference to the second issue. Systematic application of top-down, bottom-up, and several other mixed strategies is traditionally recommended, as opposed to chaotic, all-at-once, *big-bang* testing.

Validation testing is a concern which overlaps with integration testing. Ensuring that the application fulfils its specification is a major criterion for the construction of an integration test. Validation testing also overlaps to a large extent with ***system testing***, where the application is tested with respect to its typical working environment. Consequently, for many processes, no clear division between validation and system testing can be made. Specific tests which can be performed in either one or both stages include the following:

- ***Regression testing:*** Where the new version of the software is tested along with the previous versions to ensure that the required features of the previous version are compatible with the new version.
- ***Recovery testing:*** Where the software is deliberately interrupted in a number of ways, for example, taking its hard disc off line or even turning the computer off, to ensure that the appropriate techniques for restoring any lost data will function.
- ***Security testing:*** Where unauthorised attempts to operate the software, or parts of it, are undertaken. It might also include attempts to obtain access to data, or harm the software installation or even the system software. As with all types of security, it is recognised that someone sufficiently determined will be able to obtain unauthorised access and the best that can be achieved is to make this process as difficult as possible.
- ***Stress testing:*** Where abnormal demands are made upon the software by increasing the rate at which it is asked to accept data, or the rate at which it is asked to produce information. More complex tests may attempt to create very large data sets or cause the software to make excessive demands on the operating system.

- ***Performance testing:*** Where the performance requirements, if any, are checked. These may include the size of the software when installed, the amount of main memory and/or secondary storage it requires and the demands made of the operating system when running within normal limits or the response time.
- ***Usability testing:*** Even if usability prototypes have been tested along with the application being constructed, a validation test of the finished product will always be required.
- ***Alpha and beta testing:*** An initial release, the alpha release, might be made to select users who would be expected to report bugs and other detailed observations back to the production team. Once the application has passed through the alpha phase, a beta release, possibly incorporating changes necessitated by the alpha phase, can be made to a larger more representative set of users, before the final release is made to all users.

The final process would be the audit of software where the complete software project is checked to ensure that it meets production requirements. This ensures that all required documentation has been produced in the correct format and is of acceptable quality. The purpose of this review is: to assure the quality of the production process and also the product, and secondly to ensure that all is in order before the initial project construction phase concludes and the maintenance phase commences. A formal hand over from the development team at the end of the audit will mark the transition between the two phases.

3.4 Acceptance Testing

The specified conditions for user acceptance testing are:

i. **Planning for User Acceptance Testing:** In this activity the analyst and the user agree on the conditions for the test. Many of these conditions may be derived from the test plan. Other points of agreement include the test schedule, the test duration and the persons designated for the test. The start and termination dates for the test should also be specified in advance.

ii. **Prepare Test Data for Transaction Path Testing:** This activity develops the data required for testing every condition and transaction to be introduced into the system. The path of each transaction from origin to destination is carefully tested for reliable results. The test verifies that the test data are virtually comparable with real-time data used after conversion.

iii. **Plan User Training:** User training is designed to prepare the user for testing and converting the system. User involvement and training take place in parallel with programming for three reasons:

   - The system group has time available to spend on training while the programs are being written.
   - Initiating a user training program gives the systems group a clearer image of the users interest in the new system.
   - A trained user participates more effectively in systems testing.

For user training, preparation of a checklist is useful which includes the provisions for developing training materials and other documents to complete the training activity. In effect, the checklist calls for a commitment of personnel, facilities, and efforts for implementing the candidate system.

The training plan is followed by preparation of the user training manual and other text materials. Facility requirements and the necessary hardware are specified and documented. A common procedure is to train supervisors and department heads who in turn train their staff. The reasons are:

i. User supervisors are knowledgeable about the capabilities of their staff and the overall operation.

ii. Staff members usually respond more favorably and accept instructions better from supervisors than from outsiders.

iii. Familiarity of users with their particular problems (bugs) makes them better candidates for handling user training than the system analyst. The analyst gets feedback to ensure that proper training is provided.

iv. Operational systems are generally not cared properly. Every system requires periodic evaluation after implementation.

The second phase of testing which is test selection, determines what type of testing is being done. There are two main types:

i. **Functional testing** requires the selection of test scenarios without regard to source code structure. Thus, test selection methods and test data adequacy criteria, must be based on attributes of the specification or operational environment and not on attributes of the code or data structures. Functional testing is also called as specification-based testing, behavioral testing, and black-box testing.

ii. **Structural testing** requires that inputs be based solely on the structure of the source code or its data structures. Structural testing is also called code-based testing and white-box testing.

System testing plays an important role as testing is essential for the success of the system. System testing makes a logical assumption that if all the parts of the system are correct, the goal will be successfully achieved. Inadequate testing or non-testing leads to errors that may not appear apparently in near future. This creates two problems: (1) the time lag between the cause and the appearance of the problem (the longer the time interval, the more complicated the problem has become), and (2) the effect of system errors on files and records within the systems. A small system error can conceivably explode into a much larger problem. Early and effective testing in the process results in long-term cost savings due to reduced number of errors.

Another reason for system testing is its utility as a user-oriented vehicle before implementation. The best program is worthless if it does not meet user needs.

4. INSTALLING A SYSTEM

The organizational process of changing over from the current information system to a new one is called “Installation”. During installation process, the current system is replaced by the new system. Employees who use the current system, must adapt themselves and start using the new system that is being installed.

Deliverables and outcomes of installation are:

- User guides
- User training plans
- Installation and conversion plan.

Installation approaches are:

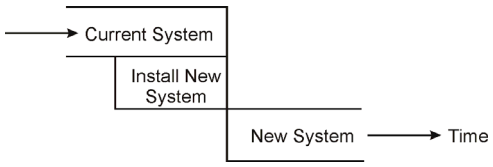
- Direct
- Parallel
- Single location (Pilot)
- Phased.

The details of these approaches are given below:

4.1 Direct Installation

Changing over from the old information system to a new one by turning "OFF" the old system when the new one is turned on. This installation is shown in the figure given below.

Figure 1



Advantages

- This may be the only possible approach if new and existing systems cannot coexist in some form.
- Operating cost is low.
- High interest in making installation a success.

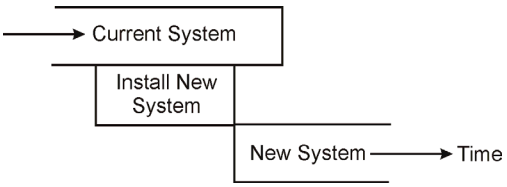
Disadvantages

- Operational errors have direct impact on users and organization.
- It may be too long to restore old system, if necessary.
- Time-consuming, and benefits may be delayed until the whole system is installed.

4.2 Parallel Installation

Running the old information system and the new one simultaneously until the management decides that the old system can be turned off. This installation is shown in figure 2.

Figure 2



Advantages

- New system can be checked against the old system.
- Impact of operational errors is minimized because the old system is also processing all data.

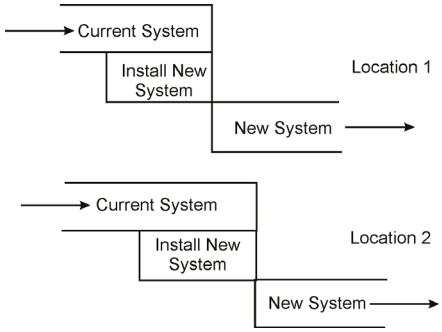
Disadvantages

- Not all aspects of the new system can be compared to the old system.
- Very expensive due to duplication of effort to run and maintain two systems.
- Can be confusing to users.
- May not be feasible due to costs or system size.

4.3 Single Location Installation (Pilot)

In this method of installation, an information system is installed at one site and is studied its performance to decide whether the new system should be deployed and the way of its deployment throughout the organization. This type of installation is shown in figure 3.

Figure 3



Advantages

- One can gain knowledge about the problems fixed by concentrating on one site.
- Limits potential harm and costs from system errors or failures to select pilot sites.
- If the installed system is successful, then it would be easy to convince others to convert to the new system.

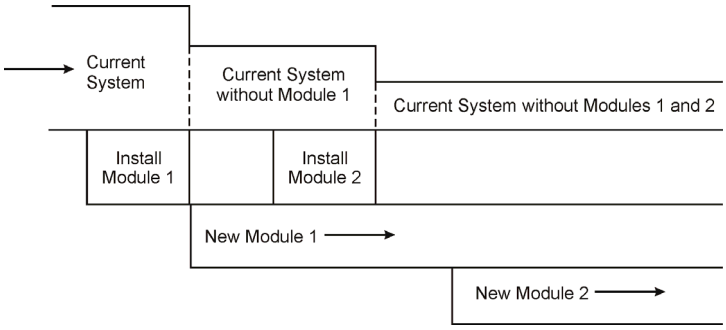
Disadvantages

- Burden on Information System (IS) staff to maintain old and new systems.
- If different sites require data sharing, extra programs need to be written to "bridge" the two systems.
- Some parts of the organization get benefits earlier than other parts.
- May give the impression that the old system is unreliable and error-prone.

4.4 Phased Installation

The phased installation is changing from the old information system to the new one incrementally, starting with one or a few functional components and then gradually extending the installation to cover the whole new system. This installation is shown in figure 4.

Figure 4



Advantages

- It would also make it possible to undertake system development in a phased manner.
- Limits potential harm and costs from system errors or failures to certain business activities/functions.
- Risks spread over time.
- Some benefits can be achieved early.
- Each phrase is small and more manageable.

Disadvantages

- Old and new systems must be able to work together and share data, which will likely require extra programming to "bridge" the two systems.
- Conversion is constant and may extend over a long period, causing frustration and confusion for users.

4.5 Planning Installation

Installation strategy involves converting not only software but also data and hardware, documentation, work methods, job descriptions, and other aspects of the system.

One of the special considerations is **data conversion**. Because existing systems usually contain data required by the new system, current data must be error-free, unloaded from the current files, combined with new data and loaded into new files. New data fields may have to be entered in large quantities so that every record copied from the current system has all the new fields populated. The total conversion process can be tedious. This process may require that current system be shut off while the data are extracted so that updates to old data, which would contaminate the extract process, can not occur.

Another consideration is **business cycle of organization**. Most organizations face heavy workloads at particular times of year and relatively light loads at other times.

Planning for installation may begin as soon as the analysis of the present system of an organization is completed. Some installation activities must be done after software installation can occur. The project leader is responsible for anticipating all installation tasks and assigns responsibility for each to different analysts.

5. TRAINING AND TRAINING METHODS

Well-designed and technically elegant systems may succeed or fail because of the way they are operated and used. The quality of training received by the personnel involved with the system in various capacities helps or hinders, and may even prevent, the successful implementation of an information system. Those who will be associated with or affected by the system must know in detail what their roles will be, how they can use the system, and what the system will or will not do. Both system operators and users need training.

5.1 Training Systems Operators

Many systems depend on the computer-center personnel, who are responsible for keeping the equipment running as well as for providing the necessary support service. Their training must ensure that they are able to handle all possible operations, both routine and extraordinary. Operator training must also involve the data entry personnel.

As part of their training, operators should be given both a troubleshooting list that identifies possible problems and remedies for them, as well as the names and telephone numbers of individuals to contact when unexpected or unusual problems arise.

Training also involves familiarization with *run procedures*, which involves working through the sequence of activities needed to use a new system on an ongoing basis. These procedures allow the computer operators to become familiar with the actions they need to take, and when these actions must occur. In addition, they find out how long applications will run under normal conditions. This information is important both to enable users to plan work activities and to identify systems that run longer or shorter than expected – a sign that typically indicates problems with the run.

5.2 User Training

User training may involve familiarizing with the usage of equipment particularly in the case where, say, a microcomputer is in use and the individual involved is both the operator and user. In these cases, users must be instructed first on how to operate the equipment. Questions that seem trivial to the analyst, such as how to turn the terminal "ON" how to insert a diskette into a microcomputer, or when it is safe to turn "OFF" equipment without the danger of data loss may seem out of the blue for new users who are not familiar with computers.

User training must also instruct individuals in troubleshooting the system, determining whether a problem that arises is caused by the equipment or software or by something they have done in using the system. Including a troubleshooting guide in systems documentation will provide a useful reference long after the training period is over. The time to prevent the frustration is during training.

Much user training deals with the operation of the system itself. Training in data coding emphasizes the methods to be followed in capturing data from transactions or preparing data needed for decision support activities.

Data-handling activities that receive much attention in user training are adding data (how to store new transactions), editing data (how to change previously stored data), formulating inquiries (finding specific records or getting responses to questions), and deleting records of data. The bulk of systems use involves this set of activities; so, it follows that most training time will be devoted to this area.

From time to time, users will have to prepare disks, load paper into printers, or change ribbons on printers. No training program is complete without some time devoted to systems maintenance activities. If a microcomputer or data entry system will use disks, users should be instructed in formatting and testing disks. They should also actually perform ribbon changes, equipment cleaning, and other routine maintenance. It is not enough to simply include this information in a manual, even though that is essential for later reference.

There are two aspects to user training:

- Familiarization with the processing system itself (that is, the equipment used for data entry or processing).
- Training in using the application (that is, the software that accepts the data, processes it, and produces the results).

Weaknesses in either aspect of training are likely to lead to awkward situations that produce user frustration, errors, or both. Good documentation, although essential, does not replace training. There is no substitute for hands-on operation of the system while learning its use.

5.3 Training Methods

The training of operators and users can be achieved in several different ways. Training activities may take place at vendor locations; or at rented facilities. The methods and content of the training often vary depending on the source and location of the training.

5.3.1 VENDOR AND IN-SERVICE TRAINING

Often the best source of training on equipment is the vendor supplying the equipment. Most vendors offer extensive educational programs as part of their services either for a fee or without any charges. The courses, for instance offered by experienced trainers and sales personnel, cover all aspects of using the equipment, from how to turn it on and off, to the storage and removal of data, to handling malfunctions. This training is hands-on; so, the participants actually use the system in the presence of the trainers. If questions arise, they can quickly be answered.

If a special software such as a teleprocessing package or database management system is being installed, sending personnel to off-site short-term courses and providing in-depth training is preferable to in-service training. These courses, which are generally provided for a fee, are presented to personnel from many organizations that are acquiring or using the same system. The benefit of sharing questions, problems, and experiences with persons from other companies is substantial. The personal contacts made during the sessions frequently last for years, with the continual sharing of information benefiting both parties.

5.3.2 IN-HOUSE TRAINING

The advantage of offering on-site training on the usage of the system is that the instruction can be tailored to the needs of the organization where it is being offered and proper focus can be given on special procedures used in that setting, and the organization's plans for growth, and any problems that may arise can be tackled in a practical way. Often, the vendors or training companies negotiate fees and charges that are more economical and that enable the organization to involve more personnel in the training program than is possible when travel is required.

There are also disadvantages. The mere fact that employees are in their own surroundings is a distraction, since telephone calls and emergencies can disrupt training sessions. Moreover, when outside firms come on-site, they may present courses that emphasize general concepts but that lack sufficient hands-on training. The training coordinator must recognize this possibility and deal with it in advance to ensure that the course content will meet operating needs.

In-house training can also be offered through specially purchased instructional materials. There is no substitute for hands-on experience. Training manuals are acceptable for familiarization, but the experiences of actually using the equipment, making and correcting mistakes, and encountering unexpected situations are the best and most lasting.

Training manuals generally take one or two approaches. Some have the user work through different activities step by step. The other common approach is to create a case-study example that includes all frequently encountered situations that the system is able to handle and that the users should be able to handle. Then, the users must use the system to handle the actual situations; that is, enter data as required, process the data, and prepare reports. If the system is inquiry-oriented, the case study should require the users to pose and receive responses to inquiries. Sample data and individual transactions are included, so individuals use the system and receive immediate feedback about the correctness of their actions. If the results they produce do not match those provided in the training guide, the users will know that mistakes were made.

During training, systems personnel should be alert to comments made by users or to problems that users may encounter. Although human factors are difficult to test some problems may not occur until inexperienced users are directly interacting with the system. Despite testing, awkward keying requirements to enter data, unexpected transactions, or unusual ways of preparing transactions may still arise during training. The trainer must to involve systems personnel when problems in the design are found, while assisting users who are reluctant to change from their old ways to the new methods. Of course, the trainer must first be certain that the new methods are necessary and do represent an improvement over current methods.

6. CONVERSION

Conversion involves moving over from one system to another. After conversion, a newly tested system is put to operation without causing costs, risks and personnel dissatisfaction to escalate. The conversion from one system to another involves (i) creating computer-compatible files, (ii) Training the operating staff, and (iii) installing terminals and hardware. It should be kept in mind that the conversion process should not disturb the normal operations of the entire organization. There may arise many problems during conversion process in the form of inadequate training of the employees, technical breakdown of the system,

damage to data files etc. The steps that arise before the culmination of conversion process are as follows:

i. The user feels an urge to go for conversion due to additional tasks.

ii. The user expresses his desire for conversion before the analyst and the analyst conducts a study for preparing a proposal.

iii. The proposal contains general specifications of software and hardware and also a vendor is selected and a date is set for conversion.

iv. The vendor or the project team undertakes the installation of the project developed according to the user's specification.

v. Inspite of delays or other impediments, the installation of new system happens with very little involvement of user.

vi. The training of employees is undertaken to reduce their resistance to the new system.

Given below is the list of activities that are part of the conversion process:

i. Conversion begins with a review of the project plan, the system test documentation, and the implementation plan. The people involved are the user, the project team, programmers and operators.

ii. The conversion part of the implementation plan is finalized and approved.

iii. Files are converted.

iv. Parallel processing between the existing and the new systems is initiated.

v. Results of computer runs and operations for the new system are logged on a special form.

vi. There would be no need to run the old system in parallel with the new one if there are no problems with the new system. The results emanating from the operation of the new system are recorded as documents for future reference.

vii. The conversion process is complete with the new system coming into formal operational stage. At last, plans are drawn up for post-implementation review.

7. POST-IMPLEMENTATION REVIEW

A post-implementation review measures the system's performance against predefined requirements. Unlike system testing, which determines the system failures a post-implementation review determines how well the system continues to meet performance specifications after the design and conversation are complete. It also provides information to determine whether major redesign is necessary.

Post-implementation review is the evaluation of a system in terms of the extent to which the system accomplishes stated objectives and the actual project costs exceeding initial estimates. It is usually the review of major problems that need converting and those that surfaced during the implementation phase. The primary responsibility for initiating the review lies with the user organization, which assigns the special staff for this purpose.

7.1 Request for Review

The initiating study begins with the review team, which gathers and reviews requests for evaluation. It also files discrepancy notices after the system has been accepted. Unexpected change in the system that affects the user or system performance is a primary factor that prompts system review. Once a request is filed, the user is asked how well the system is functioning to specifications or how well the measured benefits have been realized. Suggestions regarding changes and improvements are also sought. This phase sets the stage for a formal post-implementation review.

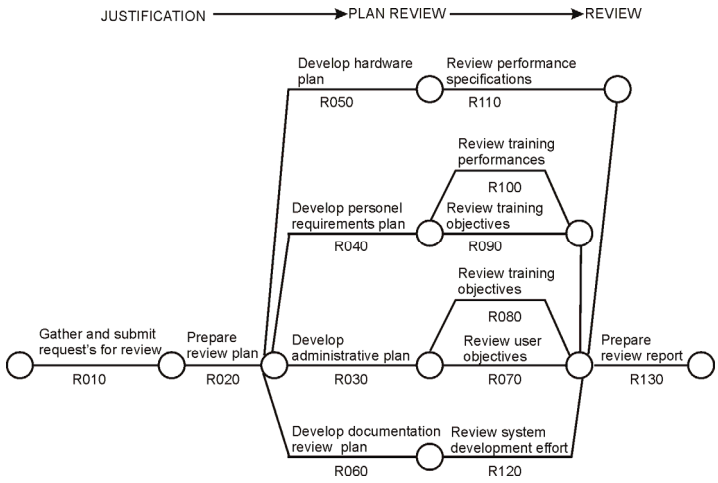
7.2 A Review Plan

The review team prepares a formal review plan around the objectives of the review, the type of evaluation to be carried out and the time schedule required. An overall plan covers the following areas:

- **Administration Plan:** It is to review area objectives, operating costs, actual operating performance and benefits.
- **Personnel Requirements Plan:** It is to review performance objectives and training performance till date.
- **Hardware Plan:** It is to Review performance specifications.
- **Documentation Review Plan:** It is to review the system development effort.

Once drafted, the review should be verified and approved by the requester or the end-user. The overall plan is shown in figure 5.

Figure 5



The details of all the plans are given below:

7.2.1 ADMINISTRATIVE PLAN

The review group investigates the effect of the operational system on the administrative procedures of the user. The following activities are reviewed:

- **User Objectives:** This is an extremely critical area since it is possible that other time either the system fails to meet the user's initial objectives or the user objectives change as a reflection of changes in the organizational objectives. The results of the evaluation are documented for future reference.
- **Operating Costs and Benefits:** Under the administrative plan, the cost structure of the system is closely reviewed. This includes a review of all costs and savings, a review and update of the non-cost benefits of the system and a current budget designed to manipulate the costs and savings of the system.

7.2.2 PERSONAL REQUIREMENT PLAN

This plan evaluates all activities involving system personnel and the staff as they deal directly with the system. The emphasis is on productivity, morale and job satisfaction.

After the plan is developed, the review group evaluates the following:

- **Personnel Performance Objectives Compared with Current Performance Levels:** Turnover, tardiness, and absenteeism are also evaluated. The results are documented and made available to the maintenance group for follow-up.
- **Training Performance:** Through testing, interviews and other data gathering techniques, the review group attempts to answer questions about the adequacy of the training materials.

7.2.3 HARDWARE PLAN

The hardware of the new system is also reviewed, including terminals, CRT screens, software programs, the and the communication network. The primary target is the comparison of current performance specifications with design specifications. The outcome of the evaluation indicates any differences between expectations and realized results. It also points to any necessary modifications to be made.

7.2.4 DOCUMENTATION REVIEW PLAN

The reason for developing a documentation review plan is to evaluate the accuracy and completeness of the documentation complied to date and its conformity with pre-established documentation standards. Irregularities prompt action where changes in documentation would improve the format and content.

8. SYSTEM AUDIT

System Audit is also called **Process Audit**. It can be conducted for any activity, based on a specific document such as operating procedure, work instruction, training manual, etc.

Information Systems (IS) performs specific audits of computer applications and of information technology infrastructure and engages in the review of controls around new systems design and implementation. Information Systems assists the management to ensure that key system security controls are in place and to facilitate universal system security standards. In addition, through partnerships with other information system specialists from research institutions, IS ensures the adequacy of application controls supporting key business processes.

A software system audit consists of the following:

a. An overall view of the system documentation and an assessment of the quality of data files and databases. It also includes system maintainability, reliability and efficiency.

b. Functional information gathered on all the programs in the system to determine how well they do the job. Each program is assigned a preliminary ranking value.

c. A detailed program audit which considers the ranking value, mean time between failures, and size of the maintenance backlog.

Following are the steps involved in software modification:

a. Program rewrites which include logic simplification, documentation updates and error correction.

b. System level update which completes system level documentation and brings upto date data flow diagrams or system flowcharts and cross reference programs.

c. Reaudit of low ranking programs to make sure that the errors have been corrected.

The planned test of any system ought to include a thorough auditing technique and introduce control elements unique to the system. The Data Processing (DP) auditor should be involved in most phases of the system life cycle, especially system testing. If auditing is done after installing the system then the cost escalates and it would then not be wise to revert back and modify the system in order to incorporate adequate controls. Therefore, audit controls must be built during the phase of system design and tested. Then, the results and recommendations must be submitted to the system team in charge of the project. The user department should participate in reviewing the control specifications for the system to ensure that adequate control has been provided.

For testing programs, test data must include transactions that are specifically designed to violate control procedures incorporated in the program as well as valid transactions to test their acceptance by the system tested.

SUMMARY

- Testing is the last chance to detect and correct errors before the system is installed. Test data may be artificial or live. In either case they should provide all combinations of values or formats to test all logic and transaction path subordinates.
- Verification is the process of determining if a system meets the conditions set forth at the beginning. Validation is the process of evaluating a system to determine whether it satisfies the specified requirements.
- The different types of testing are unit testing, integration testing, system testing, positive testing and acceptance testing.
- Unit testing is the process of validating such small building blocks of a complex system much before testing an integrated large module or the system as a whole.
- Functional testing requires the selection of test scenarios without regard to source code structure.
- Structural testing requires that inputs be based solely on the structure of the source code or its data structures.
- The different trends in testing are installation, training and so on.
- The organizational process of changing over from the current information system to a new one is called “Installation”. Installation approaches are Direct, Parallel, Single location (Pilot), and Phased.
- The quality of training received by the personnel involved with the system in various capacities helps or hinders, and may even prevent, the successful implementation of an information system.
- As part of their training, operators should be given both a troubleshooting list that identifies possible problems and remedies for them, as well as the names and telephone numbers of individuals to contact when unexpected or unusual problems arise.
- User training may involve equipment use, particularly in the case where, say, a microcomputer is in use and the individual involved is both the operator and user. In these cases, users must be instructed first how to operate the equipment.
- Training methods are of two types – Vendor and In-Service Training, and In-House Training.
- A post-implementation review measures the system’s performance against predefined requirements.
- System audit can be conducted for any activity, based on a specific document such as operating procedure, work instruction, training manual, etc.

Chapter V

Object-Oriented System Development Life Cycle

After reading this chapter, you will be conversant with:

- Software Development Process
- Object-Oriented Systems Development
- Object-Oriented Analysis
- Object-Oriented Design
- Prototyping
- Component-Based Development
- Incremental Testing
- Reusability
- Object-Oriented Methodologies
- Unified Approach
- Modelling based on the Unified Modelling Language

Object-oriented approaches are seeking to resolve some of the problems of tranditional structured analysis and design. The Objects model in Object-oriented System Development (OOSD) provides a more realistic representation, which an end-user can more readily understand.

OOSD seeks to identify the objects in a problem to understand the structural and behavioral modularity and properties of each object and to recognize objects which are members of a common class and to share modularity, behavior, and properties, in a single consistent abstract model. In requirements analysis, this model identifies the required objects, classes, functions, behavior, and properties of the problem.

In design, this model facilitates the framing of architecture for software components with a smooth transition towards coding. The model is developed and viewed through graphic and textual representations which provide convenience in communication.

OOSD formally defines the properties of objects, describing the system as an object. The system is then refined into its component objects. Classes are methodically identified by generating them from objects in the system. A particularly thorough verification procedure establishes that the system is correctly implemented and achieves the required properties. Objects are treated uniformly, including the system object. By performing the same essential activities in analysis, preliminary design, and detailed design, an unusually consistent model is built, which would represent the real world very closely. This model can then be tested early and is easy to modify and re-use.

OOSD can be applied in a variety of development models including evolutionary, spiral, waterfall and prototyping.

1. SOFTWARE DEVELOPMENT PROCESS

Software development process is used to develop computer software. It may be an ad hoc process devised by the team for one project, but the term often refers to a standardized, documented methodology which has been used before on similar projects or one which is being used traditionally within an organization.

Software development process incorporates change, refinement, transformation or additions to the existing product. Within the process, it is possible to replace one sub-process with a new one, as long as the new sub-process has the same interface as the old one, to allow it to fit into the process as a whole. With this method of change, it is possible to adopt the new process. For example, the object-oriented approach provides a set of rules for describing inheritance and specialization in a consistent way when a sub-process changes the behavior of its parent process.

The process can be divided into small, and interacting phases known as sub-processes. The sub-processes must be defined in such a way that they are clearly understood to allow each activity to be performed as independently of other subprocesses as possible. Each subprocess must have the following:

- A description in terms of how it works.
- Specification of the input required for the process.
- Specification of the output to be produced.

The software development process also can be divided into smaller, interacting sub-processes. It can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation. Let us study three types of transformations:

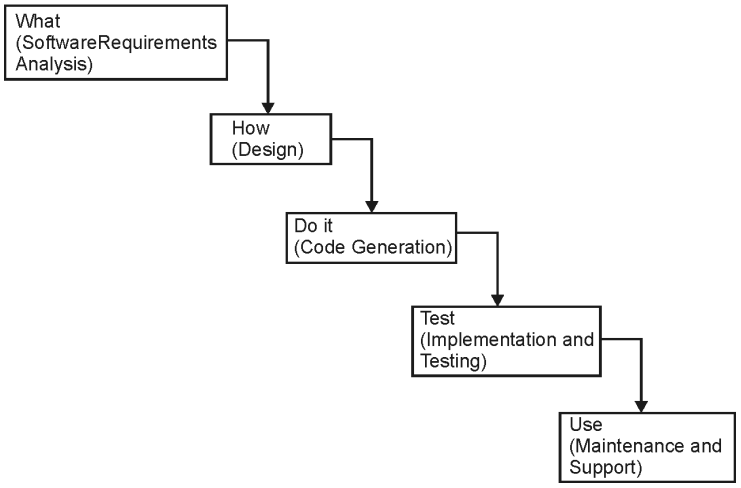
- *Transformation 1 (analysis)* translates the users' needs into system requirements and responsibilities. The usage of the system can provide insights into the users' requirements. For example, one use of the system might be analyzing an incentive payroll system, that needs to be included in the system requirements.

- *Transformation 2 (design)* begins with a problem statement and ends with a detailed design that can be transformed into an operational system. This transformation includes the bulk of the software development activity, including the definition of how to build the software, its development, and its testing. It also includes the design descriptions, the program and the testing materials.
- *Transformation 3 (implementation)* refines the detailed design into system deployment that will satisfy the users' needs. This takes into account the equipment, procedures, people, etc. It represents installing the software product within its operational environment. For example, for a new compensation method a program is written, new forms are put to use, and new reports are generated.

The waterfall approach to software development process starts with deciding *what* is to be done (what is the problem). Once the requirements have been determined, *how* to accomplish them must be decided. This is followed by actually doing the required things. Then, the results must be *tested* to see if the users' requirements are satisfied.

The waterfall approach to software development process is illustrated in figure 1.

Figure 1: Water Software Development Process



In the real world, the problems are not always well-defined and that is why the waterfall model has limited utility. In this model one step invariably follows the other. For example, if a company has experience in building accounting systems, then building another such system based on the existing design is best managed with the waterfall model. Where there is uncertainty regarding what is required or how it can be built, the waterfall model would not be of much use. This model assumes that the requirements are known before the design begins, but one may need experience with the product before the requirements can be fully understood. It also assumes that the requirements will remain static over the development cycle and that a product delivered months after it was specified will meet all the needs at its delivery time.

Finally, even when there is a clear specification, it assumes that sufficient design knowledge would be available to build the product. The waterfall model is the best way to manage a project with a well-understood product, especially very large projects.

Its failure can be traced to its inability to accommodate special needs of the software and its inappropriateness for resolving partially understood issues; it also neither emphasizes nor encourages software reusability.

After the system is installed in the real world, the environment frequently changes, altering the accuracy of the original problem statement and consequently generating revised software requirements. This can complicate the software development process even more. For example, a new class of employees or another shift of workers may be added or the standard workweek or the piece rate changed. Any such changes also change the environment, requiring changes in the programs. As each such request is processed, system and programming changes make the process increasingly complex, since each request must be considered with regard to the original statement of needs as modified by other requests.

2. OBJECT-ORIENTED SYSTEMS DEVELOPMENT

The world which we live in consists of objects of different kinds. The objects are found in nature, in the products made by human beings and also in businesses. An object represents an entity in real life and it may also be an abstraction. For instance, when we consider an airline reservation system, the objects in this system are aeroplanes, icons on the screen or a complete screen through which an operator issues tickets. Other examples of objects include customers in a bank, cats, atoms, molecules, students, balls of a string, bureaucrats etc. Smaller objects may be combined to form larger objects. Objects that we see around us can be described by means of their attributes and operations. Attributes are the characteristics of an object. For instance attributes of a cat are its color, size and weight. The operations that a cat performs are that it can catch mice, eat, make peculiar sounds, lick the owner etc.

Another point of interest is that an object can belong to or is a member of a larger class of objects. For instance, **chair** is an object that belongs to a larger class of objects called **furniture**. The attributes of furniture class are cost, dimensions, location, weight, color etc. Since chair is a member of this class, it inherits all attributes that are defined for the class. The example of class and an object is shown below.

Class: Furniture
Cost
Weight
Dimensions
Color
Location

Object: Chair
Cost
Weight
Dimensions
Color
Location

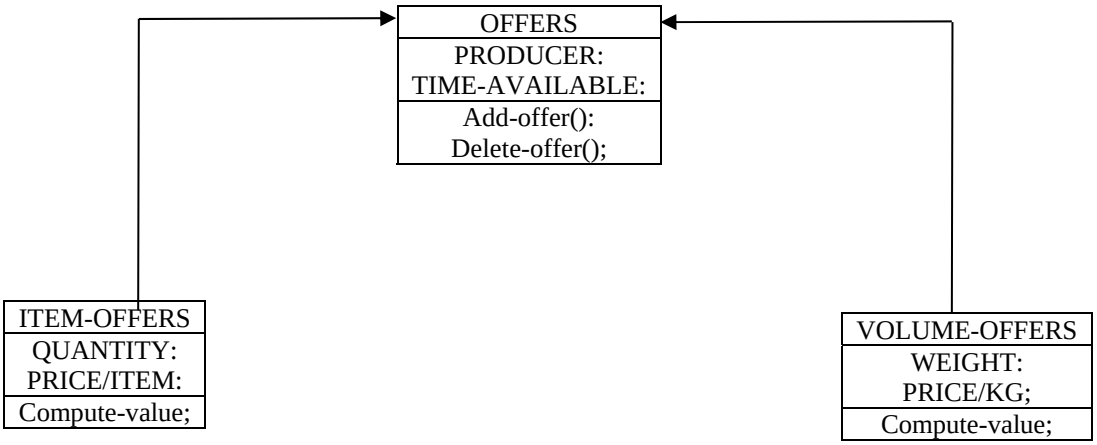
The object-oriented world that we live in has been extended to the design and construction of software in an abstract manner. In an object model, the entire data is stored as attributes of some object and these attributes are maneuvered by operations. Objects are permitted to use each others operations and it is only through operations that an object can manipulate another object. Object modeling is nothing but finding objects, their attributes and their operations, and putting them together in an object model. These objects work together to perform the tasks that are required. To perform the task, objects communicate with one another by sending messages. For instance, a client object requests the execution of a function (known as method) from a server object by sending it a message. Objects can function either as clients or servers according to the situation. An object is in the role of a client when it invokes another object irrespective of whether the objects are located in the same memory space or on different computers. Messages are like external forces that affect the state of the object which receives them. The sending of messages between objects is dependent on the architecture of the system that is being designed and the location of the objects that communicate with one another.

The functions of an object can be stated as follows:

- Storing or memorization of data or references and providing lookup.
- Performing some function on its own such as any computation.
- Invoking other objects to perform some action by sending them messages.

Inheritance

An object can inherit features from another object. It can also have additional features or, if needed, replace some of the features of another object. Consider an example of inheritance. We shall consider two objects: ITEM-OFFERS and VOLUME-OFFERS. Both of these objects inherit the features of another object OFFERS and also have their own additional properties and methods. This is represented below:



In the above given example, both ITEM-OFFERS and VOLUME-OFFERS will have the properties PRODUCER and TIME-AVAILABLE. ITEM-OFFERS also has the additional properties QUANTITY and PRICE/ITEM while VOLUME-OFFERS has the properties of WEIGHT and PRICE/KG. In addition, both the objects ITEM-OFFERS and VOLUME-OFFERS have a method, Compute-value but the computation performed by both these methods is different.

Polymorphism

Polymorphism is related to the concept of inheritance. It means the ability of one construct to take many forms. In object-oriented programming, it refers to the ability of a message to change its effect depending on the instance of object called. In the above given example, a message 'compute-value' addressed to OFFERS will select the appropriate method depending on the type of offer being considered. In addition, adding a new specialized object with a new specification of 'compute-value' will mean that the message 'compute-value' will select that method if addressed to an instance of the new class.

Data Abstraction

We have understood the concept of an object in our discussion. Let us consider an example of a real-life object, car. Now, a person who is interested in driving a car need not have the knowledge of internal combustion engine or technical details of its control system. It is enough for the driver to know the external components of a car such as the steering wheel and the horn.

In other words, the term abstraction refers to focusing on the essential, inherent aspects of an entity without worrying about its secondary aspects. It allows a user to put his/her attention on only those aspects that are essential for completing the task. Thus, only critical elements of the system are captured.

Encapsulation

By encapsulation we mean data hiding in which the external aspects of an object are separated from being accessed by other objects. Thus, the interface of an abstraction is separated from its implementation.

For instance, a Stack abstraction provides methods like push(), pop(), isEmpty(), isFull() etc. The Stack can be implemented as a singly linked list, a doubly linked list, an array, or a binary search tree. This feature is called encapsulation. It hides the details of the implementation of an object.

An object encapsulates both data and the operation, and all functionality is defined by operations. We can construct classes of objects through this important characteristic, leading to reusable classes and objects. Reusability has become the hallmark of modern software engineering due to the adoption of object-oriented paradigm in software development organizations. Another advantage is that the software components which come out through this paradigm bring with them the design characteristics such as functional independence and information hiding that is associated with high-quality software. The structure of object-oriented software is inherently decoupled. One more advantage of object-oriented software systems is that it is easier to adopt and scale because one can construct large systems by putting together reusable subsystems of other systems.

To hide the details of a class, its data or implementation can be declared in its private part so that it remains hidden from other classes and they cannot access it. Another advantage of encapsulation is that it would become possible to delay the resolution of the details till the design stage is reached and it would also be possible to have modular approach in the design of the code.

In the present times, the object-oriented paradigm has become the standard method in the software development process. Particularly, object-oriented languages like C++ or Java have become the de facto standards in the programming world. In the analysis and design phases of software development, object-oriented modeling approaches are becoming more and more acceptable to the software industry as standards. In this chapter we shall study various aspects of object-oriented software engineering such as analysis, design, modeling and object-oriented metrics.

2.1 Object-oriented Software Development Life Cycle (SDLC)

The object-oriented ***Software Development Life Cycle*** (SDLC) consists of three macro processes:

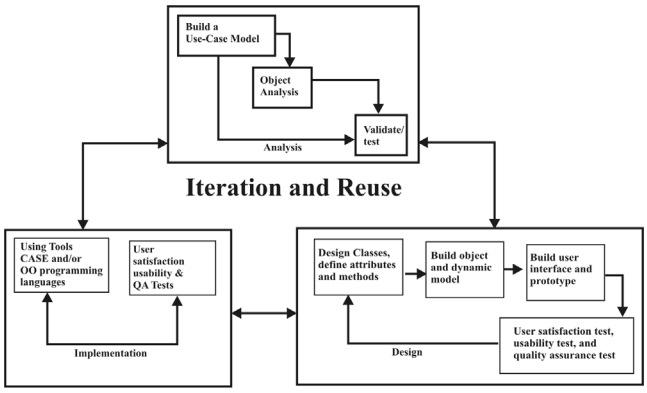
i. Object-oriented analysis,

ii. Object-oriented design, and

iii. Object-oriented implementation.

Figure 2 depicts the object-oriented Systems Development Approach.

Figure 2: Object-Oriented Systems Development Approach



The use-case model can be employed throughout most activities of software development. By following the life cycle model of Jacobson, Ericsson etc., one can produce designs that are traceable across requirements, analysis, implementation, and testing. The main advantage is that all design decisions can be traced back directly to user requirements. Usage scenarios can become test scenarios.

Object-oriented system development includes the following activities:

- Object-oriented analysis – use case driven.
- Object-oriented design.
- Prototyping.
- Component-based development.
- Incremental testing.

The characteristic of Object-oriented software development is different objects cooperate with one another. It advocates incremental development.

3. OBJECT-ORIENTED ANALYSIS

3.1 Definition

"Object-Oriented Analysis (OOA) is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain".

Object-Oriented Analysis (OOA) is concerned with developing software engineering requirements and identifying classes and their relationship with other classes in the problem domain. To understand the system requirements, one needs to identify the users or the actors. In object-oriented as well as traditional development, scenarios are used to help analysts understand requirements. These scenarios may not be fully documented. Ivar Jacobson came up with the concept of the *use case*, a name given by him for a scenario, to describe the user-computer system interaction. It became a primary element in system development. The object-oriented programming community has adopted use cases quiet extensively. Scenarios help in examining who does what in the interactions among objects and what ***role*** they play; that is, their interrelationships. This intersection among objects' roles to achieve a given goal is called ***collaboration***. The scenarios represent only one possible example of the collaboration. All aspects of the collaboration and all potential actions have several different scenarios which may be required, some showing usual behaviors, others showing situations involving unusual behavior or exceptions.

A use case is a typical interaction between a user and a system that captures users' goals and needs. Expressing these high-level processes and interactions with customers in a scenario and analyzing it, is referred to as *use-case modeling*. The use-case model represents the users' view of the system or users' needs.

The physical objects in the system also provide important information to the objects in the object-oriented systems. The objects could be individuals, organizations, machines, units of information, pictures, or anything related to the application and makes sense in the context of the real-world system. While developing the model, objects emerge that help establish a workable system. It is necessary to work iteratively between use-case and object models.

For example, objects in a flight reservation system might include an airplane, an airline flight, an icon on a screen, or even a full screen with which a travel agent interacts. OOA specifies the structure and the behavior of the object which comprise the requirements of that object. Different types of models are required to specify the requirements of the objects. The information or object model contains

the definition of objects in the system, which includes: the object name, the object attributes and the object's relationship with other objects. The behavior or state model describes the behavior of the object in terms of the state in which the object exists, the transitions allowed between object and the events that cause objects to change states. These models can be created and maintained using CASE tools that support representation of objects and object behavior.

Documentation is another important activity which does not end with object-oriented analysis but should be carried out throughout the system development process.

The results of analysis are requirement specifications which clearly describe the external behavior of the software, without any prejudgement about how the software will produce this exact behavior.

3.2 OO Analysis vs Structured Analysis

- OO technique provides a more consistent approach to system modelling.
- OO view more closely reflects the real world reflecting the way of thinking of human beings in terms of things which possess both attributes and behaviors.
- OO provides reuse possibility from the class hierarchy views of the system.
- OO analysis is able to model the user interface to a system.

3.3 Benefits of OOA

- **Maintainability** through simplified mapping to the real world, which provides for less analysis effort, less complexity in system design and easier verification by the user.
- **Reusability** of the analysis artifacts which saves time and costs depending on the analysis method and programming language.
- **Productivity** gains through direct mapping to features of Object-Oriented Programming Languages.

3.4 Shortcomings of OOA

- OO analysis techniques are still in the process of research and debate.
- The mixing of analysis and design methods is a problem with OO techniques.

4. OBJECT-ORIENTED DESIGN

"Object-oriented design is the construction of software systems as structured collections of abstract data type implementations, or "classes".

The goal of Object-Oriented Design (OOD) is to design the classes identified during the analysis phase and the user interface. During this phase, additional objects and classes that support implementation of the requirements are identified. The result of preliminary design is a description independent of language and technology. Detailed design results in the development of code. Many OOD methods have been described since the late 1980s. The most popular OOD methods include Booch, Buhr, Wasserman and the HOOD developed by the European Space Agency.

OOD builds on the products developed during Object-Oriented Analysis (OOA) by refining candidate objects into classes, defining message protocols for all objects, defining data structures and procedures, and mapping these into an Object-Oriented Programming Language (OOPL). Several OOD methods describe these operations on objects, although none is an accepted industry standard.

The term Object-Oriented Design (OOD) means different things to different people. For example, OOD has been used to imply such things as:

- The design of individual objects, and/or the design of individual methods contained in those objects,
- The design of an inheritance (specialization) hierarchy of objects,
- The design of a library of reusable objects, and
- The process of specifying and coding of an entire object-oriented application.

In general terms, analysis can mean listening to customers, making some notes and sketches, thinking about both the problems and potential solutions, and even constructing a few software prototypes. Design can mean the code-level design of an individual object, the development of an inheritance (specialization) hierarchy, or the informal definition and implementation of a software product (e.g., identify all the objects, create instances of the objects, and have the instances send messages to each other).

Hence, object-oriented design and object-oriented analysis are distinct discipline, which can be intertwined. Object-oriented development is highly incremental.

A design method in which a system is modelled as a collection of cooperating objects and individual objects is treated as an instance of a class within a class hierarchy. Four stages can be identified. They are:

- Identify the classes and objects,
- Identify their semantics,
- Identify their relationships, and
- Specify class and object interfaces and implementation.

Object-oriented design is one of the stages of object-oriented programming. First, the object model based on objects and their relationships is built and then the model is iterated and refined:

- Design and refine classes.
- Design and refine attributes.
- Design and refine methods.
- Design and refine structures.
- Design and refine associations.

A few guidelines to use in object-oriented design are:

- Reuse, rather than build, a new class. Know the existing classes.
- Design a large number of simple classes, rather than a small number of complex classes.
- Design methods.

The life cycle model of Jacobson et al., produces designs that are traceable across requirements, analyses, implementation and testing.

Problems associated with traditional structured design are as follows:

- It fails to take the evolutionary nature of software systems into accounts.
- Often the data structure aspect is neglected.
- Working top-down does not promote reusability.

Benefits of OO design are as follows:

- Information hidding.
- Weak coupling.

- Strong cohesion.
- Exensibility.

Shortcomings OOD are as follows:

- Difficulty in identifying a class.
- Blurred boundaries between design and both analysis and implemetation.
- Variable degrees of OO support in existing CASE tools.
- Elaborate and complex notations.

5. PROTOTYPING

A prototype is a working model that is functionally equivalent to a component of the product.

Although object-oriented analysis and design describe system features, it is important to construct a prototype of some of the key system components as soon as the products are selected. A prototype is a version of a software product developed in the early stages of the product's life cycle for specific, experimental purposes. A prototype enables to fully understand how easy or difficult it will be to implement some of the features of the system. It also can give users a chance to comment on the usability and usefulness of the user interface design and strike a balance between the software tools selected, the functional specification, and the user needs. Prototyping can also help in defining use cases, which makes use-case modelling much easier. The main idea is to build a prototype with uses-case modelling to design systems that users like and need. Companies may produce prototype products to display certain features or simply get a working model before refining other parts of the design. Prototyping can help resolve conflicts over requirements. When users actually use different prototypes they may change their preferences.

The **Prototyping Model** was developed on the assumption that it is often difficult to know all the requirements at the beginning of a project. Typically, users know many of the objectives that they wish to address with a system, but they do not know all the nuances of the data, nor do they know the details of the system features and capabilities. The **Prototyping Model** allows for these conditions, and offers a development approach that yields results without first requiring all information at the same time or in the beginning. When using the **Prototyping Model,** the developer builds a simplified version of the proposed system and presents it to the customer for consideration as part of the development process. The customer in turn provides feedback to the developer, who goes back to refine the system requirements to incorporate the additional information. Generally, the prototype code is not the final version. It is subsequently refined, modified and updated, resulting in the development of new programs, once requirements are identified.

5.1 Type of Prototype

Prototypes can be of various types. The following categories are some of the commonly accepted and represent very distinct ways of viewing a prototype, each having its own strengths:

- A ***horizontal prototype*** is a simulation of the interface but contains no functionality. This has the advantage of being very quick to implement, providing a good overall feel of the system, and allowing users to evaluate the interface on the basis of their normal, expected perception of the system.

- A ***vertical prototype*** is a subset of the system features with complete functionality. The principal advantage of this method is that the few implemented functions can be tested in great depth. In practice, prototypes are a hybrid between horizontal and vertical. The major portions of the interface are established such that the user can get the feel of the system, and risky features are prototyped with much more functionality.
- An ***analysis prototype*** is an aid for exploring the problem domain. This class of prototype is used to inform the user and demonstrate the proof of a concept It is not used as the basis of development and is discarded when it has served its purpose. The final product will use the concepts exposed by the prototype, not its code.
- A ***domain prototype*** is an aid for the incremental development of the ultimate software solution. It is often used as a tool for the staged delivery of subsystems to the users or other members of the development team. It demonstrates the feasibility of the implementation and eventually will evolve into a deliverable product.

The typical time required to produce a prototype is anywhere from a few days to several weeks, depending on the type and function of the prototype. Prototyping should involve representations from all user groups that will be affected by the project, especially the end-users and management members to ascertain that the general structure of the prototype meets the requirements established for the overall design. The purpose of this review is threefold:

i. To demonstrate that the prototype has been developed according to the specification and that the final specification is according to the requirements.

ii. To collect information about errors or other problems in the system, such as user interface problems that need to be addressed in the intermediate prototype stage.

iii. To provide an understanding of the role and importance of technology to everyone connected with the project.

The evaluation can be performed easily if the necessary supporting data is readily available. Testing considerations must be incorporated into the design and sub-sequent implementation of the system.

Prototyping is useful at any stage of the development. As key features are specified, prototyping those features usually results in modifications to the specification and even bring to light the need for additional features or highlight problems that are not obvious until the prototype is built.

6. COMPONENT-BASED DEVELOPMENT

A software component is something that can be deployed as a black box. It has an external specification, which is independent of its internal mechanisms.

Component-Based Development offers a new approach to the design, construction, implementation and evolution of software applications. Software applications are assembled from components from a variety of sources; the components themselves may be written in several different programming languages and run on several different platforms. A new generation of CASE tools is beginning to support component-based development.

Two basic ideas underlie Component-Based Development (CBD):

- The application development can be improved significantly if applications can be assembled quickly from prefabricated software components.
- An increasingly large collection of interpretable software components could be made available to developers in both general and specialist catalogs.

These two ideas move application development from a craft activity to an industrial process fit to meet the needs of modem, highly dynamic, competitive, global businesses.

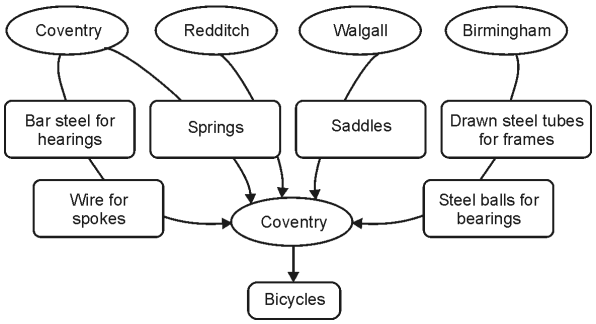
A CBD developer can assemble components to construct a complete software system. Components themselves may be constructed from other components and so on down to the level of pre-built components or old-fashioned code written in a language such as C or COBOL. Visual tools or actual code can be used to put together components. Although it is practical to do simple applications by wiring components together as in Digitalk's Smalltalk PARTS or IBM's Visual Age, putting together a practical application still poses some challenges. All these are invisible to end-users. The impact to users will come from faster product development cycles, increased flexibility and improved customization features. CBD will allow independently developed applications to work together and do so more efficiently and with less development effort.

CBD can be regarded as an extension to conventional software development and management. In other words, some component requirements are satisfied through CBD while some other requirements are satisfied using other (conventional) techniques. Conventional development is then a special case of CBD, which lacks some of the techniques and opportunities (and of course the benefits) that characterize full CBD. We then get a spread of possibilities, from conventional development at one end, and extreme componentization at the other end. One of the key questions to be addressed by a designer is the degree of componentization in a particular situation.

Software components are the functional units of a program – the building blocks offering a collection of reusable services. A software component can request a service from another component or deliver its own services on request. The delivery of services is independent, which means that components work together to accomplish a task. Components may depend on one another without interfering with each other. Each component is unaware of the context or inner workings of the other components i.e., the object-oriented concept addresses analysis, design, and programming, whereas component-based development is concerned with the implementation and system integration aspects of software development.

Figure 3, shows the traditional manufacturing process of bicycles, with different components sourced from different places

Figure 3: Traditional Manufacturing Process



As shown in figure 3, the manufacture of bicycles illustrates some of the principal ideas of component-based software development.

7. INCREMENTAL TESTING

Quality assurance is an important part of the software development process. In this context, however, we want to focus on performance testing rather than quality testing. Performance testing is really about benchmarking. It is to be noted that benchmarks are constructed with regard to performance of a running system. These benchmarks should be based on the performance requirements. Having a solid set of benchmarks facilities tracking progress over time and see where we

stand with regard to requirements. Meeting or exceeding the performance requirements should be part of the shipping criteria for the final product.

Testing should be a planned and documented activity. Especially in incremental and iterative development processes the repeatability of tests is very important.

7.1 Testing Types

Following are the different testing types.

- **Unit Testing:** Test methods within each object.
- **Integration Testing:** Test collaborations between objects.
- **System Testing:** Test the entire system as a collection of objects.
- **Acceptance Testing:** Test for standards and customer satisfaction.

Debugging should actually take place continuously during the development cycle. In particular, it should be done in both:

a. **Design Phase:** This involves detecting and removing **logical errors**. It is accomplished by "tracing" the algorithm (penciling through each step); this is also called performing a "walkthrough".

b. **Coding Phase:** This involves removing **syntax errors** which are mistakes in the spelling and grammar of the particular programming language.

After coding, testing and debugging is also needed to catch faults that might have slipped through:

a. **Testing** involves using **simple but complete test data** specifically chosen to expose possible errors ("bugs").

b. **Debugging** is the process of removing these errors.

8. REUSABILITY

Reusability is "the degree to which a software module or other work product can be used in more than one computing program or software system". A major benefit of object-oriented system development is reusability. For an object to be really reusable, a large effort must be spent designing it. To deliver a reusable object, the development team must have the up-front time to design reusability into the object.

The reuse strategy can be based on the following:

- Information hiding (encapsulation).
- Conformance to naming standards.
- Creation and administration of an object repository.
- Encouragement by strategic management of reuse as opposed to constant redevelopment.
- Establishing targets for a percentage of the objects in the project to be reused.

The benefits of Reuse are:

- Increased reliability,
- Reduced time and cost for development, and
- Improved consistency.

9. OBJECT-ORIENTED METHODOLOGIES

Object-oriented methodology is a set of methods, models and rules for developing systems. Modelling is the process of describing an existing or proposed system. It can be used during any phase of the software life cycle. A model is an abstraction of a phenomenon for the purpose of understanding it. Modelling provides a means for communicating ideas in an easy to understand and unambiguous form while also accommodating a system's complexity.

An appropriate lifecycle methodology for OO developments must contain all of the following components:

- A full lifecycle process for both business and technological issues;
- A full set of concepts and models which are internally self-consistent;
- A collection of rules and guidelines;
- A full description of all deliverables;
- A workable notation;
- Ideally supported by third party drawing tools;
- A set of tried and tested techniques;
- A set of appropriate metrics, standards and test strategies; and
- Identification of organizational roles to be performed by business analysts, etc., and programmers; etc., and guidelines for project management and quality assurance.

Some of the important points with regard to the development of object-oriented methodologies are as follows:

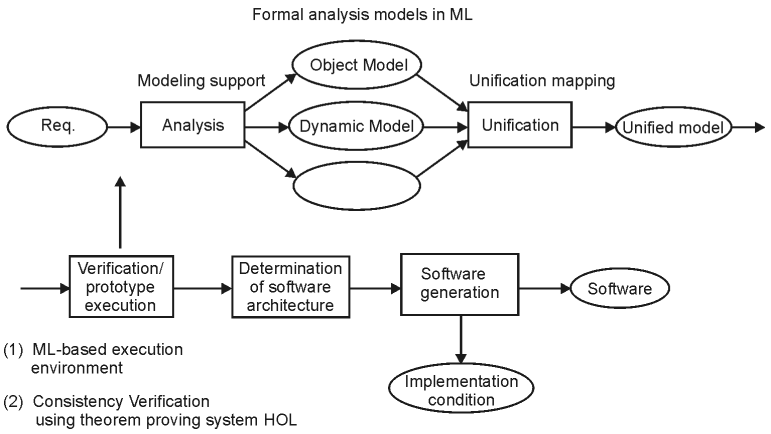
- Booch developed the object-oriented design concept – the Booch method.
- Sally Shaler and Steve Mellor created the concept of recursive design approach.
- Beck and Cunningham produced class-responsibility-collaboration cards.
- Wirfs-Brock, Wilkerson and Wiener came up with responsibility driven design.
- Jim Rumbaugh led a team at the research labs of General Electric to develop the object modelling technique.
- Peter Coad and ED Yourdon developed the Coad lightweight and prototype-oriented approach to methods.
- Ivar Jacbson introduced the concept of the use case and Object-Oriented Software Engineering (OOSE).

Many methodologies are available to choose from for system development. Each methodology is based on modelling the business problem and implementing the application in an object-oriented fashion; the differences lie primarily in the documentation of information and modelling notations and languages. Two people using the same methodology may produce application designs that look radically different. This does not necessarily mean that one is right and one is wrong, just that they are different.

In this, we look at the methodologies and their modelling notations developed by Rumbaugh, Booch and Jacobson which led to the development of the Unified Modelling Language (UML). Each method has its advantages. The Rumbaugh method is well-suited for describing the object model or the static structure of the system. The Jacobson method is good for producing user-driven analysis models. The Booch method produces detailed object-oriented design models.

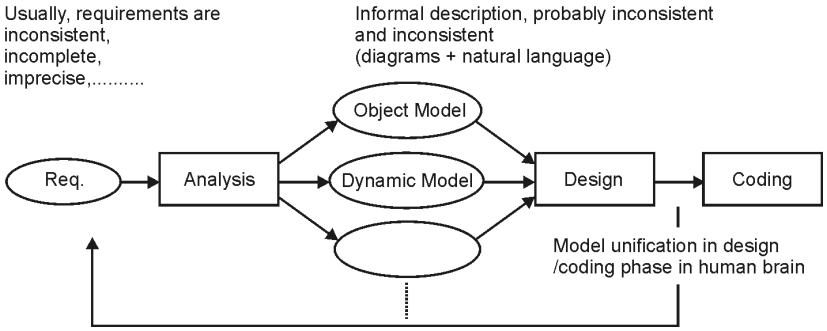
The formal OO Methodology is shown in figure 4:

Figure 4: Formal OO Methodology



The Current OO Methodology is shown in figure 5:

Figure 5: Current OO Methodology



9.1 Rum Baugh's Object Modelling Technique (OMT)

OMT, presented by Jim Rum Baugh and his co-workers, describes the method for analysis, design and implementation of a system using an object-oriented technique. OMT is a fast, and intuitive approach for identifying and modelling all the objects making up a system. OMT consists of four phases, which can be performed iteratively:

i. ***Analysis:*** Starting from a statement of the problem, the analyst builds a model of the real-world situation showing its important properties. The analysis model is a concise, precise abstraction of *what* the desired system must do, not *how* it will be done. The objects in the model should be application-domain concepts and not computer implementation concepts such as data structures. A good model can be understood and criticized by application experts who are not programmers. The analysis model should not contain any implementation decisions. For example, a *Window* class in a workstation windowing system would be described in terms of the attributes and operations visible to a user.

ii. ***System Design:*** The system designer makes high-level decisions about the overall architecture. During system designing, the target system is organized into subsystems based on both the analysis structure and the proposed architecture. The system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem, and make tentative resource allocations. For example, the system designer might decide that changes to the workstation screen must be fast and smooth even when windows are moved or erased and choose an appropriate communications protocol and memory buffering strategy.

iii. ***Object Design:*** The object designer builds a design model based on the analysis model but containing implementation details. The designer adds details to the design model in accordance with the strategy established during system design. The focus of object design is the data structures and algorithms needed to implement each class. The object classes from analysis are still meaningful, but they are augmented with computer-domain data structures and algorithms chosen to optimize important performance measures. Both the application-domain objects and the computer-domain objects are described using the same object-oriented concepts and notation, although they exist on different conceptual planes. For example, the *Window* class operations are now specified in terms of the underlying hardware and operating system.

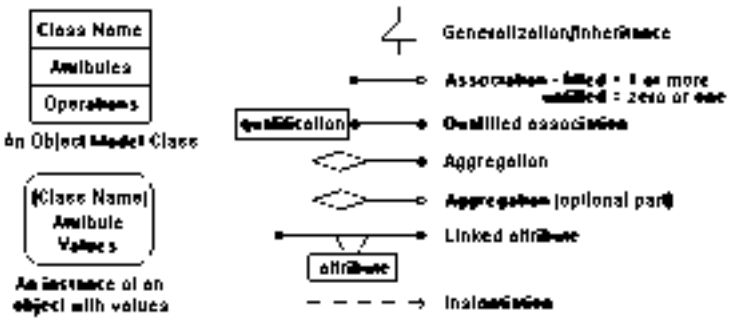
iv. ***Implementation:*** The object classes and relationships developed during object design are finally translated into a particular programming language, database or hardware implementation. Programming should be a relatively minor and mechanical part of the development cycle, because all of the hard

decisions should be made during design. The target language influences design decisions to some extent, but the design should not depend on fine details of a programming language. During implementation, it is important to follow good software engineering practices so that traceability to the design is straight forward and the implemented system remains flexible and extensible. For example, the *Window* class would be coded in a programming language using calls to the underlying graphics system on the workstation.

Object-oriented concepts can be applied throughout the system development life cycle, from analysis through design to implementation. The same classes can be carried from stage to stage without a change of notation. Although the analysis view and the implementation view of *Window* are both correct, they serve different purposes and represent a different level of abstraction. The object-oriented concepts of identity, classification, polymorphism and inheritance apply through the entire development cycle.

The symbols used in this method are shown in figure 6:

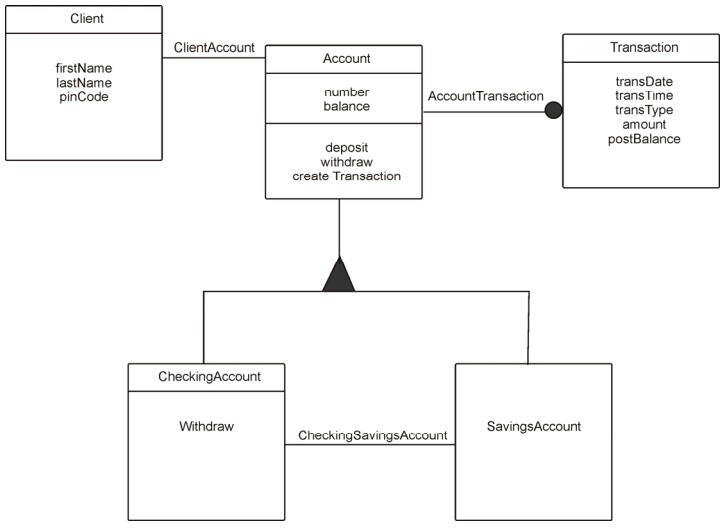
Figure 6: Symbols used in OO Paradigm



The Object Modeling Technique provides three sets of concepts which provide three different views of the system. There is a method which leads to three models of the system corresponding to these views. The models are initially defined, and then refined as the phases of the method progress. The three models are discussed below.

i. The *object model* describes the static structure of the objects in a system and their relationships. The object model contains object diagrams. An *object diagram* is a graph whose nodes are object *classes* and whose arcs are *relationships* among classes. Each class represents a set of individual objects. The object diagram of a bank system is shown in figure 7:

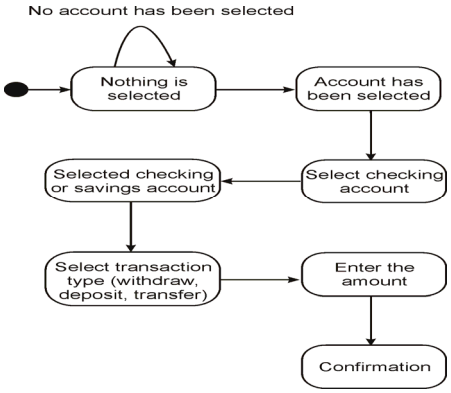
Figure 7: Example of a Bank System in OO Model



ii. The *dynamic model* describes those aspects of a system which are concerned with time and the sequencing of operations – events that mark changes, sequences of events, states that define the context for events, and the organization of events and states. The dynamic model captures *control* – that aspect of a system which describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented.

The dynamic model is represented graphically with state diagrams. Each state diagram shows the state and event sequences permitted in a system for one class of objects. State diagrams also refer to the other models. Actions in the state diagrams correspond to functions from the functional model; events in a state diagram become operations on objects in the object model. The state diagram for the bank application user interface is shown in figure 8.

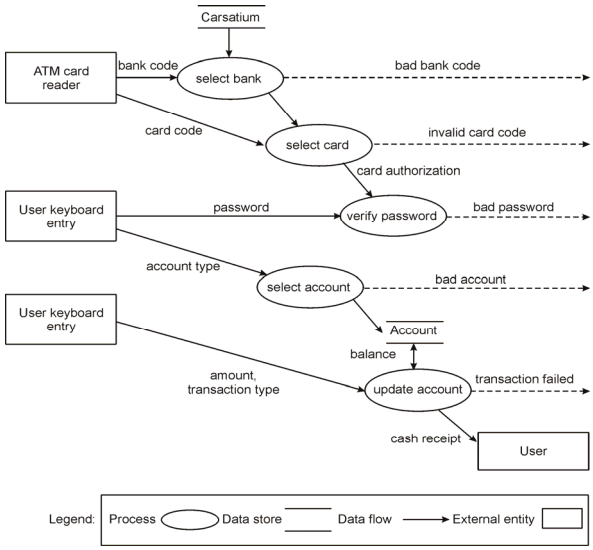
Figure 8: State Diagram for the Bank Application



iii. The *functional model* describes those aspects of a system concerned with transformations of values – functions, mappings, constraints, and functional dependencies. The functional model captures what a system does, without regard for how or when it is done.

The functional model is represented with data flow diagrams. Data flow diagrams show the dependencies between values and the computation of output values from input values and functions. Traditional computing concepts such as expression trees are examples of functional models. Functions are invoked as actions in the dynamic model and are shown as operations on objects in the object model. The Data Flow Diagram of the ATM system is given below.

Figure 9: DFD of the ATM System



9.2 Booch Methodology

The Booch methodology is a widely used object-oriented method that helps to design the system using the object paradigm. It covers the analysis and design phases of an object-oriented system. Booch defines a number of symbols to document almost every design decision. The Booch method consists of the following diagrams:

- Class diagrams
- Object diagrams
- State transition diagrams
- Module diagrams
- Process diagrams
- Interaction diagrams.

The Booch methodology prescribes.

- A macro development process, and
- A micro development process.

9.2.1 THE MACRO DEVELOPMENT PROCESS

The macro process serves as a controlling framework for the micro process. The primary concern of the macro process is technical management of the system. Such management is interested less in the actual object-oriented design than in how well the project corresponds to the requirements set for it and whether it is produced on time. In the macro process, the traditional phases of analysis and design are preserved to a large extent.

The macro development process consists of the following steps:

- *Conceptualization* – In this, the core requirements of the system are established. Also, a set of goals are established to develop a prototype for proving the concept.
- *Analysis and development of the model* – In this,
  a. The **class diagrams** are used to describe the roles and responsibilities of objects in the system.
  b. The **object diagram** is used to describe the behavior of the system, and the interaction diagram is used to describe the behavior of the system in terms of scenarios.
- *Designing or Creating* the system architecture – In this,
  a. The **class diagram** is used to decide what classes exist and how they relate to each other.
  b. Then, **object diagram** is used to decide what mechanisms are used to regulate the collaboration among the objects.
  c. Next, **module diagram** is used to map out where each classes and objects should be declared.
  d. Finally, the **process diagram** is used to determine which processor to be allocated to a process.
- *Evolution or implementation* – In this step, a stream of software implementations are produced, each of which is a refinement of the prior one.
- *Maintenance* – Localized changes to the system are made so as to add new requirements and eliminate bugs.

9.2.2 THE MICRO DEVELOPMENT PROCESS

The micro process is a description of the day-to-day activities by a single or small group of software developers. It consists of the following steps:

i. Identify classes and objects.
ii. Identify classes and object Semantics.
iii. Identify classes and object relationships.
iv. Identify classes and object interfaces and implementation.

9.3 Jacobson Methodologies

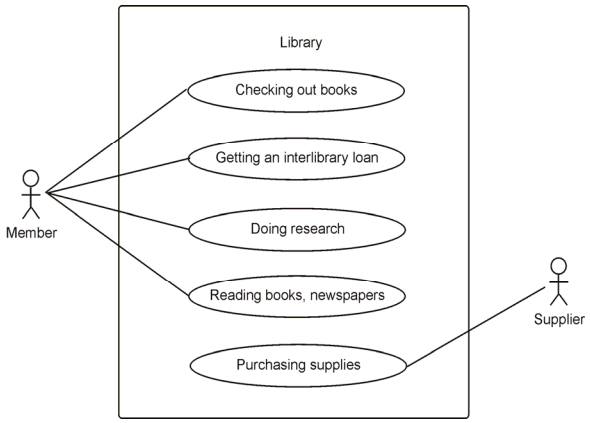
Jacobson methodologies cover entire life cycle and stress traceability between the different phases, both forward and backward. At the heart of Jacobson methodologies is the use-case concept, which evolved with Object Factory for Software Development. Jacobson methodologies include:

- Object-Oriented Business Engineering (OOBE).
- Object-Oriented Software Engineering (OOSE).

9.3.1 USE CASES

Use cases are scenarios for understanding system requirements. A use case is an interaction between users and a system. The use-case model captures the goal of the user and the responsibility of the system to its users. The use case diagram for the library system is shown below:

Figure 10: Use-Case Diagram for the Library System



In requirements analysis, the use cases are described as one of the following:

- No formal text with no clear flow of events.
- Text, easy to read but with a clear flow of events to follow (this is a recommended style).
- Formal style using pseudocode.

The use case description must contain the following:

- How and when the use case begins and ends?

  The interaction between the use case and its actors, including when the interaction occurs and what is exchanged.

  How and when the use case will need data store in the system or will store data in the system.

- Exceptions to the flow of events.
- How and when the concepts of the problem domain are handled?

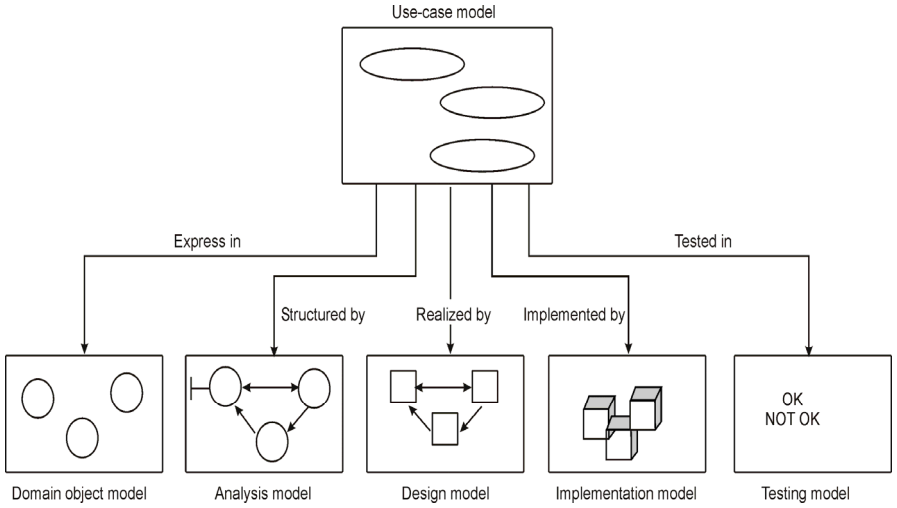
  Every single use case should describe one main flow of events. An exceptional or additional flow of events could be added. The exceptional use case extends another use case to include the additional one. The use case model employs extends and uses relationships. The extends relationship is used when there is one use case that is similar to another use case but does a bit more. It extends the functionality of the original use case (like a subclass). The uses relationship reuses common behavior in different use cases.

Use cases could be viewed as concrete or abstract. An abstract use case is not complete and has no actors that initiate it but is used by another use case. This inheritance could be used in several levels. Abstract use cases also are the ones that have uses or extend relationships.

9.3.2 OBJECT-ORIENTED SOFTWARE ENGINEERING (OBJECTORY)

Object-oriented Software Engineering (OOSE) is also called as **Objectroy**. It is a method of object-oriented development with the specific aim to fit the development of large, real-time systems. The development process, called use-case driven development, stresses that use cases are involved in several phases of the development, including analysis, design, validation and testing. It is shown in figure 11. The use-case scenario begins with a user of the system initiating a sequence of interrelated events.

Figure 11: Use-case Development



The system development method based on OOSE is a disciplined process for the industrialized development of software, based on a use-case driven design. It is an approach to object-oriented analysis and design that centers on understanding the ways in which a system actually is used. By organizing the analysis and design models around sequences of user interaction and actual usage scenarios, the method produces systems that are both more usable and more robust, adapting more easily to changing usage.

Objectory is built around different models:

- *Use case-model:* The use-case model defines the outside (actors) and inside (use case) of the system's behavior.
- *Domain object model:* The objects of the "real" world are mapped into the domain object model.
- *Analysis object model:* The analysis object model presents how the source code (implementation) should be carried out and written.
- *Implementation model:* The implementation model represents the implementation of the system.
- *Test model:* The test model constitutes the test plans, specifications, and reports.

The maintenance of each model is specified in its associated process. A process is created when the first development project starts and is terminated when the developed system is taken out of service.

9.3.3 OBJECT-ORIENTED BUSINESS ENGINEERING

Object-Oriented Business Engineering (OOBE) is object modeling at the enterprise level. Use cases again are the central vehicle for modeling, providing traceability throughout the software engineering process.

- *Analysis Phase:* The analysis phase defines the system to be built in terms of the problem-domain object model, the requirements model and the analysis model. The analysis process should not take into account the actual implementation environment. This reduces complexity and promotes maintainability over the life of the system, since the description of the system will be independent of hardware and software requirements. This model should be developed just enough to form a base of understanding for the requirements model. The analysis process is iterative but the requirements and analysis models should be stable before moving on to the subsequent models. Jacobson suggests that prototyping with a tool might be useful during this phase to help specify user interfaces.
- *Design and Implementation Phases:* The implementation environment must be identified for the design model. This includes factors such as Database Management System (DBMS), distribution of process, constraints due to the programming language, available component libraries and incorporation of graphical user interface tools. It may be possible to identify the implementation environment concurrently with analysis. The analysis objects are translated into design objects that fit the current implementation environment.
- *Testing Phase:* Finally, Jacobson describes several testing levels and techniques. The levels include unit testing, integration testing and system testing.

10. UNIFIED APPROACH

Unified approach is based on the best practices that have proven successful in systems development and more specifically, the work done by Booch, Rumbaugh, and Jacobson in their attempt to unify their modelling efforts. The Unified Approach (UA) establishes a unifying and unitary framework around their works by utilizing the Unified Modelling Language (UML) to describe, model, and document the software development process. The main motivation is to combine the best practices, processes, methodologies and guidelines along with UML notations and diagrams for better understanding object-oriented concepts. The unified approach to software development revolves around (but it is not limited to) the following processes and concepts:

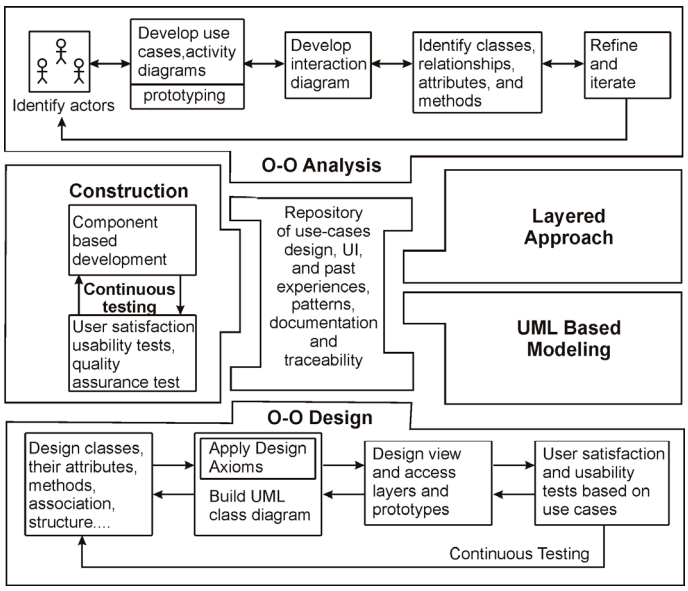
- Use-case driven development.
- Object-oriented analysis.
- Object-oriented design.
- Incremental development and prototyping.
- Continuous testing.

The methods and technology employed include:

- Unified modelling language used for modelling.
- Layered approach.
- Repository for object-oriented system development patterns and frameworks.
- Component-based development.

The processes and components of the unified approach are shown in figure 12.

Figure 12: Processes and Components of the Unified Approach



10.1 Object-Oriented Analysis

Analysis is the process of describing the needs and duties of a system so as to satisfy the users' requirements. The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system. This is accomplished by constructing several models of the system. These models concentrate on describing what the system does rather than how it does it. Separating the behavior of a system from the way it is implemented requires viewing the system from the user's perspective rather than that of the machine.

OOA Process consists of the following steps:

- Identify the Actors.
- Develop a simple business process model using UML Activity diagram.
- Develop the Use Case.
- Develop interaction diagrams.
- Identify classes.

10.2 Object-Oriented Design

Booch provides the most comprehensive object-oriented design method. Rumbaugh and Jacobson's high-level models provide good avenues for getting started. UA combines these by utilizing Jacobson's analysis and interaction diagrams, Booch's object diagrams and Rumbaugh's domain models. By following Jacobsons life cycle model, we can produce designs that are traceable across the various phases of life cycle viz., requirements, analysis, design, coding and testing. OOD process consists of,

- Designing classes, their attributes, methods, associations, structures, protocols, and applying design axioms.
- Designing the Access Layer.
- Designing and prototyping User Interface.
- Conducting User Satisfaction and Usability Tests based on the Usage/Use Cases.
- Iterating and refining the design.

10.3 Iterative Development and Continuous Testing

We must iterate and reiterate until, we obtain a satisfactory system. Since testing often uncovers design weaknesses or at least provides additional information that is needed. While reprototyping and retesting, it is necessary to learn from each repetition and rework accordingly. This refining cycle is continued through the development process until one is satisfied with the results. During this iterative process, the prototype will be incrementally transformed into the actual application. UA encourages the integration of testing plans from day one of the project.

11. MODELING BASED ON THE UNIFIED MODELING LANGUAGE

The Unified Modeling Language (UML) was developed with the joint efforts of leading object technologists Grady Booch, Ivar Jacobson and James Rumbaugh with contributions from many others. The UML merges the best of the notations used by the three most popular analysis and design methodologies: Booch's methodology, Jacobson's use case and Rumbaugh's object modeling technique. The UML is becoming the universal language for modeling systems; it is intended to be used to express models of many different kinds and purposes. The UML has become the standard notation for object-oriented modeling systems. The UA uses the UML to describe and model the analysis and design phases of system development (UML) notations.

11.1 The UA Proposed Repository

The idea is to create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks and user interfaces in an easily accessible manner with a completely available and easily utilized format. Everything from the original user request to maintenance of the project as it goes to production should be kept in the repository. The advantage of repositories is that, the organizations can use the objects of past projects, stored in the repositories for future projects. For instance, one can select from repository, a data element, a diagram, various symbols and associated dependents for reuse.

The UA' s underlying assumption is that, if we design and develop applications based on previous experience, creating additional applications will require no more than assembling components from the library. Additionally, applying lessons learned from the past developmental mistakes to future projects will increase the quality of the product and reduce the cost and development time. If a new requirement surfaces, new objects will be designed and stored in the main repository for future use. Specifications of the software components, describing the behavior of the component and how it should be used, are registered in the repository for future reuse by teams of developers.

The repository should be accessible to many people. Furthermore, it should be relatively easy to search the repository for classes based on their attributes, methods, or other characteristics. For example, application developers could select previously built components from the central component repository that match their business needs and assemble these components into a single application, customizing where needed.

Tools to fully support a comprehensive repository are not accessible yet, but this will change quickly and, in the near future, tools would be available to capture all phases of software development into a repository for reuse.

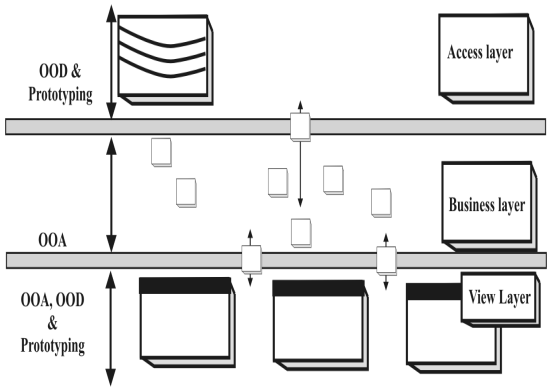
11.2 The Layered Approach to Software Development

Most systems developed with today's CASE tools or client-server application development environments tend to lean towards what is known as two-layered architecture: interface and data.

In a two-layered system, user interface screens are tied to the data through routines that sit directly behind the screens; for example, a routine that executes when a

button is clicked. With every interface that is created, the business logic needed to run the screen is re-created. The routines required to access the data must exist within every screen. Any change to the business logic must be accomplished in every screen that deals with that portion of the business. This approach results in objects that are very specialized and cannot be reused easily in other projects.

Figure 13: The Three Layer Approach



A better approach to systems architecture is one that isolates the functions of the interface from the functions of the business. This approach also isolates the business from the details of the data access. Using the three-layered approach, one is able to create objects that represent tangible elements of business that are completely independent of the way they are represented to the user through the interface or how they are physically stored (in a database). The three-layered approach as shown in figure 13, consists of a view of user interface layer a business layer, and an access layer.

11.2.1 THE BUSINESS LAYER

The business layer contains all the objects that represent the business (both data and behavior). The business layer is responsible for modeling the objects of the business and depicting the way they interact to accomplish the business processes.

The business objects should not be responsible for –

- **Displaying details:** Business objects should have no special knowledge of how they are being displayed and by whom.
- **Data access details:** Business objects also should have no special knowledge of where they come from.

A business model captures the static and dynamic relationships among a collection of business objects. Static relationships include object associations and aggregations. Dynamic relationships show how the business objects interact to perform tasks. Business models incorporate control objects that direct their processes. Business objects are identified during object-oriented analysis. Use case can provide a wonderful tool to capture business objects.

11.2.2 THE USER INTERFACE LAYER (VIEW)

The user interface layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface.

Responsibilities

- **Responding to User Interaction:** The user interface layer objects must be designed to translate actions by the user.
- **Displaying Business Objects:** The user interface layer paints the best possible picture of the business objects for the user.

The user interface layer's objects are identified during the object-oriented design phase.

11.2.3 THE ACCESS LAYER

The access layer contains objects that know how to communicate with the place where the data actually reside.

Responsibilities

- **Translate Request:** The access layer must be able to translate any data-related requests from the business layer into the appropriate protocol for data access.
- **Translate Results:** The access layer must also be able to translate the data retrieved back into the appropriate business objects and pass those objects back into the business layer.
- Access objects are identified during object-oriented design.

SUMMARY

- In an object-oriented environment, software is a collection of discrete objects that encapsulate their data and the functionality to model real-world "objects". Once objects are defined, one can take it for granted that they will perform their desired functions and so seal them off in one's mind like black boxes. One's attention as a programmer shifts to what they do rather than how they do it. The object-oriented life cycle encourages a view of the world as a system of cooperative and collaborating agents.
- An object orientation produces systems that are easier to evolve, more flexible, more robust and more reusable than a top-down structure approach.
- In System Development Life Cycle (SDLC), the essence of the software process is the transformation of users' needs through the application domain into a software solution that is executed in the implementation domain. The concept of the use case, or a set of scenarios, can be a valuable tool for understanding the users' needs. The emphasis on the analysis and design aspects of the software life cycle is intended to promote building high-quality software (meeting the specifications and being adaptable for change).
- Object-oriented design requires more rigors up front to do things right. One needs to spend more time on gathering requirements, developing a requirement model and an analysis model, and then turning them into the design model. Object-oriented systems development consists of three macro processes: object-oriented analysis, object-oriented design, and object-oriented implementation. Object-oriented analysis requires building a use-case model and interaction diagrams to identify user's needs and the system's classes and their responsibility, then validating and testing the model documenting each step along the way. Object-oriented design centers around establishing design classes and their protocol; building class diagrams, user interfaces and prototypes; testing user satisfaction and usability based on usage and use software development; furthermore, by following Jacobson's life cycle model, one can produce designs that are traceable across requirements, analysis, design, implementation and testing.
- Component-Based Development (CBD) is an industrialized approach to software development. Software components are functional units, or building blocks offering a collection of reusable services. A CBD developer can assemble components to construct a complete software system. Components themselves may be constructed from other components and so on down to the level of pre-built components or old-fashioned code written in a language such as C, assembler or COBOL.

- Reusability is a major benefit of object-oriented system development. It is also the most difficult promise to deliver. To develop reusable objects, one must spend time up front to design reusability in the objects.
- Rumbaugh et al. has a strong method for producing object models (sometimes known as domain object models). Jacobson et al. has a strong method for producing user-driven requirement and object-oriented analysis models. Booch has a strong method for producing detailed object-oriented design models.
- Each method has a weakness, too. While Rumbaugh et al.'s OMT has string methods for modeling the problem domain, OMT models cannot fully express the requirements. Jacobson et al. deemphasize object modeling and, although they cover a fairly wide range of the life cycle, they do not treat object-oriented design to the same level as Booch who focuses almost entirely on design, and not analysis.
- Booch and Rambaugh et al. are object centered in their approaches and focus more on figuring out what are the objects of a system, how are they related, and how do they collaborate with each other. Jacobson et al. are more user centered, in that everything in their approach derives from use cases or usage scenarios.
- The UA is an attempt to combine the best practices, processess and guidelines along with UML notations and diagrams for better understanding object-oriented concepts and object-oriented system development. The UA consists of the following processes: (a) Use case driven development, (b) Object oriented analysis, (c) Object oriented design, (d) Incremental development and prototyping, and (e) Continuous testing.
- Furthermore, it utilizes the methods and technologies such as unified modeling language and layered approach. It promotes repository for all phases of software development.
- A software engineering methodology consists of a process for organized development based on a set of techniques. The OMT methodology is based on the development of a three-part model of the system, which is then refined and optimized to constitute a design. The object model captures the objects in the system and their relationships. The dynamic model describes the reaction of objects in the system to events and the interactions between objects. The functional model specifies the transformation of object values and constraints on these transformations. The object modeling technique produces systems that are more stable with respect to changes in requirements than traditional function-oriented approaches.
- The purpose of analysis is to understand the problem and the application domain so that a correct design can be constructed. A good analysis captures the essential features of the problem without introducing implementation artifacts that prematurely restrict design decisions.
- The object model shows the static structure of the real world. First, identify the object classes and then identify associations between objects, including aggregations. Object attributes and links should be defined, although minor ones can be deferred. Inheritance should be used to organize and simplify the class structure. Organize tightly-coupled classes and associations into modifies.

- The dynamic model shows the behavior of the system – especially sequencing of interactions. First, prepare scenarios of typical and exceptional sessions. Then, identify external events between the system and the outside world. Build a state diagram for each active object showing the patterns of events it receives and sends, together with actions that it performs. Match events between state diagrams to verify consistency; the resulting set of state diagrams consume the dynamic model.
- The functional model shows the functional derivation of values without regard for when they are computed. First, identity input and output values of the system as parameters of external events. Then, construct data flow diagrams to show the computation of each output value from other values and ultimately input values. Data flow diagrams interact with internal objects that serve as data stores between iterations.
- Methodologies are never linear. This one is no exception. Any complex analysis is constructed by iteration on multiple levels. All parts of the model need not be developed at the same pace. The result of analysis replaces the original problem statement and serves as the basis for design.

Chapter VI

UML Models

After reading this chapter, you will be conversant with:

- Static and Dynamic Models
- Unified Modeling Language
- The Meta-Model
- Use-Case Diagrams
- Activity Diagrams
- Interaction Diagrams
- Sequence Diagrams
- Collaboration Diagrams
- Class Diagrams
- Object Diagrams
- State Chart Diagrams
- Implementation Diagrams
- Component Diagrams
- Deployment Diagrams

A model is an abstraction of a phenomenon for the purpose of understanding the system prior to building or modifying it. It is a simplified representation of reality. Modelling provides a means for communicating ideas in an easy to understand manner in addition to accommodating a system's complexity. The characteristics of simplification and representation are difficult to achieve in the real world since they frequently contradict each other.

Most modelling techniques used for analysis and design involve graphic languages which are a set of symbols. These symbols are used according to certain rules of the methodology for communicating the complex relationships of information more clearly than descriptive text. The goal of CASE tools is to assist in using these graphic languages, along with their associated methodologies.

Modelling is used during many of the phases of the SDLC. Objectory is built around several different models i.e., use-case model, domain object model, analysis object model, implementation model and test model.

Modelling is an iterative process; as the development of model progresses from analysis to implementation stage, more details are added but it remains the same.

Models can represent static or dynamic situations. Each representation has different implications with regard to organization and representation of knowledge.

Developing a model for a software system prior to its construction or renovation is as essential as having a blueprint for a large building. Good models are essential for communicating the complexity of a system among project teams and to assure architectural soundness. As the complexity of systems increase, so does the importance of good modeling techniques. There are many additional factors of a project's success, but having a rigorous modeling language standard is one essential factor. A modeling language must include:

- *Model elements:* Fundamental modelling concepts and semantics.
- *Notation:* Visual rendering of model elements.
- *Guidelines:* Expression of usage within the trade.

In the face of increasingly complex systems, visualization and modeling become essential. The Unified Modelling Language (UML) is a well-defined and widely accepted response to that need. It is the visual modeling language of choice for building object-oriented and component based systems. The use of visual notation to represent or model a problem can provide several benefits. Those are clarity, familiarity, maintenance and simplification.

Turban cites the following advantages of modeling:

- Models make it easier to express complex ideas.
- Models reduce complexity.
- Models enhance and reinforce learning and training.
- The cost of modelling analysis is much lower than the cost of similar experimentation conducted with a real system.
- Manipulation of model is much easier than manipulating a real system.

Here are a few key ideas regarding modelling:

- A model is rarely correct when constructed for the first time.
- It is necessary to seek the opinion of others.
- It is essential to avoid excess model revisions as they can distort the essence of the model.

1. STATIC AND DYNAMIC MODELS

A static model can be viewed as a snapshot of a system's parameters at rest or at a specific point in time. Static models are needed to represent the structural or static aspect of a system. For example, a customer could have more than one account or an order could be aggregated from one or more line items. Static models assume stability and an absence of change in data over time. The UML class diagram is an example of a static model.

A dynamic model can be viewed as a collection of procedures or behaviours that, taken together, reflect the behaviour of a system over time. Dynamic relationships show how the business objects interact to perform tasks. For example, an object order interacts with another object, inventory to determine the availability of a given product.

A system can be described by first developing its static model, which is the structure of its objects and their relationships to each other. Dynamic modelling is most useful during the design and implementation phases of the system development. The UML interaction diagrams and activity models are examples of UML dynamic models.

Static models depicting classes, inheritance relationships and aggregation relationships are often the first diagrams that are created. Unfortunately, they are sometimes the *only* diagrams that are create. In fact, a static emphasis on object oriented design is inappropriate. Software design is about a dynamic behavior. Object oriented design is a technique used to separate and encapsulate behaviors.

The interplay that exists between the static and dynamic models is also more important. A static model cannot be proven accurate without associated dynamic models. Dynamic models, on the other hand, do not adequately represent considerations of structure and dependency. Thus, the designer must iterate between the two kinds of models, in order to converge on an acceptable solution.

2. UNIFIED MODELING LANGUAGE

The Unified Modeling Language (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, creating a "blueprint" for construction.

The Unified Modeling Language (UML) is becoming a standardized modeling notation for expressing object-oriented models and designs. The UML is based on an intuitive and easy to understand diagrammatic notation. More and more, software developers are using UML to model their software in the early stages of software development. Recent research shows that software errors are most likely to be introduced during the requirement analysis and design stage and these errors can have a lasting impact on the reliability, cost and safety of a system. Furthermore, requirement errors are more costly to fix during later stages of the software lifecycle than during the requirements stage.

A UML model usually includes both static and dynamic aspects so as to completely model a real application. The UML is a graphical language with sets of rules and semantics. The rules and semantics of a model are expressed in English, in a form known as Object Constraint Language (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system. In general, the static aspect of a model can be represented by the static diagrams in UML, such as class diagrams, together with some constraints written in the Object Constraint Language (OCL); the dynamic aspect of a model can be given by the UML dynamic diagrams such as state machine diagrams or activity diagrams. We think that any tool supporting the validation of a UML model should include static and dynamic validation.

2.1 Benefits of UML

Following are the advantages of UML:

- UML is a ready-to-use and expressive visual modeling language that can be used to develop and exchange appropriate models.
- UML provides extensibility and specialization mechanisms to extend the core concepts.
- It is independent of particular programming languages and development processes.
- Encourages the growth of OO tools market.
- Supports higher-level development concepts.
- Integrates best practices and methodologies.

2.2 Scope of UML

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing and documenting the artifacts of a software-intensive system.

It is to be noted that the UML combines the concepts of Booch, OMT and OOSE. The result is a single, common and widely used modeling language for users of these and other methods. Second, Unified Modeling Language expands the scope of existing methods. As an example, the UML authors targeted the modeling of concurrent, distributed systems to assure that the UML adequately addresses these domains. Third, the Unified Modeling Language focuses on a standard modeling language, not a standard process. Although the UML must be applied in the context of a process, it is our experience that different organizations and problem domains require different processes.

Therefore, the efforts concentrated first on a common meta model (which unifies semantics) and second on a common notation (which provides a human rendering of these semantics). The UML authors promote a development process that is use-case driven, architecture centric, and iterative and incremental. The UML specifies a modeling language that incorporates the object-oriented community's consensus on core modeling concepts. It allows deviations to be expressed in terms of its extension mechanisms. The Unified Modeling Language provides the following:

- Semantics and notation to address a wide variety of contemporary modeling issues in a direct and economical fashion.
- Semantics to address certain expected future modeling issues, specifically related to component technology, distributed computing, frameworks and executability.
- Extensibility mechanisms so that individual projects can extend the meta model for their application at low cost. We do not want users to directly change the UML meta model.
- Extensibility mechanisms so that future modeling approaches could be grown on top of the UML.
- Semantics to facilitate model interchange among a variety of tools.
- Semantics to specify the interface to repositories for the sharing and storage of model artifacts.

2.3 UML Diagrams

At the core of the UML are its nine kinds of modeling diagrams. They are:

i. Class diagrams
ii. Object diagrams
iii. Use case diagrams
iv. Sequence diagrams
v. Collaboration diagrams
vi. Statechart diagrams
vii. Activity diagrams
viii. Component diagrams
ix. Deployment diagrams.

UML diagrams are divided into three categories: four diagrams represent static application structure, three diagrams represent general types of behavior, and two diagrams represent different aspects of interactions i.e.,

- *Structure diagrams* include the Class Diagram, Object Diagram, Component Diagram and Deployment Diagram.
- *Behavior diagrams* include the Use Case Diagram (used by some methodologies during requirements gathering), Activity Diagram, and State Machine Diagram.
- *Interaction diagrams*, all derived from the more general Behavior Diagram, include the Sequence Diagram and Collaboration Diagram.

UML consists of Use Case diagrams which pictorially show a sequence of actions and represent a functional requirement. A Use Case also contains a textual description, which describes the preconditions and the main flow. UML uses class diagrams which show the static structure of the system. Class diagrams define the class in terms of its attributes and operations, but also show association, subclassing, and aggregation. Interfaces are used in UML to reduce coupling. UML also supports division of a system into subsystems called packages, which constrains the effect of changes and limits the coupling. Deployment diagrams are used with package diagrams to show the physical relationship among the components and where they are physically located.

Dynamic behavior is shown in UML by using state diagrams, interaction diagrams, and activity diagrams. State diagrams describe the behavior of objects in terms of how they change from one state to another. Interaction diagrams describe how groups of objects interact. Some common interaction diagrams are sequence diagrams, which show the order and interaction of a sequential set of steps, and collaboration diagrams, which show a sequence of messages. An activity diagram describes parallel processing and shows the flow from one activity to another.

2.4 UML Validation

First, static validation can be used to check whether a model is syntactically valid, i.e., whether the model satisfies the UML meta-model including the well-formed rules given by OCL. On the other hand, as the application becomes more complicated, it is harder for a developer to find whether some state, represented by an object diagram, is included in the model which (s)he is developing. The second function for the static validation is that it can help a developer check whether his/her model includes some related snapshots or not.

After designing a static structure of a model, a developer can specify dynamic behavior for a class and this kind of behavior can be represented by UML dynamic diagrams such as state chart diagrams. Dynamic validation is used to check whether the dynamic aspect of a model satisfies some important properties such as safety.

There are not many research tools available to support either static or dynamic validation. One of the reasons most tools do not support model validation is the lack of formal semantics of UML and OCL.

Generally, research work to support UML model validation usually includes two steps. First, researchers present a formal semantics for a diagram or language in which they will work; and then, according to the formal semantics, they either translate the diagram or language into some language supporting the validation or use some programming language to execute the diagram or language. One of the problems in the above tools is that the researchers have not given a proof of correctness for these tools, although the validation model they assume is the same as the semantic model.

UML represents a unification of the concepts and notations. The goal is for UML to become a common language for creating models of object oriented computer software. UML comprises of two major components: a Meta-model and a notation.

3. THE META-MODEL

UML is unique in that it has a standard data representation. This representation is called the Meta model. The meta-model is a description of UML in UML i.e., a meta-model is a model of modeling elements. The purpose of the UML meta-model is to provide a single, common, and definitive statement of the syntax and semantics of the elements of the UML. The meta-model provides a means to connect different UML diagrams. The connection between the different diagrams is very important. The UML attempts to make these couplings more explicit through defining the underlying model (meta-model) while imposing no methodology.

The presence of meta-model has made it possible for its developers to agree on semantics and how those semantics would be best rendered. This is an important step forward, since it can assure consistency among diagrams. The meta-model can serve as a means to exchange data between different CASE tools. The meta-model has made it possible for a team to explore ways to make the modelling language.

3.1 The Notation

The UML notation is rich and full bodied. It is comprises of two major subdivisions. The first one is a notation for modeling the static elements of a design such as classes, attributes and relationships. The second one is a notation for modeling the dynamic elements of a design such as objects, messages and finite state machines.

4. USE-CASE DIAGRAMS

"The Use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases". The Use case diagram shows which actors interact with each use case".

The use case model is based on Jacobson's use cases with some minor changes to better match UML's overall approach. Use Case scenarios are used in process modelling and analysis of requirements stages of UML. Use Cases are used in the analysis phase of software development to articulate the high-level requirements of the system.

A Use Case Diagram is a diagram that helps system analys to discover the requirements of the target system from the user's perspective. A use case diagram:

- Describes the behaviour of a system from a user's standpoint.
- Provides functional description of a system and its major processes.
- Provides graphic description of the users of a system and what kind of interactions to expect within that system.
- Displays the details of the processes that occur within the application area.

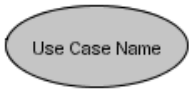
A Use case diagram displays the relationship among actors and use cases. Hence, the basic components of Use Case diagrams are the Actor, the Use Case, and the Association.

4.1 Actor

An Actor, as mentioned, is a user of the system, and is depicted using a stick figure. The role of the user is written beneath the icon. Actors are not limited to humans. If a system communicates with another application, and expects input or delivers output, then that application can also be considered an actor. In a banking application, a customer entity represents an actor. Similarly, the person who provides service at the counter is also an actor. The graphical notation for actor is given below:

4.2 Use Case

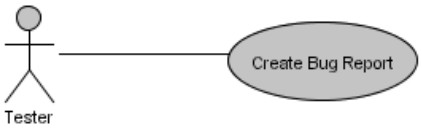
Use case is a collection of possible sequences of interactions between the system under discussion and its Users (or Actors), relating to a particular goal. The collection of Use Cases should define all systems' behavior relevant to the actors to assure them that their goals will be carried out properly. Any system behavior that is irrelevant to the actors should not be included in the use cases. These are represented by ellipses with actions written inside. The graphical notation for use case is as follows:



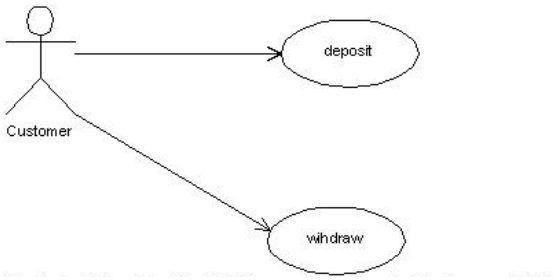
4.3 Association

Associations are used to link Actors with Use Cases and indicate that an Actor participates in the Use Case in some form. Associations are depicted by a line connecting the Actor and the Use Case. The graphical notation for the association in use case diagrams is as follows:

The following image shows how these three basic elements work together to form a use case diagram:



For a banking application it might be: deposit money and withdraw money. The use case diagram is given below:



it shows the relationships between actors and use cases.

4.4 Communicates

The participation of an actor in a use case is shown by connecting the actor symbol to the use case symbol by a solid path. The actor is said to 'communicate' with the use case. This is only the relation between actor and use cases.

4.5 Extends

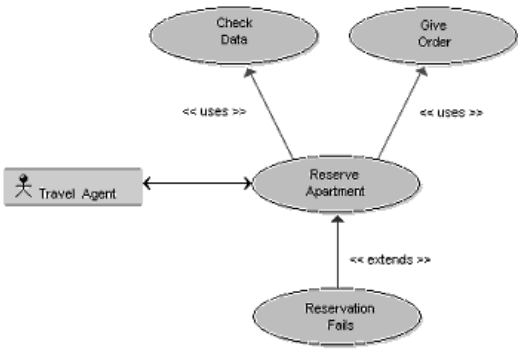
Extends shows the relationships between use cases. Relationship between use case A and use case B indicates that an instance of use case B may include (subject to specific conditions specified in the extension) the behavior specified by A. An 'extends' relationship between use cases is shown by a generalization arrow from the use case providing the extension to the base use case. The arrow is labeled with the stereotype «extends».

4.6 Uses

A uses relationship from use case A to use case B indicates that an instance of the use case A will also include the behavior as specified by B. A 'uses' relationship between use cases is shown by a generalization arrow from the use case doing the use to the use case being used. The arrow is labeled with the stereotype «uses».The diagram below shows use case for travel agent reserving apartment scenario.

Figure 1 below shows all the relationships between use cases and actors.

Figure 1



5. ACTIVITY DIAGRAMS

The easiest way to visualize an Activity diagram is to think of a flowchart of a code. The flowchart is used to depict the business logic flow and the events that cause decisions and actions in the code to take place.

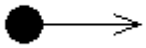
An activity diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation.

A State diagram shows the different states an object is in during the lifecycle of its existence in the system and the transitions in the states of the objects. These transitions depict the activities causing transitions, and are shown by arrows. An Activity diagram describes transitions and activities that cause changes in the object states.

Let us take a look at the building blocks of an Activity diagram. An Activity diagram consists of the following behavioral elements:

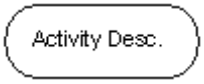
5.1 Initial Activity

This shows the starting point or first activity of the flow. It is denoted by a solid circle followed by an arrow. This is similar to the notation used for Initial State,



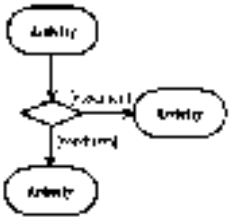
5.2 Action states

Action states represent the non-interruptible actions of objects. It is represented by a rectangle with rounded (almost oval) corners.



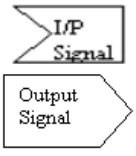
5.3 Branching

Similar to flowcharts, a logic where a decision is to be made is depicted by a diamond. A diamond represents a decision with alternate paths. The outgoing alternates should be labeled with a condition or guard expression.



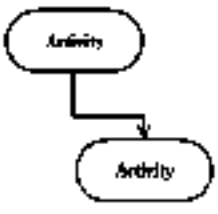
5.4 Signal

When an activity sends or receives a message, that activity is called a signal. Signals are of two types: Input signal (Message receiving activity) shown by a concave polygon and Output signal (Message sending activity) shown by a convex polygon.



5.5 Action Flow

Action flow arrows illustrate the relationships among action states.

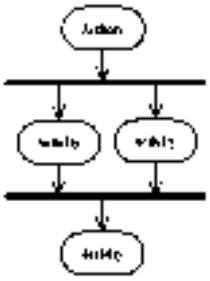


5.6 Object Flow

Object flow refers to the creation and modification of objects by activities. An object flow arrow from an action to an object means that the action creates or influences the object. An object flow arrow from an object to an action indicates that the action state uses the object.

5.7 Concurrent Activities

Some activities occur simultaneously or in parallel. Such activities are called concurrent activities. This is also called as synchronization. This is represented by a horizontal split (thick dark line) and the two concurrent activities next to each other, and the horizontal line is again drawn to show the end of the parallel activity as given below:



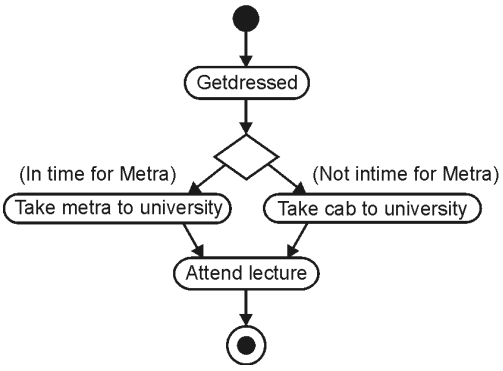
5.8 Final State

An arrow pointing to a filled circle nested inside another circle represents the final action state as shown below:

5.9 Creating an Activity Diagram

Below given is an activity diagram to depict an example of attending a course lecture.

Figure 2: Activity Diagram



- The first activity is to get dressed to leave for the lecture.
- A decision then has to be made, depending on the time available for the lecture to start and the timings of the public trains (metro). If there is sufficient time to catch the train, then take the train; else, flag down a cab to the University.
- The final activity is to actually attend the lecture, after which the Activity diagram terminates.

5.10 Advantages and Disadvantages

The activity diagram focuses on activities; in this sense, it is like a flow chart supporting compound decisions. However, it differs from a flow chart by explicitly supporting parallel activities and their synchronization. Unlike EPC, the starting point eliminates the need to traverse through the entire process model for identifying the start of the process or triggering event. The back flow of the activities can also be the represented.

The biggest disadvantage of activity diagrams is that they do not make explicit which objects execute which activities, and the way that the messaging works between them. Labeling of each activity with the responsible object can be done. Often, it is useful to draw an activity diagram early in the modeling of a process, to understand the overall process. Then, interaction diagrams can be used to allocate activities to classes.

6. INTERACTION DIAGRAMS

An instantiated use case shows a particular series of interactions among objects in a single execution of a system; it describes a single history without conditionality. A specific pattern of message exchange to accomplish a specific purpose is called an interaction. A scenario is a single execution history of an interaction.

An interaction is a behavioral specification that comprises a sequence of message exchanges among a set of objects within a context to accomplish a specific purpose, such as the implementation of an operation. To specify an interaction, it is first necessary to specify a context, that is, to establish the objects that interact and their relationships. Then, the possible interaction sequences are specified. These can be specified in a single description containing conditionals (branches or conditional signals), or they can be specified by supplying multiple descriptions, each describing a particular path through the possible execution paths.

Interaction diagrams are models that describe how a group of objects collaborate in some behavior – typically a single use-case. The diagrams show a number of example objects and the messages that are passed between these objects within the use-case. Interaction diagrams do not give an in depth representation of the behavior. To see what a specific object is doing for several use cases, the **State Diagram** is used. To see a particular behavior over many use cases or threads the **Activity Diagram** is used.

Sequence diagrams and collaboration diagrams, can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur. Collaboration diagrams demonstrate how objects are statically connected. Both diagrams are relatively simple to draw and contain similar elements.

7. SEQUENCE DIAGRAMS

Sequence diagrams show the interactions between classes to achieve a result. Because UML is designed for object-oriented programming, these communications between classes are known as messages. It shows the objects and the messages that are passed between these objects in the use case. The sequence diagram lists objects horizontally and time vertically, and models these messages over time.

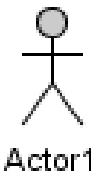
Sequence diagrams show object interactions arranged in a time sequence. The flow of Events can be used to determine what objects and interactions will be needed to accomplish the functionality specified by the flow of events. In a Sequence diagram, classes and actors are listed as columns, with vertical lifelines indicating the lifetime of the object over time.

7.1 Object

Objects are instances of classes and are arranged horizontally. The pictorial representation for an Object is a class (a rectangle) with the name prefixed by the object name (optional) and a semi-colon. The following symbol shows the object.

7.2 Actor

Actors can also communicate with objects, and hence, they too can be listed as a column. An Actor is modeled using the ubiquitous symbol, the stick figure.



7.3 Lifeline

The Lifeline identifies the existence of the object over time. The notation for a Lifeline is a vertical dotted line extending from an object. The symbol for life line is given below:

7.4 Activation

Activations, modeled as rectangular boxes on the lifeline, indicate when the object is performing an action. The symbol is as follows:

7.5 Message

Messages, modeled as horizontal arrows between Activations, indicate the communications between objects. The symbol for message is as follows:

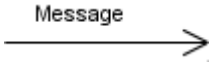


Figure 3 is an example sequence diagram, using the default named objects.

Figure 3

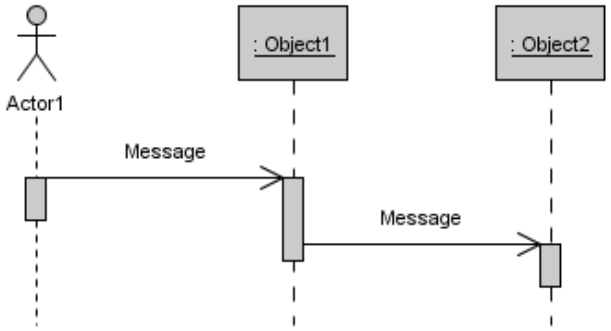
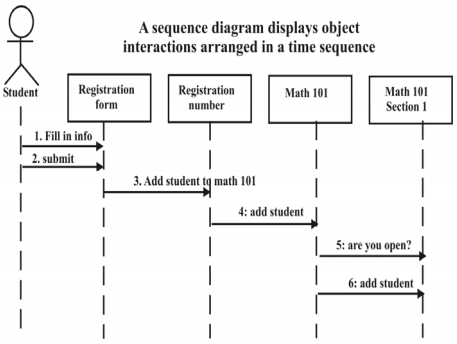


Figure 4 shows how a student successfully gets admission to a course. The student fills in some information and submits the form. The form then indicates the student has been enrolled in math 101. In this case, Section 1 is open, so Math 101 adds the student to Section 1.

Figure 4



Sequence diagrams are great tools in the beginning because they show everyone including the customer step-by-step what has to happen. One of the benefits of user-friendly computers with these types of diagrams is that every line coming from an actor that represents a person can be used to highlight testable user interface requirements.

Sequence diagrams are good for showing what's going on, for deriving. out requirements and for working with customers.

8. COLLABORATION DIAGRAMS

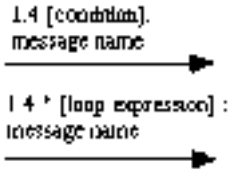
A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence and use case diagrams describing both the static structure and dynamic behavior of a system. This type of diagram is a cross between an object diagram and a sequence diagram. Unlike the sequence diagram, which models the interaction in a column and row type format, the collaboration diagram uses the free-form arrangement of objects as found in an object diagram. This makes it easier to see all interactions involving a particular object.

In order to maintain the ordering of messages in such a free-form diagram, messages are labeled with a chronological number. Reading a collaboration diagram involves starting at message 1.0, and following the messages from one object to another.

8.1 Messages

Messages, modeled as arrows between objects, and labeled with an ordering number, indicate the communications between objects.

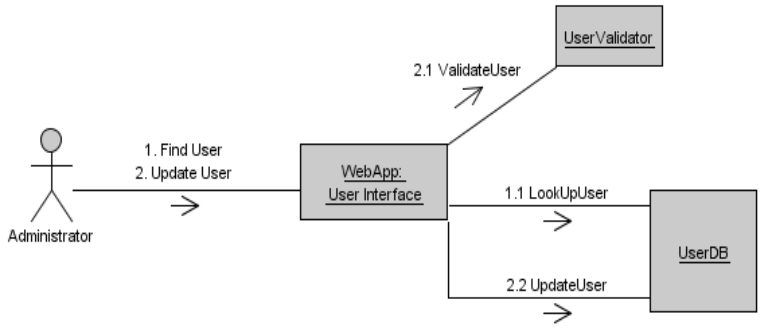
Unlike sequence diagrams, collaboration diagrams do not have an explicit way to denote time and instead number the messages in order of execution. Sequence numbering can become nested by using the Dewey decimal system. For example, nested messages under the first message are labeled 1.1, 1.2, 1.3, and so on. The condition for a message is usually placed in square brackets immediately following the sequence number. A is used after the sequence number to indicate a loop.



Here is an example (figure 5) of an administrator using a web application to manage a user account.

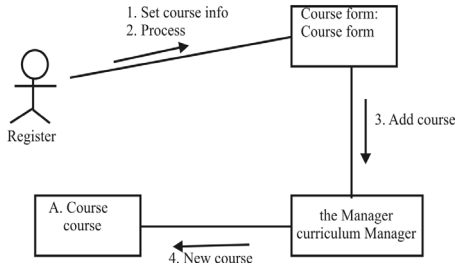
1. Find User
    1.1 LookUp User
2. Update User
    2.1 Validate User
    2.2 Update User.

Figure 5



The collaboration diagrams are shown based on the links between the objects.

Figure 6: Collaboration Diagram



A collaboration diagram displays object interactions organized around objects and their links to one another. The benefit of collaboration diagrams is that all of the messages that go between two objects for a particular use case or scenario

can be seen. Especially, in the case of a big, long scenario where the real estate is smaller, it is easier to see these messages on a collaboration diagram. A collaboration diagram is just a different view of a scenario. It is also called as "interaction diagram".

9. CLASS DIAGRAMS

Class diagrams are the essential elements of almost every object oriented method, including UML. They describe the static structure of a system. A class diagram is a graphic presentation of the static view which shows a collection of declarative (static) model elements such as classes, types, and their contents and relationships. Classes are arranged in hierarchies sharing common structure and behavior, and are associated with other classes.

The purpose of a class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relation-ships with other classes. The UML class diagram can depict all these things quite easily. The fundamental element of the class diagram is an icon that represents a class.

For example, thousands of students attend the university; you would only model one class, called *Student*, which would represent the entire collection of students. A Student class represents student entities in a system. The Student class encapsulates student information such as student id #, student name, and so forth. Student id, student name, and so on are the attributes of the Student class. The Student class also exposes functionality to other classes by using methods such as getStudentName(), getStudentId(), and the like. Let us take a look at how a class is represented in a class diagram.

The UML modeling elements found in class diagrams include:

- Classes and their structure and behavior.
- Association, aggregation, dependency and inheritance relationships.
- Multiplicity and navigation indicators.
- Role names.

A class is represented by a **rectangle**. The following diagram shows a typical class in a class diagram:

Class
Attribute
operation ( )

A class icon is simply a rectangle divided into three compartments:

- The top most compartment contains the name of the class. The class name typically has the first alphabet capitalized. If a class has more than one word, and capitalized then the first alphabet of both words is joined together.
- The middle compartment contains a list of attributes (member variables).

  The syntax is **attribute: Type = "default value (if any)"**

  e.g. studentid : int

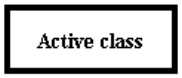
  Studentname : string = "ICFAI"
- The bottom compartment contains a list of operations (member functions).

  The syntax is **methodname(list of parameters (if any)) : return type**

  e.g., getStudentId(studentid) : int

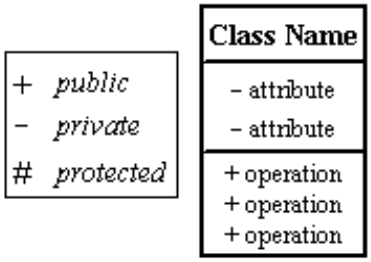
  In many diagrams, the bottom two compartments are omitted. Even when they are present, they typically do not show all attributes and operations. The goal is to show only those attributes and operations that are useful for the particular diagram.

Active classes initiate and control the flow of activity, while passive classes store data and serve other classes. Illustrate active classes with a thicker border as given below.



9.1 Visibility

Use visibility markers to signify who can access the information contained within a class. Private visibility hides information from anything outside the class partition. Public visibility allows all other classes to view the marked information. Protected visibility allows child classes to access information they inherited from a parent class.



The format for attributes visibility is:

Visibility attribute-name: type = default Value

e.g., + studentid: int

In object oriented design, it is generally preferred to keep most attributes private as the accessor methods allow you to control access to the data. The most common exception to this preference is constants.

The format for operations visibility is:

Visibility operation-name (parameters): type

e.g., + getstudentid(studentid) : int

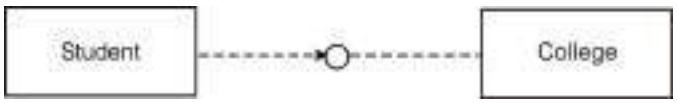
10. OBJECT DIAGRAMS

An object diagram shows the existence of objects and their relationships in the logical view of a system and; also traces the execution of a scenario. It is a static diagram. Notation is same as the class diagram. Class diagrams can contain objects and, so a class diagram with objects and no classes is an object diagram.

10.1 Interfaces

There are classes that have nothing but pure virtual functions. In Java such entities are not classes at all; they are a special language element called interface. UML has followed the Java example and has created some special syntactic elements for such entities.

Interfaces are very similar to abstract classes with the exception that they do not have any attributes. As well, unlike a class, all of the operations in an interface have no implementation. The UML notation for an interface is a small circle with the name of the interface connected to the class. A class that requires the operations in the interface may be attached to the circle by a dashed arrow. The dependent class is not required to actually use all of the operations. For example, a Student object may interact with the College object to get the Examination id; this relationship is depicted in the following figure with UML class interface notation.



Every class diagram has classes, associations and multiplicities. Navigability and roles are optional items placed in a diagram to provide clarity.

Our class diagram has three kinds of relationships:

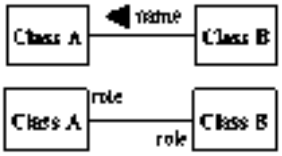
a Association.

b. Aggregation.

c. Generalization.

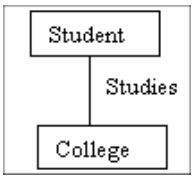
10.1.1 ASSOCIATION

Associations represent static relationships between classes. Association is a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes. An association has two ends. An end may have a role name to clarify the nature of the association. Association names are placed above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship. Place roles near the end of an association. Roles represent the way the two classes see each other.



For example: A “student studies in a college” association can be shown as:

Figure 7



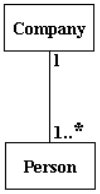
Multiplicity (Cardinality)

The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. A multiplicity specification is shown as a text string sequence of integer intervals; where an interval represents a range of integers in this format. The terms lower bound and upper bound are integer values, specifying the range of integers (from lower bound to the upper bound). The star character (*) may be used for the upper bound, denoting an unlimited upper bound. If a single integer value is specified, then the integer contains single values as shown below.

Multiplicities	Meaning
0..1	Zero or one instance. The notation n . . m indicates n to m instances.
0..* or *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

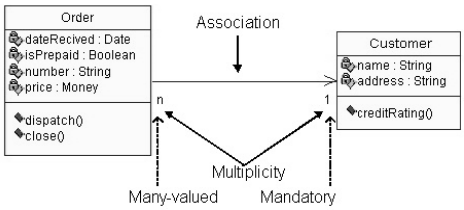
For example, one company will have one or more employees, but each employee works for one company only.

Figure 8



For example, the class Order is associated with the class Customer. The multiplicity of the association denotes the number of objects that can participate in the relationship.1. For example, an Order object can be associated with only one customer, but a customer can be associated with many orders.

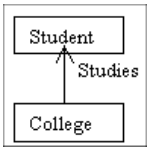
Figure 9



Directed Association

Association between classes is bi-directional by default. You can define the flow of the association by using a directed association. The arrowhead identifies the container-contained relationship. For example,

Figure 10



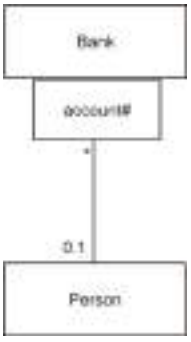
Reflexive Association

An example of this kind of relation is when a class has a variety of responsibilities. For example, an employee of a college can be a professor, a housekeeper, or an administrative assistant.

Qualifier

A qualifier is an association attribute. For example, a Person object may be associated to a Bank object. An attribute of this association is the account#. The account is the qualifier of this association. A qualifier is shown as a small rectangle attached to the end of an association path, between the final path segment and the symbol of the class to which it connects. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle usually is smaller than the attached class rectangle. It is shown in figure 11.

Figure 11

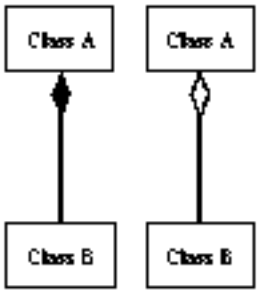


10.1.2 COMPOSITION AND AGGREGATION

An aggregation is a form of association, in which one class belongs to a collection. An aggregation has a diamond – end pointing to the part containing the whole. Composition is a special type of aggregation that denotes a strong ownership between Class A, the whole, and Class B, its part. A composition is illustrated with a filled diamond.

Use a hollow diamond to represent a simple aggregation relationship, in which the "whole" class plays a more important role than the "part" class, but the two classes are not dependent on each other. The diamond end in both a composition and aggregation relationship points toward the "whole" class or the aggregate.

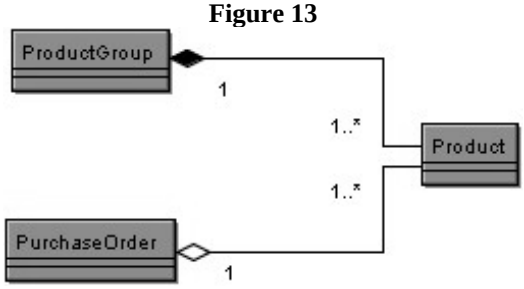
Figure 12



A stronger form of aggregation – a composite aggregation – is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time. If the parent of a composite aggregation is deleted, usually all of its parts are deleted with it; however, a part can be individually removed from a composition without having to delete the entire composition. Compositions are transitive, asymmetric relationships and can be recursive.

Figure 13 shows an aggregation association and a composition association. The composition association is represented by a solid diamond. It is said that Product Group is composed of Products. This means, if a Product Group is destroyed, the Products within the group are destroyed as well. The aggregation association is represented by the hollow diamond. Purchase Order is an aggregate of Products. If a Purchase Order is destroyed, the Products still exist.

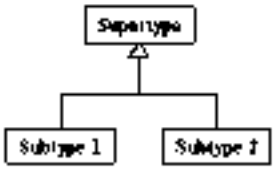
Figure 13



10.1.3 GENERALIZATION

Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another. An inheritance link indicating one class is a super class of the other. A generalization has a triangle pointing to the super class. For example, Honda is a type of car. So, the class Honda would have a generalization relationship with the class 'car'.

Figure 14

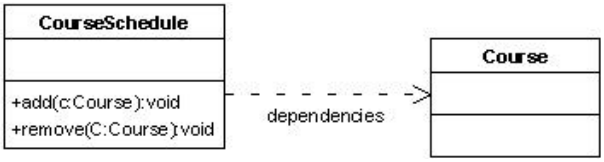


In real life coding examples, the difference between inheritance and aggregation can be confusing. For an aggregation relationship, the aggregate (the whole) can access only the PUBLIC functions of the part class. On the other hand, inheritance allows the inheriting class to access both the PUBLIC and PROTECTED functions of the superclass.

Dependencies

A dependency is a using relationship that states that a change in a specification of one thing may affect another thing that uses it. Dependencies are used in the context of classes to show that one class uses another class as an argument in its method's signature. Figure 15 below shows the dependency relation.

Figure 15



11. STATE CHART DIAGRAMS

State Chart diagrams are also referred to as State diagrams. State chart diagrams capture the life cycles of objects, sub-systems and systems. They indicate what states an object can have and how different events affect those states over time. State chart diagrams should be attached to classes that have clearly identifiable states and are governed by complex behavior. For example, the television can be in the OFF state and when the power button is pressed, the television goes into the ON state. Pressing the power button yet again causes a state transition from the ON state to the OFF state. In comparison to the other behavioral diagrams which model the interaction between multiple classes, State diagrams typically model the transitions within a single class.

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and tracks the different states of its objects through the system.

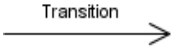
11.1 States

States represent situations during the life of an object. It is represented using a rectangle with rounded corners and the state name written inside. The State notation is:

We can also say that a state is a condition during the life of an object during which it satisfies some condition(s), performs some action(s), or waits for some event(s). The state changes when the object receives some event and; the object is said to undergo a state transition. The state of an object depends on its attribute values and links to other objects. An event is something that takes place at a certain point in time.

11.2 Transition

A Transition marks the changing of the object State caused by an event i.e., which shows the possible changes of state. The notation for a Transition is an arrow, with the Event Name written above, below, or alongside the arrow.



11.3 Initial State

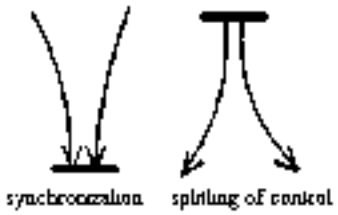
The Initial State is the state of an object before any transitions. For objects, this could be the state when instantiated. The Initial State is marked, using a solid circle followed by an arrow representing the object's initial state. Only one initial state is allowed on a diagram.

11.4 Final State

End States mark the destruction of the object whose state is being modeled such states are represented using an arrow pointing to a filled circle nested inside another circle represents the object's final state.

11.5 Synchronization and Splitting of Control

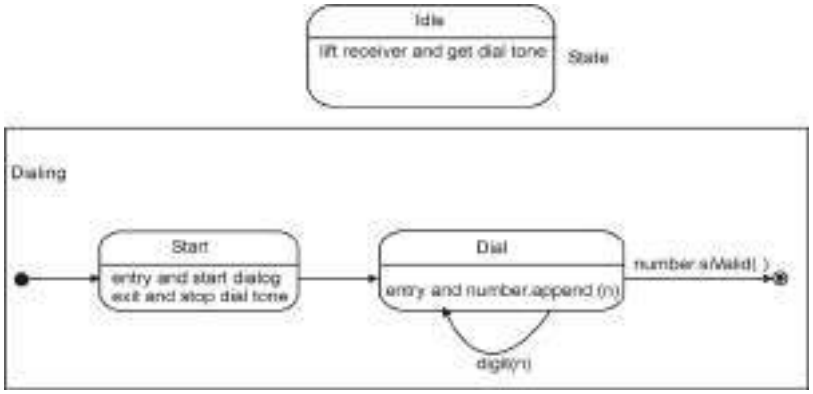
A short heavy bar with two transitions entering it represents a synchronization of control. A short heavy bar with two transitions leaving it represents a splitting of control that creates multiple states. This is shown below.



Some examples of events are a customer places an order, a student registers for a class, a person applies for a loan, and a company hires a new employee. For the purpose of modeling, an event is considered to be instantaneous. A state, on the other hand, spans a period of time. An object remains in a particular state for some time before transitioning to another state. For example, an Employee object might be in the Part-time state (as specified in its employment-status attribute) for a few months, before transitioning to a Full-time state, based on a recommendation from the manager (an event).

Figure16 below depicts the dialing state, which consists of start and dial states:

Figure 16



12. IMPLEMENTATION DIAGRAMS

A Use Case is a formal description of the functionality the system will have when constructed. An implementation diagram is typically associated with a Use Case to document what design elements (e.g., components and classes) will implement the Use Case functionality in the new system. This provides a high level of traceability for the system designer, the customer and the team that will actually build the system. UML defines two implementation diagrams: To show the relationship between the software components that make up a system (the *component diagram*) and the relationship between the software and the hardware on which it is deployed at run-time (the *deployment diagram*).

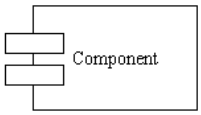
A component diagram is an implementation diagram that shows the structure of the code itself. It is used to know about compiler and run-time dependencies between software components, such as source code files. A Deployment diagram is an implementation diagram that shows the structure of a run-time system. This diagram helps us know about the physical relationship between software and hardware components and the distribution of components to processing nodes.

13. COMPONENT DIAGRAMS

A component diagram describes the organization of the physical components in a system. It depicts the components that compose an application, system or enterprise. The components, their interrelationships, interactions and their public interfaces are depicted.

13.1 Component

Components represent the physical packaging of a module of code. It is represented as a rectangle with tabs.

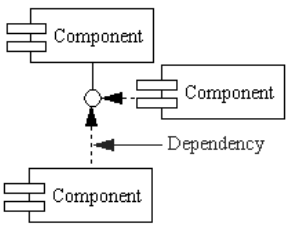


13.2 Interface

An interface describes a group of operations used or created by components. It is graphically shown as:

13.3 Dependencies

The dependencies between the components show how changes made to one component may affect the other components in the system. Dependencies in a component diagram are represented by a dashed line between two or more components.

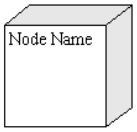


14. DEPLOYMENT DIAGRAM

Deployment diagrams serve to model the hardware used in system implementations and the associations between those components. The elements used in deployment diagrams are nodes (shown as a cube), components (shown as a rectangular box, with two rectangles protruding from the left side) and associations.

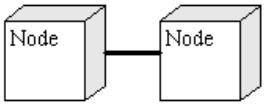
14.1 Node

A node is a physical resource that executes code components i.e., a node usually represents a piece of hardware in the system. It is represented as:



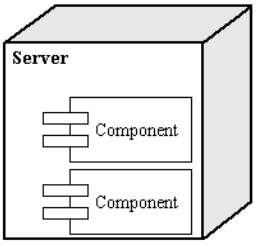
14.2 Association

Association refers to a physical connection between nodes such as Ethernet.



14.3 Components and Nodes

Place components inside the node that deploys them.



The above deployment diagram shows the hardware used in a small office network. The application server (node) is connected to the database server (node) and the database client (component) is installed on the application server. The workstation is connected (association) to the application server and to a printer.

SUMMARY

- A model is a simplified representation of reality – simplified because reality is too complex or large and much of the complexity actually is irrelevant to the problem being described or solved.
- The unified modeling language was developed by Booch, Jacobson, and Rumbaugh. The UML encompasses the unification of their modeling notations.
- The UML class diagram is the main static structure analysis diagram for the system. It represents the class structure of a system with relationships between classes and inheritance structure. The class diagrams are developed through use-case, sequence, and collaboration diagrams.
- The use-case diagram captures information on how the system or business works or how one wishes it to work. It is a scenario-building approach in which one models the processes of the system. It is an excellent way to learn object-oriented analysis of the system.

- UML sequence diagram is used for dynamic modeling, where objects are represented as vertical lines and message passed back and forth between the objects are modeled by horizontal vectors between the objects.
- The UML collaboration diagram is an alternative view of the sequence diagram, showing in a scenario how objects interrelate with one another.
- State chart diagrams, another form of dynamic modeling, focus on the vents occurring within a single object as it responds to messages. An and activity diagram is used to model an entire business process. Thus, an activity model can represent several different classes.
- Implementation diagrams show the implementation phase of systems development, such as the source code and run-time implementation structures. The two types of implementation diagrams are component diagrams, which show the structure of the code itself, and deployment diagrams, which show the structure of the run time system.
- Stereotypes represent a built-in extensibility mechanism of the UML. User-defined extensions of the UML are enabled through the use of stereotypes and constraints.
- UML graphical notations can be used not only to describe the system's components but also to describe a model itself that is known as metamodel. It is a model of modeling elements. The purpose of the UML metamodel is to provide a single, common, and definitive statement of the syntax and semantics of the elements of UML.

Chapter VII

Object-Oriented Analysis

After reading this chapter, you will be conversant with:

- Use Case Model
- Developing Effective Documentation
- Approaches for Identifying Classes
- Identifying Attributes and Methods
- Defining Attributes by Analyzing Use Cases and Other UML Diagrams

In software development, analysis is the process of studying and defining the problem to be resolved. It involves discovering the requirements that the system must perform, the underlying assumptions with which it must fit, and the criteria by which its performance will be judged as success or failure.

Object-Oriented Analysis (OOA), is the process of defining the problem in terms of objects: real-world objects with which the system must interact and candidate software objects used to explore various solution alternatives. The natural fit of programming objects to real-world objects has a big impact. It is possible to define all the real-world objects in terms of their classes, attributes and operations. Its emphasis is on finding and describing the objects or concepts of the problem domain. Focus should be on the point that the system must do the right thing.

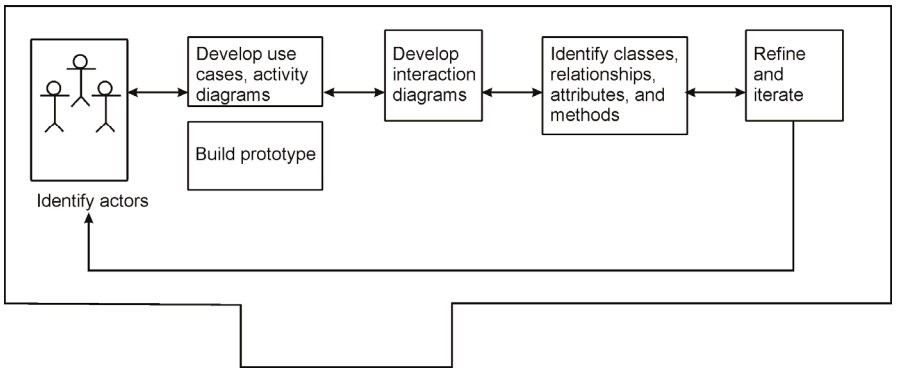
The object-oriented analysis phase of the unified approach uses actors and use cases to describe the system from the users' perspective. The use cases identified will be involved throughout the development process.

The OOA process consists of the following steps:

1. Identify the actors:
   - Who is using the system?
   - Or, in the case of a new system, who will be using the system?
2. Develop a simple business process model using UML activity diagram.
3. Develop the use cases:
   - What are the users doing with the system?
   - Or, in case of a new system, what the users will be doing with the system?
   - Use case provides us with comprehensive documentation of the system under study.
4. Prepare interaction diagrams:
   - Determine the sequence.
   - Develop collaboration diagrams.
5. Classification – develop a static UML class diagram:
   - Identify classes.
   - Identify relationships.
   - Identify attributes.
   - Identify methods.
6. Iterate and refine: if needed, repeat the preceding steps.

The above process steps are shown in figure 1 below:

Figure 1: OOA Process



Object-Oriented analysis contains the following activities:

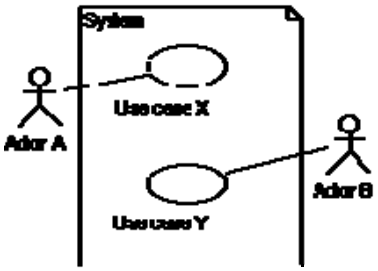
- **Identifying Objects:** Objects must always exist, so that the system is stable,
- **Organizing the Objects:** The objects that are identified are classified so that similar objects can later be defined in the same class,
- **Identifying Relationships between Objects:** This helps to determine inputs and outputs of an object,
- **Defining Operations of the Objects:** Processing of data within an object, and
- **Defining Objects Internally:** Information held by the objects.

1. USE CASE MODEL

A Use Case represents a discrete unit of interaction between a user (human or machine) and a system. A Use Case is a single unit of meaningful work. At the heart of the Unified Modeling Language (UML), are Use Cases. Use cases form the basis for the interaction of the system with the outside world (users, other systems, etc.). Use cases are designed to capture, via a combination of structured text and graphics, the functional requirements of a system. Use cases are usually described in a textual document that accompanies a use case diagram; the combination of these use case diagrams and their supporting documentation is known as a Use Case Model i.e., the Use Case Model describes the proposed functionality of the new system.

The use case model includes the actors, the system, and the use cases themselves. The set of functionality of a given system is determined through the study of the functional requirements of each actor, expressed in the use cases in the form of 'families' of interactions. Actors are represented by little stick people who trigger the use cases, which are represented as ellipses contained within the system.

Figure 2: Use Case Model



1.1 Development of the Use-case Model

Following are the steps in the development of the use-case model:

- Defining the System.
- Finding Actors and Uses Cases.
- Use Case Descriptions.
- Defining Relationships between Use Cases.
- Verifying and Validating the Model.

1.1.1 DEFINING THE SYSTEM

The formal specification of a use case includes:

1. **Requirements:** These are the formal functional requirements that a Use Case must provide to the end-user. These correspond to the functional specifications found in structured methodologies. A requirement is a contract that the Use Case will perform some action or provide some value to the system.

2. **Constraints:** These are the formal rules and limitations that a Use Case operates under, and includes pre, post and invariant conditions. A pre-condition specifies what must have already occurred or be in place before the Use Case may start. A post-condition documents what will be true once the Use Case is complete. An invariant condition specifies what will be true throughout the time the Use Case operates.
3. **Scenarios:** Scenarios are formal descriptions of the flow of events that occur during a Use Case instance. These are usually described in text and correspond to a textual representation of the Sequence Diagram.

1.1.2 ACTOR

An actor is someone or something that interacts with the system. The actor is a type (a class), not an instance. The actor represents a role, not an individual user of the system. Actors can be ranked. A primary actor is one that uses the primary functions of the system. A secondary actor is one that uses secondary functions of the system, – those functions that maintain the system, such as managing databases, communication, backups, and other administration tasks. The symbol for an actor is depicted below:

Actor

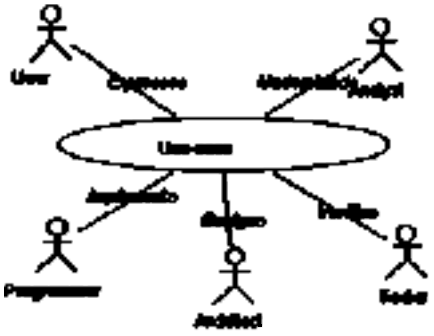
1.1.3 USE CASES

Use cases (figure 3) are determined by observing and specifying one actor after another and the interaction sequences (scenarios) from the user's standpoint. They are described in terms of the information exchanged and the way the system is used. A use case groups a family of usage scenarios according to a functional criterion. Use cases are abstractions of dialog between the actors and the system: they describe potential interactions without going into the details of each scenario.

Use cases must be seen as classes whose instances are the scenarios. Each time an actor interacts with the system, the use case instantiates a scenario. This scenario corresponds to the message flows exchanged by objects during the particular interaction that corresponds to the scenario. Analysis of requirements by use cases is very well complemented by an iterative and incremental approach.

The scope of use cases is much more in additon to defining of the requirements of the system. Indeed, use cases come into play throughout the lifecycle, from the specification stage to system testing, analysis, design, implementation, and documentation stages. From that standpoint, it is possible to navigate first towards the classes and objects that collaborate to satisfy a requirement, then towards the tests that verify whether the system performs its duties correctly.

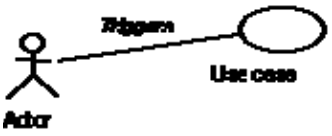
Figure 3



1.1.4 RELATIONSHIPS BETWEEN USE CASES

Use case diagrams represent use cases, actors, and relationships between use cases and actors. UML defines three types of links between actors and use cases as given below:

i. **They Communicate Relationship:** The participation of the actor is signaled by a solid line between the actor and the use case. This is the only relationship between actors and use cases.



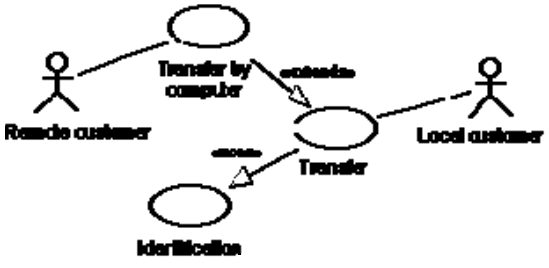
ii. **The Uses Relationship:** A uses relationship between use cases means that an instance of the source use case also includes the behavior described by the target use case. One Use Case may include the functionality of another as part of its normal processing. Generally, it is assumed that the included Use Case will be called every time the basic path is run. Representation of the uses relationship using a stereotyped generalization relationship is as follows:

An example may be to list a set of customer orders to choose from before modifying a selected order – in this case the <list orders> Use Case may be included every time the <modify order> Use Case is run. A Use Case may be included by one or more Use Cases; so, it helps to reduce duplication of functionality by factoring out common behavior into Use Cases that are re-used many times.

iii. **The Extends Relationship:** An extends relationship between use cases means that the source use case extends the behavior of the destination use case. One Use Case may extend the behavior of another – typically when exceptional circumstances are encountered. The extends relationship using a stereotyped generalization relationship is depicted below.

For example, if before modifying a particular type of customer order, a user must get approval from some higher authority, then the <get approval> Use Case may optionally extend the regular <modify order> Use Case.

Representation of an implementation example of the various relationships between use cases is given below. Money transfer by the computer is an extension of the transfer operation performed at the bank lobby. In both cases, the customer must be identified.



1.1.5 VIA NET BANK ATM – USE CASE STUDY

Consider the following case study of Via Net Bank ATM system's requirements and identify the Actors and Use Cases:

- The bank client must be able to deposit an amount and withdraw an amount from his or her accounts using the touch screen at the Via Net bank ATM kiosk. Each transaction must be recorded and the client must be able to review all transactions performed against a given account. Recorded transactions must include the date, time, transaction type, amount, and account balance after the transaction.
- A Via Net bank client can have two types of accounts: a checking account and a savings account. For each checking account, one related savings account can exist.
- Access to the Via Net bank accounts is provided by a PIN number consisting of four integer digits between 0 and 9.
- One PIN number allows access to all accounts held by a bank client.
- No receipts will be provided for any account transactions.
- The bank application operates for a single banking institution only.
- Neither a checking nor a savings account can have a negative balance. The system should automatically withdraw money from a related savings account if the requested withdrawal amount on the checking account is more than its current balance. If the balance on a savings account is less than the withdrawal amount requested, the transaction will stop and the bank client will be notified.

Identifying Actors and Use Cases for the Via Net Bank ATM System

The bank application will be used by one category of users – bank clients. Identifying the actors of the system is an iterative process and can be modified. The actor of the bank system is the bank client. The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the bank application.

The following scenarios show use-case interactions between the actor (bank client) and the bank. In real life application these use cases are created by system requirements, examination of existing system documentation, interviews, questionnaires, observation, etc.

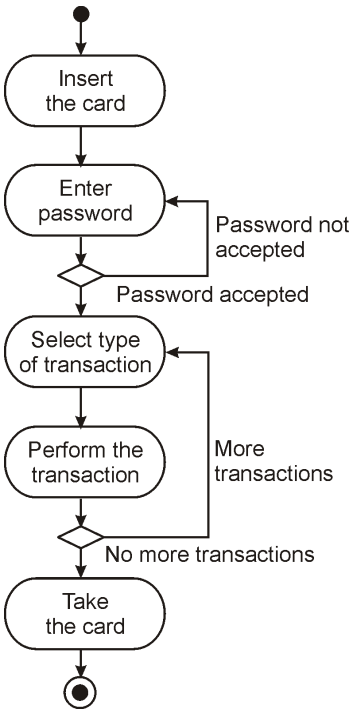
Use-case Name: Bank ATM Transaction

Bank clients interact with the bank system by going through the approval process. After the approval process, the bank client can perform the transaction. The steps involved in the ATM transaction use case are listed below:

- ATM card is inserted.
- The approval process is performed.
- Type of transaction is asked.
- Type of transaction is entered.
- Transaction is performed.
- Card is rejected.
- Requested to take card
- Card is taken.

These steps are shown in the Figure given below:

Figure 4



Use-case Name: Approval Process

The client enters a PIN that consists of four digits. If the PIN is valid, the client's accounts become available. The following are the steps:

1. Password is requested.
2. Password is entered.
3. Password is verified.

Use-case Name: Invalid PIN

If the PIN is not valid, an appropriate message is displayed to the client. It extends the approval process.

Use-case Name: Deposit Amount

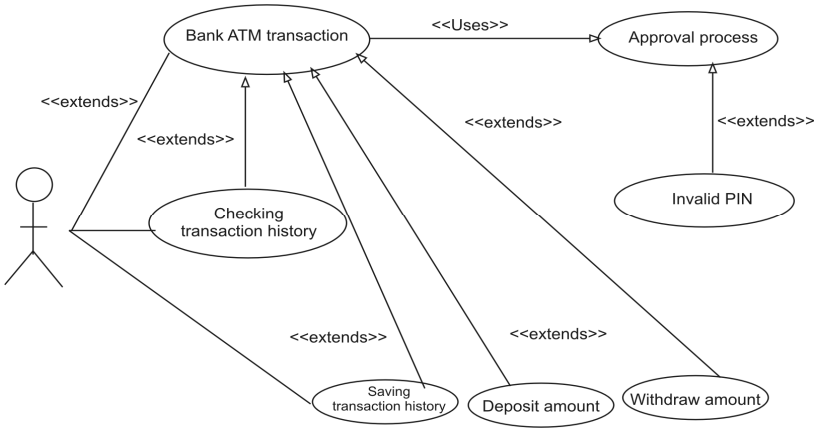
The bank client interacts with the bank system after the approval process by requesting to deposit money to an account. The client selects the account to which a deposit is going to be made and enters an amount in dollar currency. The system creates a record of the transaction. This use case extends the bank ATM transaction use case. The following are the steps:

1. Account Type is requested.
2. Deposit Amount is requested.
3. Deposit Amount is entered.
4. Either cheque or cash is kept in the envelope and deposited into the ATM.

Use-case Name: Deposit Savings

The client selects the savings account for which a deposit is going to be made. All other steps are similar to the deposit amount use case. The system creates a record of the transaction. This use case extends the deposit amount use case.

Figure 5: Transaction Use Cases



Use-case Name: Deposit Checking

The client selects the checking account for which a deposit is going to be made. All other steps are similar to the deposit amount use case. The system creates a record of the transaction. This use case extends the deposit amount use case.

Use-case Name: Withdraw Amount

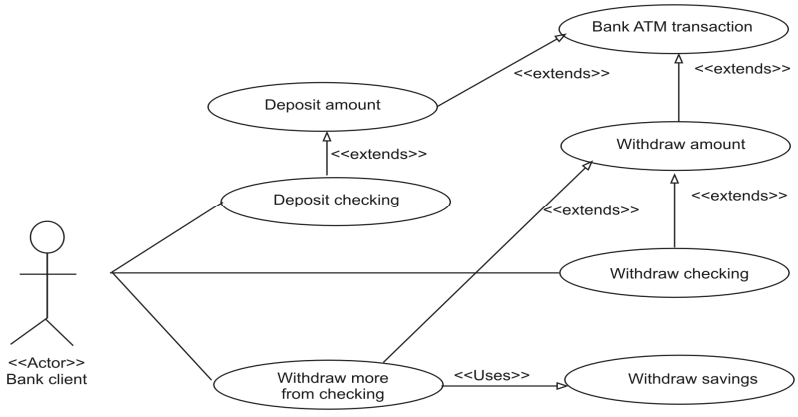
The bank client interacts with the bank system (after the approval process) by requesting to withdraw an amount from a checking account. After verifying that there are sufficient funds the transaction is performed. The system creates a record of the transaction. This use case extends the bank ATM transaction use case. The following are the steps:

1. Account Type is requested.
2. Withdrawal Amount is requested.
3. Withdrawal Amount is entered.
4. Availability of funds are verified.
5. Cash is ejected.

Use-case Name: Withdraw Checking

The client tries to withdraw an amount from his or her checking account. The amount is less than or equal to the checking account's balance and the transaction is performed. The system creates a record of the transaction. This use case extends the withdraw amount use case.

Figure 6: The Checking Account Use-cases



Use-case Name: Withdraw more from Checking

The client tries to withdraw an amount from his or her checking account. If the amount is more than the checking account's balance, the insufficient amount is withdrawn from the related savings account. The system creates a record of the transaction and the withdrawal is successful. This use case extends the withdraw checking use case and uses the withdraw savings use case.

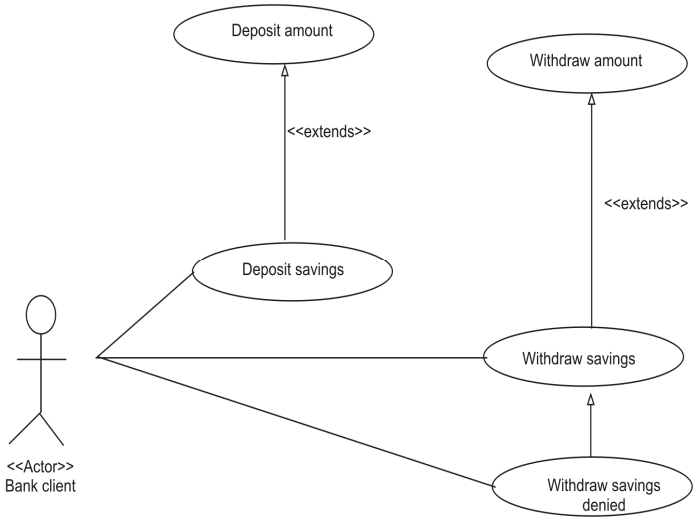
Use-case Name: Withdraw Savings

The client tries to withdraw an amount from a savings account. The amount is less than or equal to the balance and the transaction is performed on the savings account. The system creates a record of the transaction since the withdrawal is successful. This use case extends the withdraw amount use case.

Use-case Name: Checking Transaction History

The bank client requests a history of transactions for a checking account. The system displays the transaction history for the checking account. This use case extends the bank transaction use case.

Figure 7: The Savings Account Use cases Package



Use-case Name: Savings Transaction History

The bank client requests a history of transactions for a savings account. The system displays the transaction history for the savings account. This use case extends the bank transaction use case.

The use-case list contains at least one scenario of each significantly different kind of use-case instance. Each scenario shows a different sequence of interactions between actors and the system, with all decisions being definite. If the scenario consists of if statement, for each condition create one scenario.

The extends association is used when one has a use case that is similar to another use case but does a bit more. It is a subclass. In the example, the checking withdraw use case extends the withdraw amount use case. The withdraw amount use case represents the case when all goes smoothly. Many things can affect the flow of events, such as when the withdrawal is for more than the amount of money in the checking account. Withdraw more from checking is the use case that extends the checking withdraw. Association occurs when a behavior is common to more than one use case and one wants to avoid copying the description of that behavior.

Use cases are an essential tool for identifying requirements. Developing use cases is an iterative process. Although most use cases are generated at this phase of system development, one will uncover more as one proceeds. Every use case represents a potential requirement.

2. DEVELOPING EFFECTIVE DOCUMENTATION

Documenting one's project not only provides a valuable reference point and form of communication but often helps reveal issues and gaps in the analysis and design. A document can serve as a communication vehicle among the project's team members, or it can serve as an initial understanding of the requirements. In many projects, documentation can be an important factor in making a decision about committing resources. Application software is expected to provide a solution to a problem. It is very difficult, if not impossible, to document a poorly understood problem. The main issue in documentation during the analysis phase is to determine the task of the system. Decisions about how the system works are delayed to the design phase. The following points are important with regard to documentation:

- The usage of document
- Objective of the document.
- Management perspective of the document.
- Readers of the document.

2.1 Organization Conventions for Documentation

The documentation depends on the organization's rules and regulations. Most organizations have established standards or conventions for developing documentation. In many organizations, the standards are non-existent. In other cases, the standards may be excessive. Documentation should neither be too much nor too little. Each organization defines what is best for it, and one must respond to that definition and refinement.

2.2 Guidelines for Developing Effective Documentation

Bell and Evans provide the following guidelines for making documents fit the needs and expectations of the audience:

- **Common cover:** All documents should share a common cover sheet that identifies the document, the current version, and the individual responsible for the content. As the document proceeds through the life cycle phases, the responsible individual may change. That change must be reflected in the cover sheet.
- **80-20 Rule:** As for many applications, the 80-20 rule generally applies for documentations: 80 percent of the work can be done with 20 percent of the documentation. The trick is to make sure that the 20 percent is easily accessible and the rest (80 percent) is available to those (few) who need to know.
- **Familiar Vocabulary:** The formality of a document will depend on how it is used and who will read it. When developing documentation, use a vocabulary that the readers understand and are comfortable with. The main objective is to communicate with readers and not impress them with buzz words.
- **Make the Document as Short as Possible:** Assume that one is developing a manual. The key in developing an effective manual is to eliminate all repetitions; present summaries, reviews, organization chapters in less than three pages; and, make chapter headings task oriented so that the table of contents also could serve as an index.
- **Organize the Document:** Use the rules of good organization within each section. Most CASE tools provide documentation capability by providing customizable reports. The purpose of these guidelines is to assist in creating an effective documentation.

After the use cases are scheduled and ranked, the next step within iteration is to design classes, their qualities and the relationships among them. A class is a specification of the data and behaviors that instances of the class have in common. Classification is concerned with identifying classes rather than individual objects in a system.

3. APPROACHES FOR IDENTIFYING CLASSES

Following are the four approaches for identifying classes:

1. Noun phrase approach.
2. Common class patterns approach.
3. Use-case-driven approach (Sequence/Collaboration modeling).
4. Classes-Responsibilities-Collaborators (CRC) approach.
5. Class Relationships.

The use-case-driven approach is used for identifying classes and understanding the behavior of objects. The CRC approach is more useful for identifying responsibilities (e.g., methods) than classes.

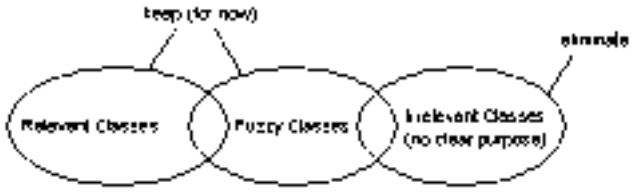
3.1 Noun Phrase Approach

Using this method, one has to read through the use cases, interviews, and requirements specification carefully, looking for noun phrases. Nouns are classes. Verbs are methods of the classes. Change all plurals to singular and make a list. Divide the list into three categories:

1. Relevant Classes.
2. Fuzzy Classes.
3. Irrelevant Classes.

These three categories are shown in figure 8. Classes are selected from the first two categories.

Figure 8



Note that some classes are implicit or taken from general knowledge. Avoid computer implementation classes; defer them to the design stage.

Selecting Classes

Selecting classes from the sets of relevant and fuzzy classes in the following problem domains:

- Redundant classes
- Adjective classes
- Attribute classes
- Irrelevant classes.

Redundant Classes: It is not advisable to keep two classes that express same information. If more than one word is being used to describe same idea, select one that is most meaningful in the system's context. This is part of building a common vocabulary for the system as a whole.

Adjective Classes: Does the object represented by the noun behave differently when the adjective is applied to it? If yes, then make new class. If no, ignore adjective. For example, if Adult Membership and youth membership behave differently, then they should be classified as different classes.

Attribute Classes: Tentative objects which are only used as values should be attributes, not classes.

Irrelevant Classes: Each class must have a purpose and every class should necessarily be clearly defined.

In software development, the process of identifying relevant classes and eliminating the irrelevant classes is an incremental process. Each iteration often uncovers some classes that have been overlooked. Classification is the essence of good object-oriented analysis and design.

3.1.1 THE VIANET BANK ATM SYSTEM: IDENTIFYING CLASSES BY USING NOUN PHRASE APPROACH

In this approach, we must start by reading the use case and applying the principles for identifying classes:

Initial list of Noun Phrases: The use cases of the bank system produce the following noun phrases:

Account

Account Balance

Amount

Approval Process

ATM Card

ATM Machine

Bank

Bank Client

Card

Cash

Check

Checking

Checking Amount

Client

Client's Account

Currency

Dollar

Envelope

Four Digits

Fund

Invalid PIN

Message

Money

Password

PIN

PIN Code

Record

Savings

Savings Account

Step

System

Transaction

Transaction History

The candidate classes must be selected from relevant and fuzzy classes. It is safe to eliminate the irrelevant classes. The relevant classes are Envelop, Digits and Step. Strikeouts indicate eliminated classes.

Account

Account Balance

Amount

Approval Process

ATM Card

ATM Machine

Bank

Bank Client

Card

Cash

Check

Checking

Checking Amount

Client

Client's Account

Currency

Dollar

~~Envelope~~

~~Four Digits~~

Fund

Invalid PIN

Message

Money

Password

PIN

PIN Code

Record

Savings

Savings Account

~~Step~~

System

Transaction

Transaction History

Reviewing the Redundant Classes and Building a Common Vocabulary: If different words are being used to describe the same idea, we must select the one that is most meaningful in the context of the system and eliminate the others i.e.

Client, Bank Client	=	Bank Client
Account, Client's Account	=	Account
PIN, PIN Code	=	PIN
Checking, Checking Account	=	Checking Account
Savings, Savings Account	=	Savings Account
Fund, Money	=	Fund
ATM Card, Card	=	ATM Card

Here is the revised list of candidate classes:

Account
Account Balance
Amount
Approval Process
ATM Card
ATM Machine
Bank
Bank Client
~~Card~~
Cash
Check
~~Checking~~
Checking Amount
~~Client~~
~~Client's Account~~
Currency
Dollar
~~Envelope~~
~~Four Digits~~
Fund
Invalid PIN
Message
~~Money~~
Password
PIN
~~PIN Code~~
Record
~~Savings~~
Savings Account
~~Step~~
System
Transaction
Transaction History

Reviewing the Classes Containing Adjectives: In this example, we have no classes containing adjectives that we can eliminate.

Reviewing the Possible Attributes: Now, we focus on identifying the noun phrases that are attributes, not classes. The noun phrases used only as values should be restated as attributes.

Amount	:	A value, not a class.
Account Balance	:	An attribute of the Account Class
Invalid PIN	:	It is only a value, not a class
Password	:	An attribute, possibly of the Bank Client class
Transaction History	:	An attribute, possibly of the Transaction Class
PIN	:	An attribute, possibly of the Bank Client Class

Here is the revised list of candidate classes:

Account
~~Account Balance~~
~~Amount~~
Approval Process
ATM Card
ATM Machine
Bank
Bank Client
~~Card~~
Cash
Check
~~Checking~~
Checking Amount
~~Client~~
~~Client's Account~~
Currency
Dollar
~~Envelope~~
~~Four Digits~~
Fund
~~Invalid PIN~~
Message
~~Money~~
~~Password~~
~~PIN~~
~~PIN Code~~
Record
~~Savings~~
Savings Account
~~Step~~
System
Transaction
~~Transaction History~~

Reviewing the class Purpose: Identifying the classes that play a role in achieving system goals and requirements is a major activity of object-oriented analysis. The classes that add no purpose to the system have been deleted from the list. The candidate classes are as follows:

ATM Machine class	:	Provides an interface to the Via Net bank.
ATM Card class	:	Provides a client with a key to an account.
BankClient class	:	A client is an individual that has a checking account and, possibly, a savings account.
Bank class	:	Bank clients belong to the Bank.
Account Class	:	An Account class is a formal class; it defines the common behaviors that can be inherited by more specific classes such as CheckingAccount and SavingsAccount.
CheckingAccount	:	It models a client's checking account and provides more specialized withdrawal service.
SavingsAccount	:	It models a client's savings account.
Transaction class	:	Keeps track of transaction, time, date, type, amount, and balance.

3.2 Common Class Patterns Approach

This approach is based on the knowledge-base of common classes that have been proposed by various researchers. They have been compiled and listed the following patterns for finding the candidate class and object:

1. **Event Classes:** These are points in time that must be recorded and remembered. Things happen at a given date and time, in an ordered sequence.

   Examples: landing, interrupt, request etc.,

2. **Organization Classes:** An organization class is a collection of people, resource, facilities, or group to which the users belong; their capabilities have a defined mission, whose existence is largely independent of the individuals.

   Example: An accounting department might be considered a potential class.

3. **People/Role Classes:** The different roles users play in interacting with the application. Two types – those people who use the system, such as operators or clerks; and those who do not use the system but about whom information is kept, such as clients, employees, teachers and managers.

4. **Place Classes:** These are physical locations that the system must keep information about.

   Examples: buildings, stores, sites or offices.

5. **Tangible Things/Device Classes:** Physical objects or group of objects, that are tangible, and devices with which the application interacts.

   Examples: cars, pressure sensors.

6. **Concept Classes:** Concepts are principles or ideas not tangible but used to organize or keep track of business activities and/or communications.

   Example: performance.

3.2.1 THE VIA NET BANK ATM SYSTEM: IDENTIFYING CLASSES BY USING COMMON CLASS PATTERNS

To better understand the common class patterns approach, we once again try to identify classes in the bank system by applying common class patterns. The common class patterns are concepts, events, organization, people, places, and tangible things and devices.

Events classes are points in time that must be recorded. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why. The bank system events classes are as follows:

- **Account Class:** An Account class is a formal (or abstract) class; it defines the common behaviors that can be inherited by more specific classes such as CheckingAccount and SavingsAccount.
- **CheckingAccount Class:** It models a client's checking account and provides more specialized withdrawal services.
- **SavingsAccount Class:** It models a client's savings account.
- **Transaction Class:** It keeps track of transaction, time, date, type, amount, and balance.
- **Organization Classes** specify collection of people, resources, facilities, or groups to which the users belong. These classes have a defined mission, whose existence does not depend upon individuals. The bank system's organization class is as follows:
  - **Bank class:** Bank clients belong to the Bank. It is a repository of accounts and processes the account's transactions.
  - **People and person classes answer this question:** What role does a person play in the system? Coad and Yourdon explain that a class being represented by a person can be divided into two types: those representing the users of the system, such as an operator or a clerk who interacts with systems, and those people who do not use the system but about whom information is kept by the system. The following is the bank system people and person class.
  - **BankClient class:** A client is an individual that has a checking account and, possibly, a savings account.
  - **Place classes** represent physical locations, buildings, stores, sites, or offices about which the system needs to keep track. Place classes are not applicable to this bank system.
  - The tangible things and devices classes represent physical objects or groups of objects that are tangible and devices with which the application interacts. In the banking system, tangible and device classes include these items.
  - **ATMMachine Class:** It allows access to all accounts held by a bank client.

3.3 Use-Case-Driven Approach

The use-case driven approach is the third approach. In this, the scenarios are described in text or through a sequence of steps. To identify objects of a system, the lowest level of executable use cases is further analyzed with a sequence diagram. By walking through the steps of the diagram, one can determine which objects are necessary for the steps to take place as described.

Implementation of Scenarios

The UML specification recommends that at least one scenario be prepared for each significantly different use-case instance. Each scenario shows a different sequence of interaction between actors and the system. It helps us to understand the behavior of the system's objects.

When you have arrived at the lowest use-case level, you may create a child sequence diagram and collaboration diagrams after which you can model the implementation of the scenario. While use cases and the steps or textual descriptions that define them offer a high-level view of the system, the sequence diagram enables you to model a more specific analysis and also assists in the design of the system by modeling the interactions between objects in the system.

To identify the objects of a system, we further analyze the lowest level use cases with a sequence and collaboration diagram pair. Sequence and collaboration diagrams represent the order in which things occur and how the objects in the system send messages to one another. These diagrams provide a macro-level analysis of the dynamics of a system.

3.3.1 VIA NET BANK ATM SYSTEM: DECOMPOSING A USE-CASE SCENARIO WITH A SEQUENCE DIAGRAM

In previous sections we identified the use case for the bank system. The following are the low level use cases:

Deposit Checking

Deposit savings

Invalid PIN

Withdraw Checking

Withdraw More from Checking

Withdraw Savings

Withdraw Savings Denied

Checking Transaction History

Savings Transaction History.

Let us create sequence/collaboration diagrams for the following use cases:

- Invalid PIN Use Case.
- Withdraw Checking Use Case
- Withdraw More from Checking Use Case.

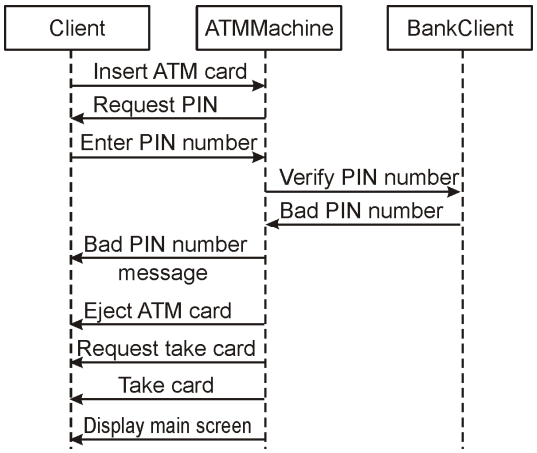
Sequence/collaboration diagrams are associated with a use case. To create a sequence you must think about the classes that probably will be involved in a use case scenario.

Consider how to prepare a sequence diagram for Invalid PIN use case. The sequence of activities that the actor Bank client performs:

- Insert ATM Card.
- Enter PIN number.
- Remove the ATM Card.

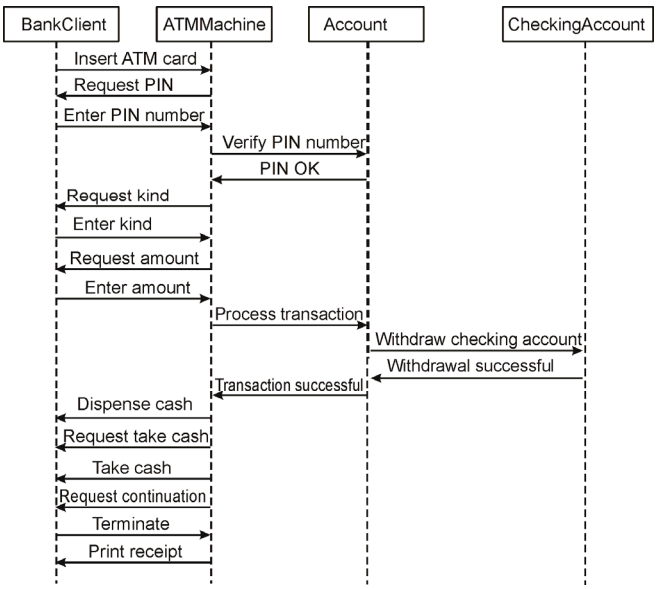
The Sequence Figure for the Invalid PIN use case is as follows:

Figure 9



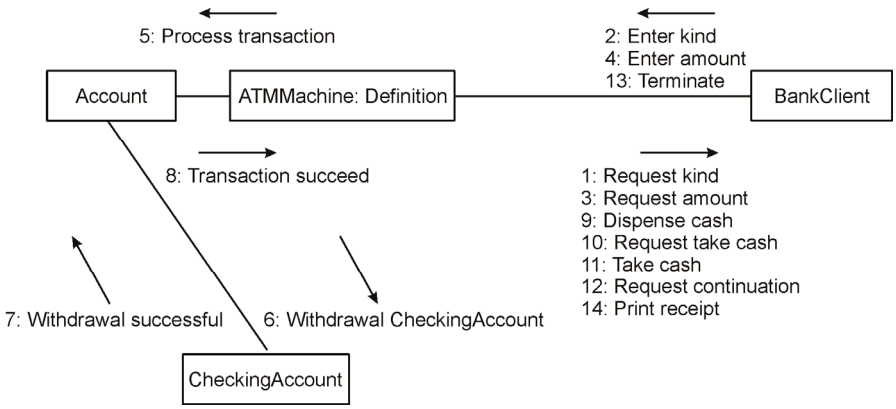
The Sequence Figure for the Withdraw Checking use case is as follows:

Figure 10



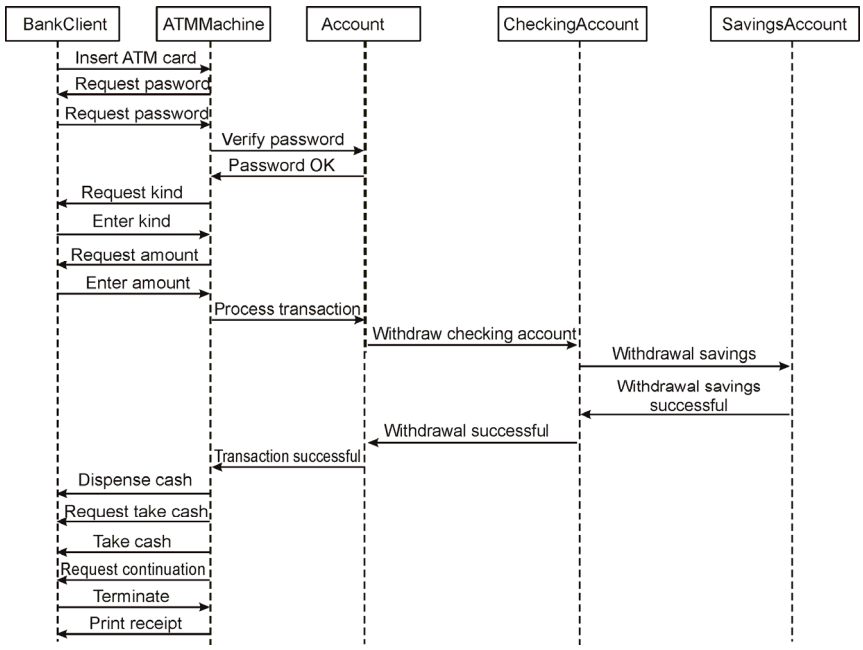
The Collaboration Figure for the Withdraw Checking use case is given in figure 11:

Figure 11



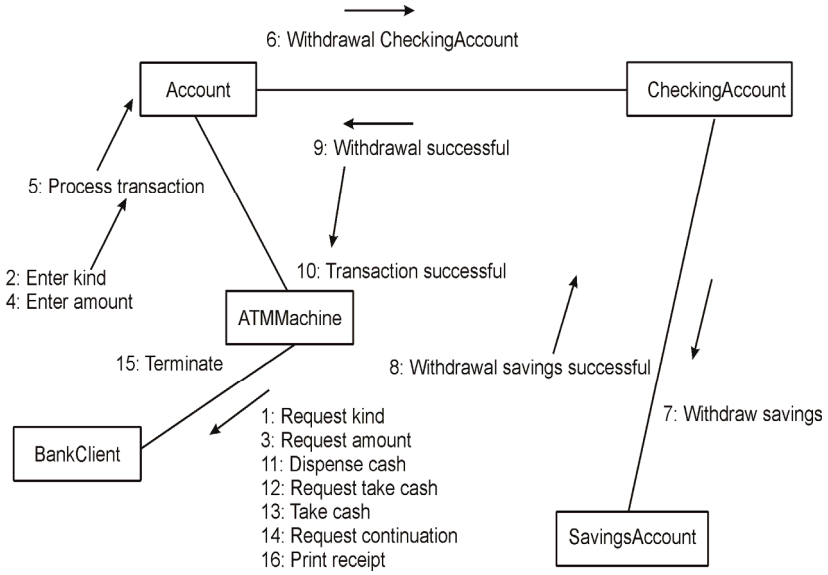
The Sequence Figure for the Withdraw more from Checking use case is as follows:

Figure 12



The Collaboration Figure for the Withdraw More from Checking use case is as follows:

Figure 13

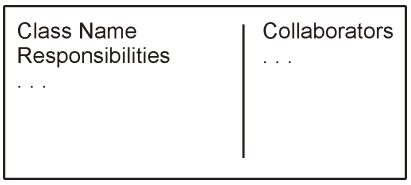


3.4 Classes, Responsibilities and Approval Collaborators

Classes, Responsibilities, and Collaborators (CRC) is a technique used for identifying responsibilities of classes and therefore their attributes and methods. Classes, responsibilities, and collaborators can help us identify classes. CRC is a teaching technique than a method for identifying classes. Classes, Responsibilities, and collaborators is based on the idea that an object either can accomplish a certain responsibility itself or it may require the assistance of other objects. By identifying an object's responsibilities and collaborators its attributes and methods can be identified.

Classes, Responsibilities, and Collaborators cards are 4" × 6" index cards. All the information for an object is written on a card, which is cheap, portable, readily available and familiar. The class name should appear in the upper left-hand corner, a bulleted list of responsibilities should appear in the left two thirds of the card, and the list of collaborators should appear in the right third. It is shown in the figure given below.

Figure 14

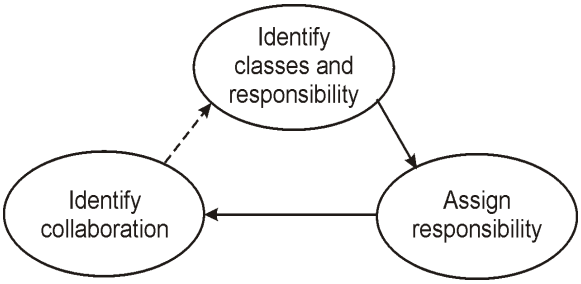


CRC Process

The Classes, Responsibilities, and Collaborators process consists of three steps:

1. Identify classes' responsibilities.
2. Assign responsibilities.
3. Identify collaborators.

Figure 15



Classes are identified and grouped by common attributes, which also provides candidates for superclasses. The class names are written onto Classes, Responsibilities, and Collaborators cards. The application's responsibilities are examined for actions and information associated with each class to find the responsibilities of each class. Next, the responsibilities are distributed. The idea in locating collaborators is to identify how classes interact.

3.4.1 THE VIA NET BANK ATM SYSTEM: IDENTIFYING CLASSES BY USING CLASSES, RESPONSIBILITIES, AND COLLABORATORS

We have already identified the initial classes of the bank system. The objective of this example is to identify objects' responsibilities such as attributes and methods in that system. Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter.

The class Account is responsible mostly to the BankClient class and it collaborates with several objects to fulfill its responsibilities. Among the responsibilities of the Account class to the BankClient class is to keep track of the BankClient balance, account number, and other data that need to be remembered. These are the attributes of the Account class. The Account class provides certain services or

methods such as means for BankClient to deposit or withdraw an amount and display the account's Balance. It is shown in the following table:

Table

Account balance number ---------- deposit withdraw getBalance	Checking Account (subclass) Savings Account (subclass) Transaction

Classes, Responsibilities, and Collaborators encourage team members to pick up the card and assume a role while "executing" a scenario. In similar fashion, other cards for the classes have been identified with the list of their responsibilities and their collaborators.

Start with few cards (classes) then implement "what if". If the situation calls for a responsibility not already covered by one of the objects, either add the responsibility to an object or create a new object to address that responsibility. If one of the objects becomes too cluttered during this process, copy the information on its card to a new card, searching for more concise ways of saying what the object does. If it is not possible to shrink the information further and the object is still too complex, create a new object to assume some of the responsibilities.

3.5 Class Relationships

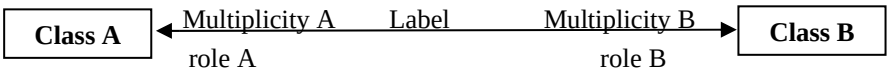
In an object-oriented environment, objects take an active role in a system. All objects stand in relationships to others on whom they rely for services and control. The relationship among objects is based on the assumptions each makes about the other objects, including what operations can be performed and what behavior results. Three types of relationships among objects are:

- **Association:** This defines association between objects that will be used in designing classes.
- **Super-sub Structure:** This defines the organization of objects into super classes and subclasses, proving a base for inheritance.
- **Aggregation and A-part-of Structure:** This defines the composition of complex classes which would provide a mechanism for managing an object within another object.

3.5.1 ASSOCIATIONS

The relationship between the concepts/objects is actually called 'Association'. Association is used to show how two classes are related to each other. Association is stronger than a 'dependency' relationship. Associations are modeled as lines connecting the two classes whose instances (objects) are involved in the relationship.

Notation for associations



Identifying Associations

A relationship should exist if a class:

– Controls
– is connected to
– is related to
– is a part of
– is a member of
– has as parts some other class in a given model.

Guidelines for Identifying Associations

The following are general guidelines for identifying the tentative associations:

- A dependency between two or more classes may be an association. Association often corresponds to a verb or prepositional phrase, such as part of, next to, works for, or contained in.
- A reference from one class to another is an association.

Common Association Patterns

The common association patterns are based on some of the common associations defined by researchers and practitioners:

- *Location Association* – next to, part of, contained in. The a-part-of relation is a special type of association.
- *Communication Association* – talk to, order to.

These associations' patterns and similar ones can be stored in the repository and added to as more patterns are discovered.

Eliminating Unnecessary Associations

- *Implementation Association*: Defer implementation-specific associations to the design phase. Implementation associations are concerned with the implementation or design of the class within certain programming or development environments and not relationships among business objects.
- *Ternary Associations*: Ternary association is an association between more than two classes.
- *Directed Actions Association*: Directed actions associations can be defined in terms of other associations. Since they are redundant, avoid these types of association.

3.5.2 SUPER-SUB CLASS RELATIONSHIPS

The super-sub class relationship represents the inheritance relationships between related classes, and the class hierarchy determines the lines of inheritance between classes. Class inheritance is useful for a number of reasons. Super-sub class relationships, also known as generalization hierarchy, allow objects to be built from other objects. The super-sub class hierarchy is a relationship between classes, where one class is the parent class of another class. The real advantage of using this technique is that we can build on what we already have and more importantly reuse what we already have. Inheritance allows classes to share and reuse behaviors and attributes.

Guidelines for Identifying Super – Sub Class Relationships

- **Top-down:** Look for noun phrase composed of various adjectives in a class name. Avoid excessive refinement. Specialize only when the subclasses have significant behavior.
- **Bottom-up**. Look for classes with similar attributes or methods. It is possible to group them by moving the common attributes and methods to an abstract class.
- **Reusability:** Move attributes and behaviors as high as possible in the hierarchy. At the same time, do not create very specialized classes at the top of the hierarchy.
- **Multiple Inheritance:** Avoid excessive use of multiple inheritance. It brings with it complications such as how to determine which behavior to get from which class, particularly when several ancestors define the same method. One way of achieving the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of another class as attribute.

3.5.3 A-PART-OF RELATIONSHIPS – AGGREGATION

A-part-of relationship, also called aggregation, represents the situation where a class consists of several component classes. A class that is composed of other classes does not behave like its parts. Aggregation is the whole-part or assembly-part relationship. It is shown by a straight line with a diamond towards the "whole". The relationship between human body and human leg is an aggregation relationship. Similarly, the relationship between a company and its departments, department and its employee are also aggregation relationships.

Properties

1. **Transitivity:** The property where, if A is part of B and B is part of C, then A is part of C.
2. **Antisymmetry:** The property of a-part-of relation, where if A is part of B, then B is not part of A.

A-part-of Relationship Patterns

- **Assembly:** An assembly is constructed from its parts and an assembly-part situation physically exits.
- **Container:** A physical whole encompasses but is not constructed from physical parts.
- **Collection-member:** A conceptual whole encompasses parts that may be physical or conceptual.

3.5.4 CASE STUDY: RELATIONSHIP ANALYSIS FOR VIA-NET-BANK ATM SYSTEM

To understand object relationship analysis, we use the familiar bank system case and apply the concepts for identifying associations, super-sub relationships, and a-part of relationships for the classes identified in previous sections.

Initially, requirement specification is understood. Object-oriented analysis and design are performed in an iterative process using class diagrams. Analysis is performed on a system, design details are added to this partial analysis model, and then the design is implemented. Changes can be made to the implementation and brought back into the analysis model to continue the cycle. This iterative process is not like the traditional waterfall technique, in which analysis is completed before design begins.

Identifying Classes' Relationships

One of the strengths of object-oriented analysis is the ability to model objects as they exist in the real world. It is also necessary to model how objects relate to each other. Several different relationships exist in the Via Net bank ATM system, which we need to define.

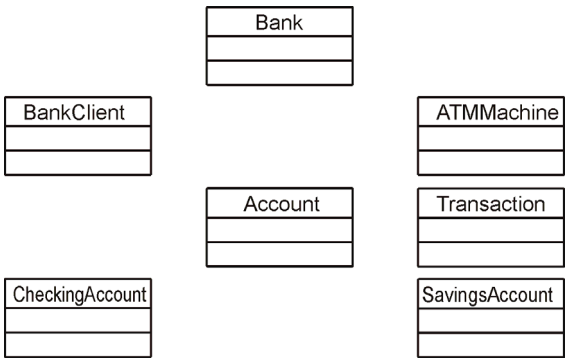
Developing a UML Class Diagram based on the Use-Case Analysis

The UML class diagram is the main static analysis and design diagram of a system. The analysis generally consists of the following class diagrams:

- One class diagram for the system, which shows the identity and definition of classes in the system, their interrelationships, and various packages containing groupings of classes.
- Multiple class diagrams that represent various pieces, or views, of the system class diagram.
- Multiple class diagrams that show the specific static relationships between various classes.

First, we need to create the classes that are shown in figure 17 and then add the relationships.

Figure 17



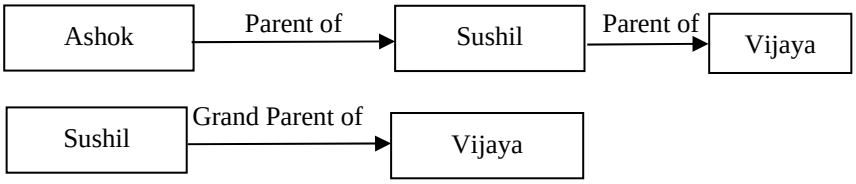
Defining Association Relationships

Identifying association begins by analyzing the interactions of each class. Remember that any dependency between two or more classes is an association. The following are general guidelines for identifying the tentative associations:

- Association often corresponds to verb or prepositional phrases, such as part of, next to, works for, or contained in.
- A reference from one class to another is an association. Some associations are implicit or taken general knowledge.

Some common patterns of associations are:

- **Location Association:** For example, next to, part of, contained in (notice that a-part-of relation is a special type of association).
- **Directed Actions Association:** They are defined in terms of other associations. For instance,



These redundant associations should be done away with.

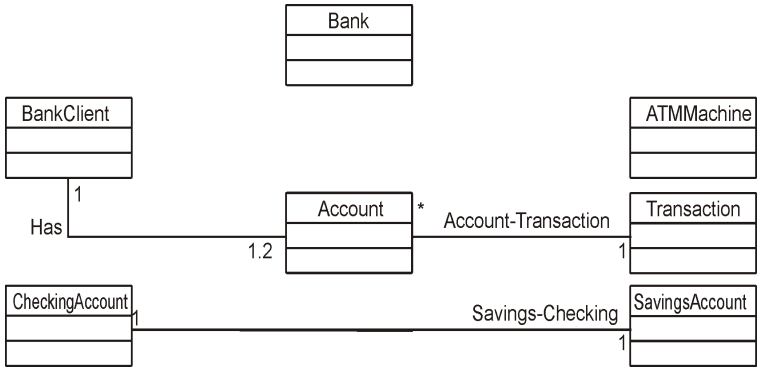
- **Communication Association:** For example, talk to, order from.

The first obvious relation is that each account belongs to a bank client since each BankClient has an account. Therefore, there is an association between the BankClient and Account classes. We need to establish cardinality among these classes. By default all associations are considered one-to-one (one client can have only one account and vice versa). However, since each BankClient can have one or two accounts, we need to change the cardinality of the association. Other associations and their cardinalities are defined in the following table and demonstrated in figure 18.

Some Associations and their Cardinalities in the Bank System

Class	Related class	Association name	Cardinality
Account	BankClient	Has	One
BankClient	Account		One or two
SavingsAccount	CheckingAccount	Savings-Checking	One
CheckingAccount	SavingsAccount		Zero or one
Account	Transaction	Account-Transaction	Zero or more
Transaction	Account		One

Figure 18



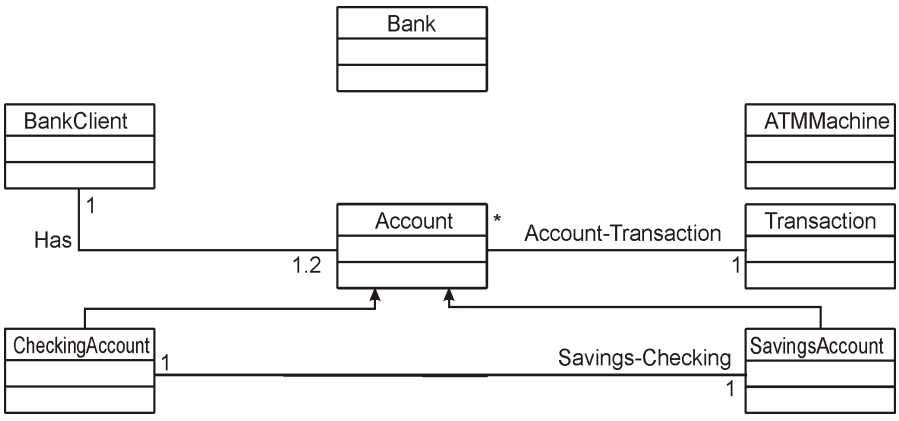
Defining Super-Sub Relationships

Let us review the guidelines for identifying super-sub relationships:

- **Top-down:** Look for noun phrases composed of various adjectives in the class name.
- **Bottom-up:** Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class.
- **Reusability:** Move attributes and behaviors (methods) as high as possible in the hierarchy.
- **Multiple Inheritance:** Avoid excessive use of multiple inheritance.

Both CheckingAccount and SavingsAccount are types of accounts. They can be defined as specializations of the Account class. When implemented, the Account class will define attributes and services common to all kinds of accounts, with CheckingAccount and SavingsAccount each defining methods that make them more specialized. Figure 19 depicts the super-sub relationships among Accounts, SavingsAccount, and CheckingAccount.

Figure 19



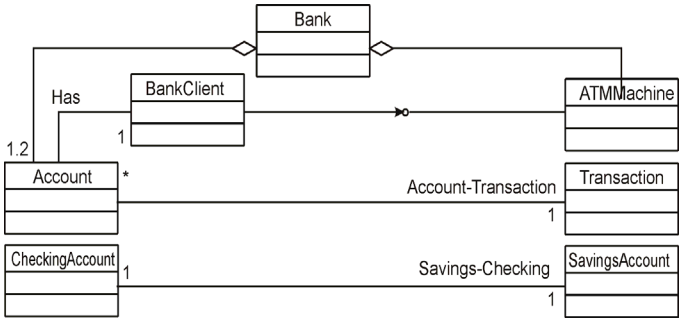
Identifying the Aggregation/a-Pat-of Relationship

To identify a-part of structures, we look for the following clues:

- **Assembly:** A physical whole is constructed from physical parts.
- **Container:** A physical whole encompasses but is not constructed from physical parts.
- **Collection-Member:** A conceptual whole encompasses parts that may be physical or conceptual.

A bank consists of ATM machines, accounts, buildings, employees, and so forth. However, since buildings and employees are outside the domain of this application, we define the Bank class as an aggregation of ATMMachine and Account classes. Aggregation is a special type of association. Figure 20 depicts the association, generalization, and aggregation among the bank systems classes.

Figure 20



4. IDENTIFYING ATTRIBUTES AND METHODS

Identifying attributes and methods is like finding classes and an iterative process. Responsibilities identify problems to be solved. A responsibility serves as a handle for discussing potential solutions. The responsibilities of an object are expressed by a handful of short verb phrases, each containing an active verb. Attributes are things an object must remember such as color, cost, and manufacturer. Identifying attributes of a system's classes starts with understanding the system's responsibilities.

The following questions help in identifying the responsibilities of classes and deciding about data elements that need to be kept track of:

- What information about an object should we keep track of?
- What services a class must provide?

Answering the first question will help us identify attributes of a class. Answering the second question will help us identify the methods of a class.

5. DEFINING ATTRIBUTES BY ANALYZING USE CASES AND OTHER UML DIAGRAMS

By analyzing the use cases and sequence/collaboration, activity, and state diagrams, one can know the responsibilities of a class and also the way they interact in order to perform a task. The goal is to understand what the class would have to know. Responsible for knowing. An object in an object-oriented environment; should have the following characteristics:

- How it is going to be used?
- How it is going to collaborate with other classes?
- How it is described in the context of this system's responsibility?
- What it should know?
- What state information should it remember over time?
- What states can it be in?

Guidelines for Defining Attributes

- Attributes usually correspond to nouns followed by prepositional phrases. Attributes may also correspond to adjectives or adverbs.
- Keep the class simple; state enough attributes to define the object state.
- Attributes are less likely to be fully described in the problem statement.
- Omit derived attributes.
- Do not carry discovery of attributes to excess.

5.1 Defining Attributes for Via Net Bank Objects

By analyzing the use cases, the sequence/collaboration diagrams, and the activity diagram it is apparent that, for the BankClient class, the problem domain and the system dictate certain attributes. By looking at the activity diagram, we notice that the BankClient must have a PIN number (or password) and Card number. Therefore, the PIN number and CardNumber are appropriate attributes for the BankClient.

The attributes of the BankClient are:

First name

Last name

PinNumber

CardNumber

Account: Account

Defining Attributes for the Account Class: We have defined the following attributes for the Account class:

Number

Balance

Defining Attributes for the Transaction Class: The Transaction class, for the most part, must keep track of the time and amount of a transaction. Here are some attributes for the Transaction class:

TransID

TransDate

TransTime

TransType

Amount

PostBalance

Defining Attributes for the ATMMachine Class: The ATMMachine class could have the following attributes:

Address

State

SUMMARY

- The main objective of analysis is to capture a complete, unambiguous, and consistent picture of the requirements of the system. The models of the system concentrate on describing what the system does rather than how the system does. Separating the behavior of a system from the way it is implemented requires viewing the system from the perspective of the users rather than that of the machine.
- Analysis is a creative activity that involves understanding the problem, its associated constraints, and methods of overcoming those constraints. It is an iterative process that goes on until the problem is well understood. The main objective of object-oriented analysis is to find out what the problem is by developing a use-case model. Jacobson et al. call this "what model".
- Use cases are an essential tool in capturing requirements. Capturing use cases is one of the first things to do in coming up with requirements. Every use case is a potential requirement. A use-case model can be developed by talking to the users and discussing the various things they might want to do. Each use case must have a name and short textual description, no more than a few paragraphs.
- Requirements must be traceable across analysis, design, coding, and testing. The unified approach follows Jacobson et al., life cycle to produce systems that can be traced across all phases of the development.
- Finding classes is one of the hardest activities in OOA. We follow four approaches for identifying the classes. The process of identifying classes can improve gradually through the incremental process.
- The first method for identifying classes is the noun phrase. Second method is the common class patterns approach based on the knowledge base of the common classes proposed by various researchers. Third method is use-case driven, in this to identify the objects of a system and their behaviors; the lowest level of executable use case is analyzed with a sequence and collaboration diagram pair. Last method is Classes, Responsibilities and Collaborators, which is a useful tool for learning about class responsibilities and identifying the classes.
- There are three types of relationships among classes i.e., association, generalization hierarchy, and aggregation. To identify associations, begin by analyzing the interactions of each class and responsibilities for dependencies. To identify super-sub relationships, look for noun phrases composed of various adjectives in the class in top-down analysis. Aggregation represents a situation where a class comprises several component classes.
- Identifying attributes and methods is like finding classes, a difficult activity and an iterative process.

Chapter VIII

Object-Oriented Design

After reading this chapter, you will be conversant with:

- Object-Oriented Design Process
- Designing Classes: Refining Attributes
- Designing Methods and Protocols
- Object Storage and Persistence
- User Interface Design

During the system design phase, strategies that are chosen are implemented in Object Oriented (OO) design. This phase of object design is the next one to follow after the analysis and system design; or in other words it is an extension of analysis phase. The OO design phase adds implementation details such as restructuring classes for efficiency, internal data structures and algorithms to implement operations, implementation of control, implementation of associations and packaging into physical modules. OO design phase is iterative where in a number of classes are added and relationships between objects are defined as one moves from one level to another of the design phase. Objects that are identified in the analysis phase are implemented in such a way that memory utilized, execution time and related costs are minimized by enforcing proper controls.

Object-oriented design requires taking the objects identified during object-oriented analysis and designing classes to represent them. As a class designer, one should know the specifics of the class that is being designed. How that class interacts with other classes should be taken care of. Once classes and their interactions have been identified, one is ready to design classes. Underlying the functionality of any application is the quality of its design.

1. OBJECT-ORIENTED DESIGN PROCESS

The object-oriented design process consists of the following activities:

1. Apply design axioms to design classes, their attributes, methods, associations, structures, and protocols.
    - 1.1 Refine and complete the static UML class diagram by adding details to the UML class diagram. This step consists of the following activities:
        - 1.1.1 Refine attributes.
        - 1.1.2 Design methods and protocols by utilizing a UML activity diagram to represent the method's algorithm.
        - 1.1.3 Refine associations between classes (if required).
        - 1.1.4 Refine class hierarchy and design with inheritance (if required).
    - 1.2 Iterate and refine again.
2. Design the access layer
    - 2.1 Create mirror classes. For every business class identified and created, create one access class. For example, if there are three business classes (Class1, Class2, and Class 3), create three access layer classes (ClassIDB, Class2DB, and Class3DB).
    - 2.2 Identify access layer class relationships.
    - 2.3 Simplify classes and their relationships. The main goal here is to eliminate redundant classes and structures.
        - 2.3.1 Redundant classes: Do not keep two classes that perform similar request and results activities. Simply select one and eliminate the other.
        - 2.3.2 Method classes: Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
    - 2.4 Iterate and refine again.

3. Design the view layer classes.
   - 3.1 Design the macro level user interface, identifying view layer objects.
   - 3.2 Design the micro level user interface, which includes these activities:
     - 3.2.1 Design the view layer objects by applying the design axioms and corollaries.
     - 3.2.2 Build a prototype of the view layer interface.
   - 3.3 Test usability and user satisfaction.
   - 3.4 Iterate and refine.
4. Iterate and refine the whole design. Reapply the design axioms and, if needed, repeat the preceding steps.

Utilizing an incremental approach, all stages of software development (analysis, modeling, designing, and implementation or programming) can be performed incrementally. Therefore, all the right decisions need not be made up front.

From the UML class diagram, one can begin to extrapolate the classes that have to be built and also the existing classes that can be reused. If there are a number of classes that are in some way related to one another, it would be possible to make them common sub-classes of an existing class. Superclasses can also be generated by considering the common characteristics. Thus, we can say that a good object-oriented design is very iterative. Design also must be reflected across requirements, analysis, design, coding, and testing. The design must progress in coherent steps from the requirements model.

1.1 Object-Oriented Design Axioms

By definition, an axiom is a fundamental truth that is always observed to be valid and for which there are no counter exceptions.

Two of the most important Object Oriented Design Axioms are:

- The Independence axiom.
- The Information axiom.

The **Independence axiom** deals with the relationships between components. One component should be able to satisfy the requirements without affecting or influencing another component.

The **Information axiom** emphasizes the importance of simplicity. Occam's razor rule states "The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straightforwardness".

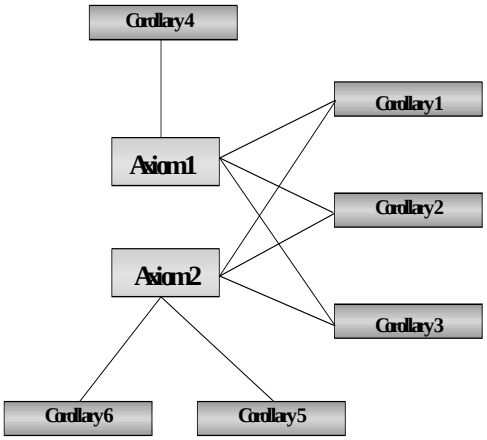
1.2 Corollaries

A collary is a proposition that follows from an axiom or another proposition that has been proved.

From the two design axioms, many corollaries may be derived as a direct consequence of the axioms as shown in figure 1. These corollaries may be more useful in making specific design decisions, since they can be applied to actual situations more easily than the original axioms. They even may be called design rules, and all are derived from the two basic axioms. Some of the more important design corollaries are:

- Uncoupled design with less information content
- Single purpose
- Large number of simple classes
- Strong mapping
- Standardization
- Design with inheritance.

Figure 1: Relationship between Axioms and Corollaries



1.2.1 COROLLARY 1 – UNCOUPLED DESIGN WITH LESS INFORMATION CONTENT

Highly cohesive objects can improve coupling because only a minimal amount of essential information need to be passed between objects.

Coupling

Coupling is a measure of the strength of association established by a connection from one object or software component to another. Coupling is a binary relationship: A is coupled with B. Coupling is important when evaluating a design because it helps to focus on an important issue in design. For example, a change to one component of a system should have a minimal impact on other components. Strong coupling among objects complicates a system, since the class is harder to understand or highly interrelated with other classes. The degree of coupling is a function of:

- The complexity of the connection.
- Whether the connection refers to the object itself or something inside it.
- What is being sent or received?

The degree or strength of coupling between two components is measured by the amount and complexity of information transmitted between them. Strong coupling complicates a system and increases the complexity or obscurity of the interface. Coupling decreases when the connection is to the component interface rather than to an internal component (Encapsulation). Coupling for data connections is lower than for control connections. Object-oriented design has two types of coupling: interaction coupling and inheritance coupling.

Interaction coupling involves the amount and complexity of messages between components. It is desirable to have little interaction. Coupling also applies to the complexity of the message. The general guideline is to keep the messages as simple as possible. In general, if a message involves more than three parameters (e.g., in Method (X, Y, Z), the X, Y, and Z are parameters), it should be examined to see if it can be simplified. It has been observed that objects connected to complex messages are tightly coupled, i.e., any change to one invariably leads to a ripple effect of changes in others. In addition to minimizing the complexity of message connections, the number of messages sent and received by an object containing different types of interaction couplings need to be reduced.

Inheritance is a form of coupling between super- and sub-classes. A subclass is coupled to its superclass in terms of attributes and methods. Unlike interaction coupling, high inheritance coupling is desirable. To achieve high inheritance coupling in a system, each specialization class should not inherit lots of unrelated and unneeded methods and attributes. For example, if the subclass is overwriting most of the methods or not using them, this is an indication that inheritance coupling is low and the designer should look for an alternative generalization-specialization structure.

Cohesion

Coupling deals with interactions between objects or software components. One also needs to consider interactions within a single object or software component, called ***cohesion***. Cohesion reflects the "single-purposeness" of an object. Highly cohesive components can lower coupling because only a minimum of essential information needs to be passed between components. Cohesion also helps in designing classes that have very specific goals and clearly defined purposes.

Method cohesion, like function cohesion, means that a method should carry only one function. A method that carries multiple functions is undesirable. Class cohesion means that all the class methods are the methods of derived classes. Inheritance cohesion is concerned with the following questions:

- How interrelated are the classes?
- Does specialization really portray specialization or is it just something arbitrary?

1.2.2 COROLLARY 2: SINGLE PURPOSE

Each class must have a purpose and it should be clearly defined which is necessary in the context of achieving the system's goals. When one documents a class, one should be able to easily explain its purpose in short. If one cannot, then the class has to be subdivided into more independent pieces. i.e., it should be kept simple; and precise. Each method must provide only one service and be of moderate size, no more than a page.

1.2.3 COROLLARY 3: LARGE NUMBER OF SIMPLER CLASSES, REUSABILITY

The less specialized the classes are, the more likely that future problems can be solved by recombining existing classes or by adding a minimal number of sub classes. Smaller classes can be used in a better way by reusing them in other projects. Large and complex classes are difficult to reuse because of their high degree of specialilzation. Since object orientation gives special consideration to encapsulation, modularization and polymorphism, the underlying objective is reused rather than building a new class.

1.2.4 COROLLARY 4: STRONG MAPPING

Object-oriented analysis and object-oriented design are based on the same model. As the model progresses from analysis to implementation, more details are added, but the structure remains the same. For example, during analysis one might identify a class Employee. During the design phase, one needs to design this class, i.e., design its methods, its association with other objects, and its view and access classes. A strong mapping links classes identified during analysis and classes are designed during the design phase.

The analyst identifies objects' types and inheritance, and determines the events that change the state of objects. The designer adds details to this model in the form of designer screens, user interaction, and client-server interaction. The thought process flows so naturally from analysis to design that there is a blurred boundary separating the analysis and design phases.

1.2.5 COROLLARY 5: STANDARDIZATION

To reuse classes, it is necessary to have a good understanding of the development environment. Most languages come with several built-in class libraries. The knowledge of existing classes will help in determining what new classes are needed and where one might inherit useful behaviors rather than to reinvent the wheel. Documentation regarding class libraries should be up-to-date and easily navigable. Design patterns might provide a way of capturing, documenting and storing design knowledge.

1.2.6 COROLLARY 6: DESIGNING WITH INHERITANCE

When one implements a class, one has to determine its ancestor, what attributes it will have, and what messages it will understand. Then, one has to construct its methods and protocols. One will choose inheritance to minimize the amount of program instructions. Satisfying these constraints sometimes means that a class inherits from a superclass that may not be obvious at first glance. For example, say, one is developing an application for a government department that manages the licensing procedure for a variety of regulated entities.

The design is approved, implementation is accomplished, and the system goes into production. Till the time the real-world problems do not cross over the boundary into the system, the design is elegant.

Multiple inheritances bring with them some complications, such as how to determine which behavior to get from which class, particularly when several ancestors define the same method. It is also difficult to understand programs written in a multiple inheritance system. One way of achieving the benefits of multiple inheritances in a language with single inheritance is to inherit from the most appropriate class and add an object of another class as an attribute or aggregation.

1.3 Object-Oriented Design Philosophy

Object-oriented development requires one to think in terms of classes. A great benefit of the object-oriented approach is that classes organize related properties into units that stand on their own. We go through a similar process as we learn about the world around us. As new facts are acquired, we relate them to existing structures in our environment. The activity in designing an application is coming up with a set of classes that work together to provide the desired functionality.

The first step in building an application should be to design a set of classes, each of which has a specific expertise and all of which can work together in useful ways. Applying design axioms and carefully designed classes can have a synergistic effect, not only on the current system but on its future evolution.

1.4 Designing Classes

Designing classes consists of the followings activities:

1. Apply design axioms to design classes, their attributes, methods, associations, structures and protocols.

    1.1 Refine and complete the static UML class diagram by adding details to the UML class diagram. This step consists of the following activities:

        1.1.1 Refine attributes.

        1.1.2 Design methods and protocols by utilizing a UML activity diagram to represent the method’s algorithm.

        1.1.3 Refine associations between classes (if required).

        1.1.4 Refine class hierarchy and design with inheritance (if required).

    1.2 Iterate and refine again.

Object-oriented design is an iterative process with the design improving at iteration.

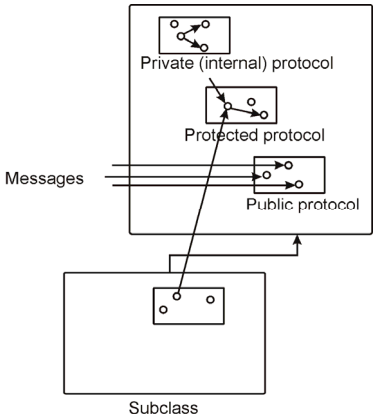
1.5 Class visibility: Designing well-defined Public, Private and Protected Protocols

In designing methods or attributes for classes, there are two problems:

1. Protocol or interface to the class operations and its visibility.
2. The way of implementing methods.

The protocols, or the messages that a class understands can be hidden from other objects (private protocol) or made available to other objects (public protocol). Public protocols define the implementation of an object. Implementation, by definition, is hidden and beyond the limit of other objects. It is shown in figure 2 below.

Figure 2



It is important in object-oriented design to define the public protocol between the associated classes in the application. This is the set of messages that a class of a certain generic type must understand, although the interpretation and implementation of each message is up to the individual class.

Classes have a set of methods that it uses only internally, to pass messages to itself. The **private protocol (visibility)** of the class includes messages that normally should not be sent from other objects; it is accessible only to the operations of that class. In private protocol, only the class itself can use the method.

The **public protocol (visibility)** defines the stated behavior of the class as a citizen in a population and is important for users as well as future descendants. It is accessible to all classes.

The **protected protocol (visibility)** defines that the subclasses can use the method in addition to the class itself.

The problem of **encapsulation leakage** occurs when details about a class's internal implementation are disclosed through the interface. As more internal details become visible, the flexibility to make changes in the future decreases. If an implementation is completely open, almost no flexibility is retained for future changes. This situation hampers not only flexibility but also the quality of the design.

Design the interface between a superclass and its subclass just as the class's interface to clients; this is the contract between the super and sub classes. If the interface is not designed properly, it can lead to violating the encapsulation of the superclass. The protected portion of the class interface can be accessed only by subclasses.

1.5.1 PRIVATE AND PROTECTED PROTOCOL LAYERS: INTERNAL

Items in these layers define the implementation of the object. Apply the design axioms and corollaries, especially corollary 1 to decide what should be private – what attributes? What methods? It is to be noted that highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.

1.5.2 PUBLIC PROTOCOL LAYERS: EXTERNAL

Items in these layers define the functionality of the object. When designing class protocols, it is to be noted that:

- Good design allows for polymorphism.
- Not all protocol should be public: again apply design axioms and corollaries.

The following key questions must be answered:

- What are the class interfaces and protocols?
- What public protocol will be used or what external messages must the system understand?
- What private or protected protocol will be used or what internal messages or messages from a subclass must the system understand?

2. DESIGNING CLASSES: REFINING ATTRIBUTES

Attributes identified in object-oriented analysis must be refined with an eye on implementation during this phase. In the analysis phase, the name of the attribute is sufficient. In the design phase, the detailed information must be added to the model. The main goal of this activity is to refine existing attributes or add attributes that can elevate the system into implementation.

2.1 Attribute Types

Attributes represent the state of an object. When the state of the object changes, these changes are reflected in the value of attributes. The three basic types of attributes are:

1. Single-value attributes – It has only one value or state. Attributes such as name, address, or salary are of this type.
2. Multiplicity or multivalued attributes – As the name implies, it can have a collection of many values at any point in time.
3. Reference or another object, or instance attributes – Instance connection attributes are required to provide the mapping needed by an object to fulfill its responsibilities.

2.2 UML Attribute Presentation

The syntax of the UML presentation will be as follows:

visibility name : type-expression = initial-value

Where visibility is

\+ – **Public** visibility

\# – **Protected** visibility

\– – **Private** visibility

Type-expression is a language-dependent specification of the implementation type of an attribute.

Initial-value is a language-dependent expression for the initial value of a newly created object.

Multiplicity may be indicated by placing a multiplicity indicator in brackets after attribute name i.e., names [20]: string.

In the absence of a multiplicity indicator, an attribute holds exactly one value.

2.3 Refining Attributes for the ViaNet Bank objects

We go through the ViaNet Bank ATM system classes and refine the attributes identified during object-oriented analysis.

2.3.1 REFINING ATTRIBUTES FOR THE BANKCLIENT CLASS

During object-oriented analysis, we identified the following attributes:

- FirstName
- LastName
- PinNumber
- CardNumber

At this stage, we need to add more information to these attributes, such as visibility and implementation type. Furthermore, additional attributes can be identified during this phase to enable implementation of the class:

#firstName: String

#lastName: String

#pinNumber: String

#cardNumber: String

#account: Account (instance connection)

To design an association between the BankClient and the Account classes, we need to add an account attribute of type Account, since the BankClient needs to know about his or her account and this attribute can provide such information for the BankClient class. This is an example of instance connection, where it represents the association between the BankClient and the Account objects. All the attributes have been given protected visibility.

2.3.2 REFINING ATTRIBUTES FOR THE ACCOUNT CLASS

Below given is the refined list of attributes for the Account class:

#number: String

#balance; float

#transaction: Transaction (This attribute is needed for implementing the association between the Account and Transaction classes).

#bankClient: BankClient (This attribute is needed for implementing the association between the Account and BankClient classes).

At this point we must make the Account class very general, so that it can be reused by the checking and savings accounts.

2.3.3 REFINING ATTRIBUTES FOR THE TRANSACTION CLASS

The attributes for the Transaction class are these:

#transID: String

#transDate: Date

#transTime: Time

#transType: String

#amount: float

#postBalance: float

2.3.4 REFINING ATTRIBUTES FOR THE ATMMACHINE CLASS

The ATMMachine class could have the following attributes:

#address: String

#state: String

2.3.5 REFINING ATTRIBUTES FOR THE CHECKINGACCOUNT CLASS

Add the savings attribute to the class. The purpose of this attribute is to implement the association between the CheckingAccount and SavingsAccount classes.

2.3.6 REFINING ATTRIBUTES FOR THE SAVINGSACCOUNT CLASS

Add the checking attribute to the class. The purpose of this attribute is to implement the association between the SavingsAccount and CheckingAccount classes.

Figure 3: Complete UML Class Diagram

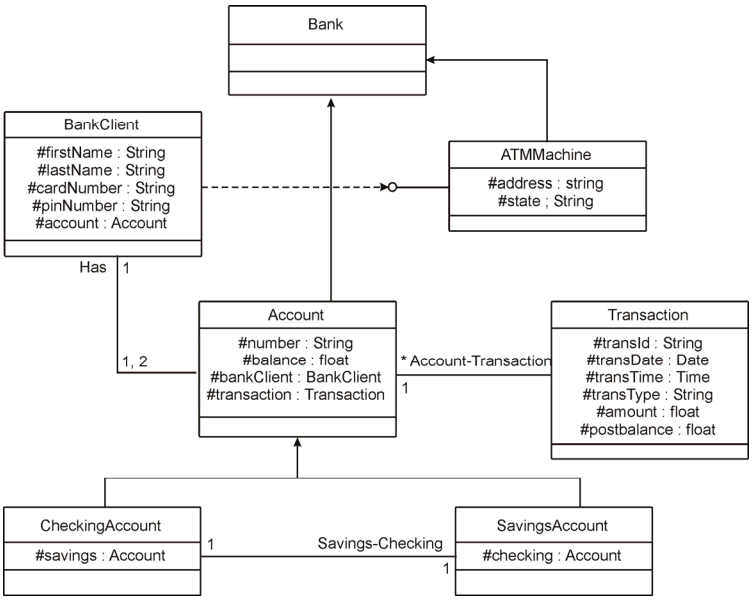


Figure 3 shows a more complete UML class diagram for the bank system. At this stage, we also need to add a very short description of each attribute or certain attribute constraints. For example,

Class ATMMachine

#address: String (The address for this ATM machine).

#state: String (The state of operation for this ATM machine, such as running, off, idle, out of money, security alarm).

3. DESIGNING METHODS AND PROTOCOLS

The goal of this activity is to specify the algorithm for methods identified. Once methods are designed in some formal structure such as UML activity diagrams with an OCL description, they can be converted to programming language manually or in automated fashion. A class can provide several types of methods:

- *Constructor*: The method that creates instances of the class.
- *Destructor*: The method that destroys instances.
- *Conversion method*: The method that converts values from one unit of measure to another.
- *Copy method*: The method that copies the contents of one instance to another instance.
- *Attribute set*: The method that sets the values of one or more attributes.
- *Attribute get*: The method that returns the values of one or more attributes.
- *I/O method*: The method that provides or receives data to or from a device.
- *Domain specific*: The method specific to the application.

The goal should be to maximize cohesiveness among objects and software components in order to improve coupling, because only a minimal amount of essential information should be passed between components.

3.1 Design Issues: Avoiding Design Pitfalls

It is important to apply design axioms to avoid common design problems and pitfalls. It is possible to gather common pieces of expertise from several classes, which in itself becomes another "peer" class that others consult; or it is possible to create a superclass for several classes for whom similar code at a single place. The aim of the designer is to make reuse existing classes in order to avoid creating new classes as much as possible.

Another problem with class definitions is that of lost object focus. A meaningful class definition starts out simple and clean but, as time goes on and changes are made, becomes larger and larger, with the class identity becoming harder to state concisely. The documentation part should be able to describe the purpose of a class in a few sentences.

Following points need to be noted:

- The design of class should be done carefully and at the same time the role of an object should be well-defined. If the object loses focus, the design needs to be modified. Apply Corollary 2.
- Some functions need to be moved into new classes that the object would use. Apply Corollary 1.
- Break up the class into two or more classes. Apply Corollary 3.
- The class definition can be modified based on experience gained.

3.2 UML Operation Presentation

The following operational presentation has been suggested by UML. The operational syntax is this:

visibility name: (parameter-list) : return-type-expression

where **visibility** is one of:

\+ **public visibility**

\# **protected visibility**

\- **private visibility**

name is the name of the operation.

parameter-list is the list of parameters, separated by commas, each specified by

name: type-expression = default value

return-type-expression is a language dependent specification of the implementation of the value returned by the method.

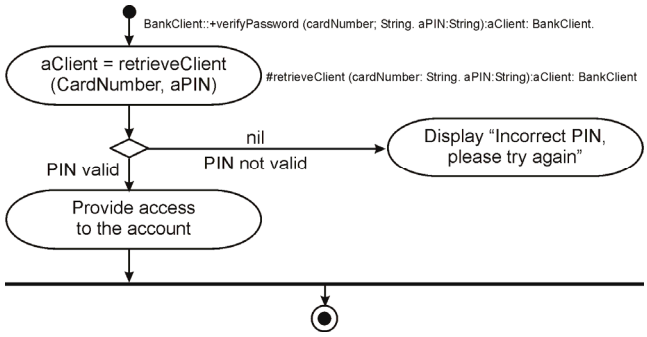
3.3 Designing Methods for the via Net Bank Objects

At this point, the design of the bank business model is conceptually complete. The objects that make up the business layer as well as what services they provide are identified. All that remains is to design methods, the user interface, database access, and implement the methods using any object-oriented programming language. We represent the methods' algorithms with UML activity diagrams, which very easily can be translated into any language. In essence, this phase prepares the system for the implementation. The actual coding and implementation should be relatively easy and, for the most part, can be automated by using CASE tools. It is always difficult to code when we have no clear understanding of what we want to do.

3.3.1 BANKCLIENT CLASS VERIFYPASSWORD METHOD

A client PIN code is sent from the ATMMachine object and used as an argument in the verifyPassword method. The verifyPassword method retrieves the client record and checks the entered PIN number against the client's PIN number. If they match, it allows the user to proceed. Otherwise, a message sent to the ATMMachine displays "Incorrect PIN, please try again".

Figure 4

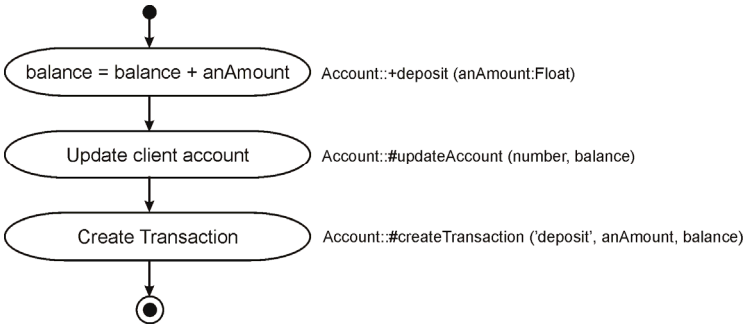


The verifyPassword method first creates a bank client object and then attempts to retrieve the client data based on the supplied card and PIN numbers.

3.3.2 ACCOUNT CLASS DEPOSIT METHOD

An amount to be deposited is sent to an account object and used as an argument to the deposit service. The account adjusts its balance to its current balance plus the deposit amount. The amount object records the deposit by creating a transaction object containing the date and time, posted balance, and transaction type and amount.

Figure 5

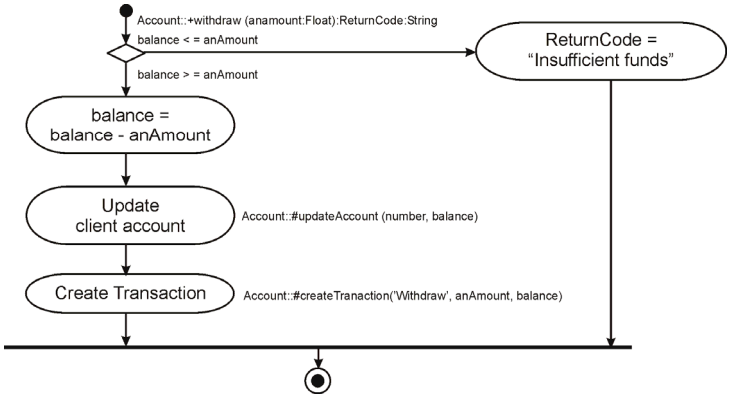


3.3.3 ACCOUNT CLASS WITHDRAW METHOD

It is designed to be inherited by the CheckingAccount and SavingsAccount classes to implement automatic funds transfer. The following describes the withdraw method.

An amount to be withdrawn is sent to an account object and used as the argument to the withdraw service. The account checks its balance for sufficient funds. If enough funds are available, the account makes the withdrawal and updates its balance; otherwise, it returns an error, saying "insufficient funds". If successful, the account records the withdrawal by creating a transaction object containing date and time, posted balance, and transaction type and amount.

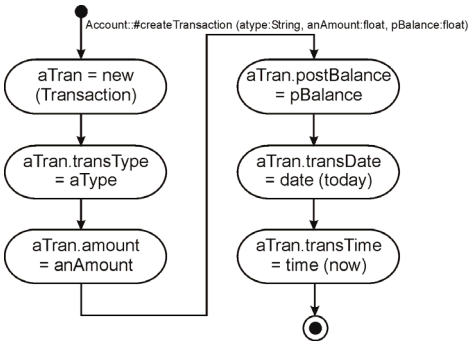
Figure 6



3.3.4 ACCOUNT CLASS CREATETRANSACTION MEHTOD

The createTransaction method generates a record of all transactions performed against it. The description is as follows: Each time a successful transaction is performed against an account, the account object creates a transaction object to record it. Arguments into this service include transaction type (withdrawal or deposit), the transaction amount, and the balance after the transaction. The account creates a new transaction object and sets its attributes to the desired information. Add this description to the createTransaction's description field.

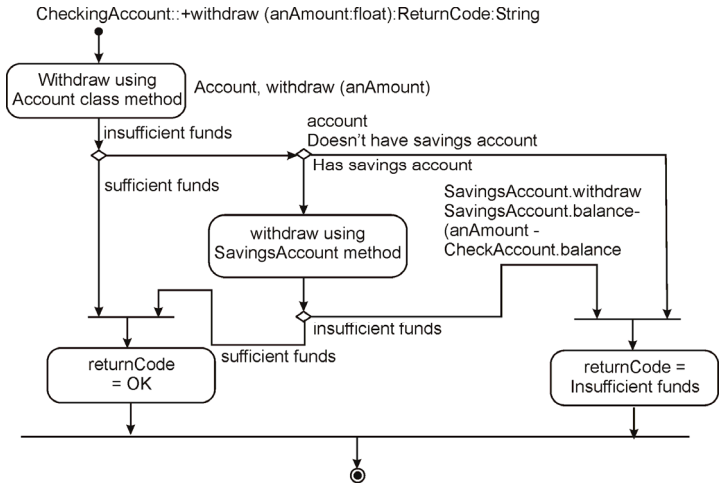
Figure 7



3.3.5 CHECKING ACCOUNT CLASS WITHDRAW METHOD

It takes into consideration the possibility of withdrawing excess funds from a companion savings account. The process will be as follows (figure 8): An amount to be withdrawn is sent to a checking account and used as the argument to the withdrawal service. If the account has insufficient funds to cover the amount but has a companion savings account, it tries to withdraw the excess from there. If the companion account has insufficient funds, this method returns the appropriate error message. If the companion account has enough funds, the excess is withdrawn from there, and the checking account balance becomes zero (0). If successful, the account records the withdrawal by creating a transaction object containing the date and time, posted balance, and the transaction type and amount.

Figure 8



4. OBJECT STORAGE AND PERSISTENCE

A program will create a large amount of data throughout its execution; each item of data will have a different lifetime. These lifetimes are categorized into six, namely:

1. Transient results to the evaluation of expressions.
2. Variables involved in procedure activation (parameters and variables with a localized scope).

3. Global variables and variables that are dynamically allocated.
4. Data that exist between the executions of a program.
5. Data that exist between the versions of a program.
6. Data that outlive a program.

The first three categories are transient data, data that cease to exist beyond the lifetime of the creating process. The other three are non-transient, or persistent, data.

Programming languages provide excellent, integrating support for the first three categories of transient data. The other three categories can be supported by a DBMS or a file system.

Objects have a lifetime and are created explicitly to exist for a period of time (during the application session). An object can persist beyond application session boundaries, during which the object is stored in a file or a database. A file or a database can provide a longer life for objects – longer than the duration of the process in which they were created. From language perspective, this characteristic is called persistence. Essential elements in providing a persistent store are:

- Identification of persistent objects or reachability (object ID).
- Properties of objects and their interconnections. The store must be able to co-herently manage non-pointer and pointer data.
- Scale of the object store. The object store should provide a conceptually infinite store.
- Stability. The system should be able to recover from unexpected failures and return to a recent self-consistent state.

An *Object-Oriented Database Management System* (*OODBMS*) is a system that provides database-like support for objects (i.e., encapsulation and operations). It is a persistent, shareable repository, and manager of an object-oriented database. The database itself is a collection of objects defined by an object-oriented data model (objects that capture the semantics of objects supported in object-oriented programming). While semantic models are oriented towards structural abstraction and data representation, object-oriented models are concerned with behavioral abstraction and data manipulation.

An OODBMS attempts to extend flexibility to "unconventional" data and associated processing tasks (including text, graphics, and voice data) that cannot be handled and integrated by conventional database systems.

The basic idea of an object-oriented database is to represent an item in the real world with a corresponding item in the database. Coupling an object-oriented database with an object-oriented programming style results in the virtual elimination of the semantic gap between a program and its supporting data. Three levels of "object orientation" have been defined by Dittrich and Manola:

- ***Structurally object-oriented*:** The data model allows definitions of data structures to represent entities of any complexity (*complex objects*).
- ***Operationally object-oriented*:** The data model includes generic operators to deal with complex objects in their entirety.
- ***Behaviorally object-oriented*:** The data model incorporates features to define arbitrarily complex object types together with a set of specific operators (*abstract data types*). Instances can only be used to call these operators.

Key features of systems that truly support the object-oriented philosophy as described by Cox:

- ***Inheritance*:** Instance variables, class variables and methods are passed down from a superclass to its subclasses. A technique that allows new classes to be built on top of older, less specialized classes instead of being rewritten from scratch.

- ***Information Hiding*:** The state of a software module is contained in *private variables*, visible only from within the scope of the module. Important for ensuring reliability and modifiability of software systems by reducing inter-dependencies between components.
- ***Dynamic Binding*:** The responsibility for executing an action on an object resides with the object itself. The same message can elicit a different response depending upon the receiver.
- ***Encapsulation*:** A technique for minimizing interdependencies among separately written modules by defining strict external interfaces. The consumer no longer applies operators to operands while taking care that the two are type compatible.
- ***Data Abstraction*:** The behavior of an abstract data object is fully defined by a set of abstract operations defined on the object. Objects in most object-oriented languages are abstract. Data abstraction can be considered as a way of using information hiding.
- ***Object Identity*:** Each object has a unique identifier independent of the values of properties.

Other notions currently associated with the object-oriented approach include messages, overloading, late binding, and interactive interfaces with windows, menus and mice.

4.1 Advantages

Object-oriented programming and database management systems offer a number of important advantages over traditional control/data oriented techniques, including, as described by Manola, King and Thomas:

- The modeling of all conceptual entities with a single concept, the object.
- The notion of a class hierarchy and inheritance of properties along the hierarchy.
- The inheritance mechanism of object-oriented languages which allows code to be reused in a convenient manner.
- Facilitates the construction of software components that loosely parallel the application domain.
- Encourages the use of modular design.
- Provides a simple and expressive model for the relationship of various parts of the system's definition and assists in making components reusable or extensible.
- Views a database as a collection of abstract objects, rather than a set of flat (though possibly interrelated) tables.
- Captures integrity constraints more easily.
- Offers a unifying paradigm in the database, programming language, and artificial intelligence domains.
- The ability to represent and reference objects of complex structures resulting in increased semantic content of databases.
- Provides a more flexible modeling tool.
- Allows protection and security mechanisms to be based on the notion of an object, a more natural unit of access control.
- Can provide version control functions.
- Incorporation of software engineering principles such as data abstraction and information hiding.

4.2 Disadvantages

Despite its many advantages, the object-oriented view is not perfect. Though there are several drawbacks to OO systems as listed below, most are a direct result of its relative infancy and lack of development. Most of these problems are expected to be resolved as the model matures:

- The object-oriented paradigm lacks a coherent data model. There is currently no established standard for object-oriented design, methodology, language facilities, etc.
- Research into the structures for efficient object storage is in the early stages.
- Use of an object-oriented database from a conventional data processing language is difficult because of the semantic gap.
- In current environments, the run-time cost of using object-oriented languages is high.
- Object-oriented database management systems provide only limited support for integrating data in existing, possibly heterogeneous, databases.
- Typical OODBMS's do not integrate existing database application code with the object methods maintained by the system. This is similar to the previous point but concerns procedures rather than data.
- Many object-oriented database systems do not support the strict notion of metaclass.
- Some OODBMS's do not emphasize the efficient processing of set-oriented queries, although most of the commercial OODBMS's provide some form of query facility.
- Object-Oriented Programming/Processing (OOP) can be very memory intensive.
- There is no adequate, accepted, standard query language based on the OO view of data.

Many of these issues are well on their way to resolution. For example, the Unified Modeling Language (UML) seems to be emerging as a standard, filling the gap mentioned in the first bullet above. The OO concept has already made great strides. As the paradigm matures, most, if not all, of these issues are expected to be resolved.

5. USER INTERFACE DESIGN

Once the analysis is complete, we can start designing the user interface for the objects and determine how these objects are to be presented. The goal of User Interface (UI) is to display and obtain needed information in an accessible, and efficient manner. A design is required to provide users the information they need and clearly tell them how to successfully complete a task. A well designed UI has visual appeal that motivates users to use the application. It should use the limited screen space efficiently.

5.1 Designing View Layer Classes

The distinguishing characteristic of view layer objects or interface objects is that they are the only exposed objects of an application with which users can interact. View layer classes or interface objects are objects that represent the set of operations in the business that users must perform to complete their tasks. Objects that have direct contact with the outside world are visible in interface objects.

The view layer objects are responsible for two major aspects of the applications:

- ***Input – Responding to user interaction:*** The user interface must be designed to translate an action by the user, such as clicking on a button or selecting from a menu, into an appropriate response. That response may be to open or close another interface or to send a message down into the business layer to start some business process.
- ***Output – Displaying or printing business objects:*** This layer must paint the best picture possible of the business objects for the user.

The process of designing view layer classes is divided into four major activities:

i. ***Macro Level UI Design Process – Identifying View Layer Object:*** This activity, for the most part, takes place during the analysis phase of system development. The main objective of the macro process is to identify classes that interact with human actors by analyzing the use cases developed in the analysis phase. Each use case involves actors and the task they want the system to do. These use cases should capture a complete, unambiguous, and consistent picture of the interface requirements of the system. Use cases concentrate on describing what the system does rather than how it does it by separating the behavior of a system from the way it is implemented, which requires viewing the system from the user's perspective rather than that of the machine. In this phase, one also needs to address, the issue of how the interface must be implemented. Sequence or collaboration diagrams can help by allowing zooming in on the actor-system interaction and extrapolating interface classes that interact with human actors; thus, assisting in identifying and gathering the requirements for the view layer objects and designing them.

ii. ***Micro Level UI Design Activities:***

- *Designing the View Layer Objects by Applying Design Axioms and Corollaries:* In designing view layer objects, how to use and extend the components are decided, so that they best support application-specific functions and provide the most usable interface.
- *Prototyping the View Layer Interfac:* After defining design model, a prototype of some of the basic aspects of the design is prepared. Prototyping is particularly useful early in the design process.

iii. ***Testing Usability and User Satisfaction:*** "One must test the application to make sure it meets the audience requirements. To ensure user satisfaction, one must measure user satisfaction and its usability along the way as the UI design takes form. Usability experts agree that usability evaluation should be part of the development process rather than a post-mortem or forensic activity. Despite the importance of usability and user satisfaction, many system developers still fail to pay adequate attention to usability, focusing primarily on functionality.

iv. ***Refining and Iterating the Design:*** This activity is essential in order to incorporate any changes suggested by the users.

5.2 Macro-Level Process: Identifying View Classes by Analysing Use Cases

The interface object handles all communication with the actor but processes no business rules or object storage activities. The interface object will operate as a buffer between the user and the rest of the business objects. The interface object is responsible for behavior related directly to the tasks involving contact with actors. Interface objects are unlike business objects, that lie inside the business layer and involve no interaction with actors. For example, computing employee overtime is an example of a business object service. The data entry for the employee overtime is an interface object.

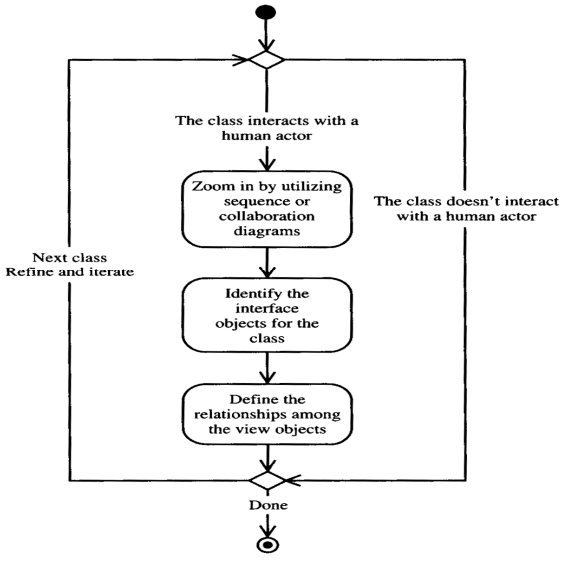
The interface object in the process, has responsibility for those tasks that come into direct contact with the user. The first step is to begin with the use cases, which help to understand the users' objectives and tasks. Different users have different needs; for example, advanced, or "power", users want efficiency whereas other users may want ease of use. Similarly, ones with disabilities or in an international market have still different requirements. The challenge is to provide efficiency for advanced users without introducing complexity for less-experienced ones. Developing use cases for advanced as well as less-experienced users might lead to solutions such as developing shortcuts to support more advanced users.

The view layer macro process consists of two steps:

1. For every class identified, it is determined whether the class interacts with human actor. If so, the following is performed; otherwise, it is moved to the next class:
   - *Identify the view (interface) objects for the class:* Zoom in on the view objects by utilizing sequence or collaboration diagrams to identify the interface objects, their responsibilities, and the requirements for this class.
   - *Define the relationships among the view (interface) objects:* The interface objects, like access classes, for the most part, are associated with the business classes. Therefore, one can let business classes guide in defining the relationships among the view classes. Furthermore, the same rule as applies in identifying relationships among business class objects also applies among interface objects.
2. **Iterate and Refine:** The advantage of utilizing use cases in identifying and designing view layer objects is that the focus centers on the user, and including users as part of the planning and design is the best way to ensure that they are important players in the design of objects. After identifying the interface objects, it is necessary to identify the basic components or objects used in the user tasks and the behavior and the characteristics that differentiate each kind of objects, including the relationships of interface objects to one another and to the user. The relationships among view class and business class objects is opposite of that among business class and access class objects. The interface object handles all communication with the user but does not process any business rules; that will be done by the business objects.

Effective interface design is more than just following a set of rules. It also involves early planning of the interface and continued work through the software development process. The process of designing the user interface involves clarifying the specific needs of the application, identifying the use cases and interface objects, and then devising a design that best meets users' needs.

Figure 9: The Macro Level Design Process



5.3 Micro Level Process

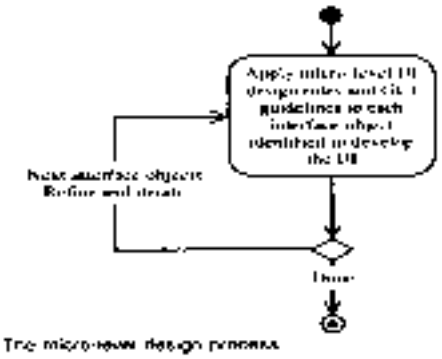
A ***user-centered interface*** replicates the user's view of doing things by providing the outcomes users expect for any action. For example, the goal of the relationships among business, access, and view objects. In some situations the view class can become a direct aggregate of the access object, as when designing a Web interface that must communicate with an application/Web server through access objects.

Application is to automate what was a paper process, and then the tool should be simple and natural. Design the application so that it allows users to apply their previous real-world knowledge of the paper process to the application interface. The design then can support this work environment and goal. The main goal of view layer design is to address users' needs.

The following is the process of designing view (interface) objects:

1. For every interface object identified in the macro UI design *apply micro-level UI design rules and corollaries to develop the UI.* Apply design rules and GUI guidelines to design the UI for the interface objects identified.
2. Iterate and refine.

Figure 10



5.4 The Purpose of a View Layer Interface

UI can employ one or more windows. Each window should serve a clear and specific purpose. These are commonly used for the following purposes:

- *Forms and Data Entry Windows:* Data entry windows provide access to data that users can retrieve, display, and change in the application. If a window serves multiple purposes, create a separate one for each.
- *Dialog Boxes:* It displays status information to ask users to supply information or make a decision before continuing with a task.
- *Application Window:* An application window is a container of application objects or icons.

5.4.1 PROTOTYPING THE USER INTERFACE

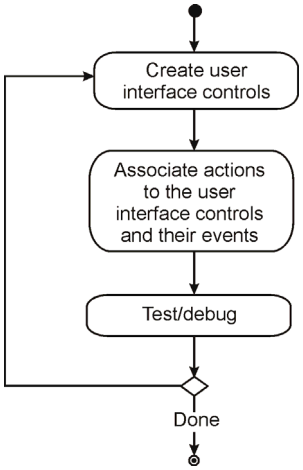
Rapid prototyping encourages the incremental development approach. Prototyping involves a number of iterations. Through each iteration, we add a little more to the application, and as we understand the problem a little better, we can make more improvements.

Visual and rapid prototyping is a valuable asset in many ways. First it provides an effective tool for communicating the design. Second, it helps to define task flow and better visualize the design. Finally, it provides a low-cost vehicle for getting user input on a design.

Creating a user interface generally consists of three steps:

1. Create the user interface objects.
2. Link or assign the appropriate behaviors or actions to these user interface objects and their events.
3. Test, debug and then add more by going back to step 1.

Figure 11



5.5 Case Study: Designing User Interface for the ViaNetBank ATM

Here, we are designing a GUI interface for the ViaNet Bank ATM for two reasons:

1. The ViaNet bank wants to deploy touch-screen instead of conventional ATM machines.
2. In the near future, ViaNet wants to create on-line banking, where customers can be connected electronically to the bank via Internet and conduct most of their banking needs.

5.5.1 The View Layer Macro Process

The first step here is to identify the interface objects, their requirements, and their responsibilities by applying the macro process to identify the view classes. When creating user interfaces for a business model, it is important to remember the role that view objects play in an application. The interface should be designed to give the user access to the business process modeled in the business layer.

For every class identified (so far we have identified the following classes: Account, ATMMachine, Bank, BankDB, CheckingAccount, SavingsAccount, and Transaction),

- Determine if the class interacts with a human actor. The only class that interacts with a human actor is ATMMachine.
- Identify the interface objects for the class. The next step is to go through the sequence and collaboration diagrams to identify the interface objects, their responsibilities and the requirements for this class.

We have already identified the scenarios or use cases for the ViaNet bank. The various scenarios involve CheckingAccount, SavingsAccount, and general bank Transactions (see figures). These use cases interact directly with actors:

1. Bank transaction
2. Checking transaction history
3. Deposit checking
4. Deposit savings
5. Savings transaction history
6. Withdraw checking
7. Withdraw savings
8. Valid/invalid PIN

Based on these use cases, we have identified eight view or interface objects. The sequence and collaboration diagrams can be very useful here to help us better understand the responsibility of the view layer objects. To understand the responsibilities of the interface objects, we need to look at the sequence and collaboration diagrams and study the events that these interface objects must process or generate. Such events will tell us the makeup of these objects. For example, the PIN validation user interface must be able to get a user's PIN number and check whether it is valid.

Define Relationships among View (Interface) Objects

Now, we need to identify the relationships among these view objects and their associated business classes.

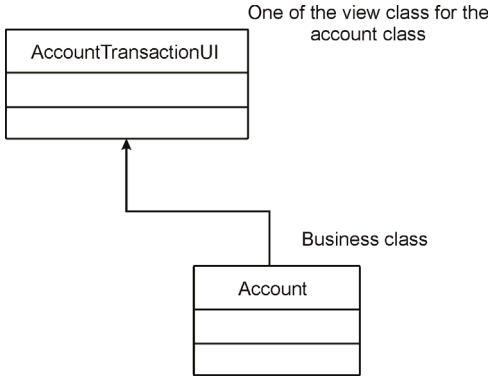
So far, we have identified eight view classes:

1. AccountTransactionUI (for a bank transaction)
2. CheckingTransactionHistoryUI
3. SavingsTransactionHistoryUI
4. BankClientAccessUI (for validating a PIN code)
5. DepositCheckingUI
6. DepositSavingsUI
7. WithdrawCheckingUI
8. WithdrawSavingsUI.

The first three transaction view objects basically do the same thing; display the transaction history on either a checking or savings account. Therefore, we need only one view class for displaying transaction history, and let us call it AccountTransactionUI.

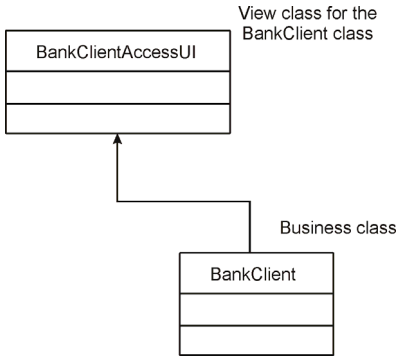
The **AccountTransactionUI view class** is the account transaction interface that displays the transaction history for both savings and checking accounts. The following figure depicts the relation between the AccountTransactionUI and the account class. The relationship between the view class and business object is opposite of that between business class and access class. We know that the interface object handles all communications with the user but processes no business rules and lets that work be done by the business objects themselves. In this case, the account class provides the information to AccountTransactionUI for displaying to the users (figure 12).

Figure 12



The BankClientAccessUI view class provides access control and PIN code validation for a bank client. It is shown in the following figure:

Figure 13



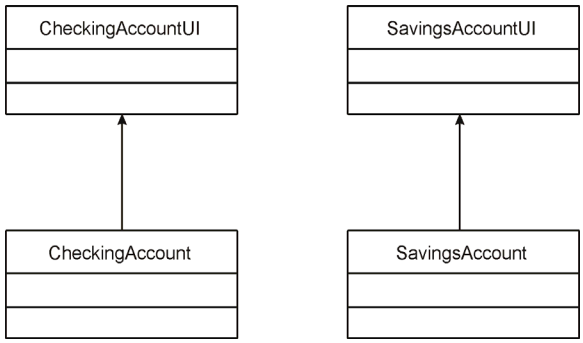
The four remaining view objects are the DepositCheckingUI view class (interface for deposit to checking accounts), DepositSavingsUI view class (interface for deposit to savings accounts), WithdrawSavingsUI view class (interface for withdrawal from savings accounts), and WithdrawCheckingUI view class (interface for withdrawal from checking accounts).

Iterate and Refine

This is the final step. Through the iteration and refinement process, we notice that the four classes DepositCheckingUI, DepositSavingsUI, WithdrawSavingsUI, and WithdrawCheckingUI basically provide a **single service**, which is getting the amount of the transaction (whether the user wants to withdraw or deposit) and sending appropriate message to SavingsAccount or CheckingAccount business classes. Therefore, they are good candidates to be combined into two view classes, one for CheckingAccount and one for SavingsAccount (by following UI rule 3). Both CheckingAccountUI and SavingsAccountUI allow users to deposit money to or withdraw money from checking and savings accounts.

The CheckingAccountUI view class provides the interface for a checking account deposit or withdrawal. The SavingsAccountUI view class provides the interface for a savings account deposit or withdrawal. These two view classes are shown in the following figure:

Figure 14



Finally, we need to create one more view class that provides the main control or the main UI to the ViaNet bank system. The MainUI view class provides the main control interface for the ViaNet bank system.

5.5.2 THE VIEW LAYER MICRO PROCESS

Based on the outcome of the macro process, we have the following view classes:

1. BankClientAccessUI
2. MainUI
3. AccountTransactionUI

4. CheckingAccountUI
5. SavingsAccountUI.

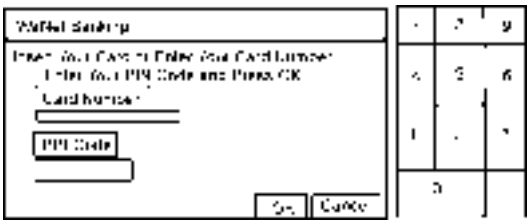
For every interface object identified in the macro UI design process,

- Apply micro-level UI design rules and corollaries to develop the UI. We need to go through each identified interface object and apply design rules (such as making the UI simple, transparent, and controlled by the user) and GUI guidelines to design them.
- Iterate and refine.

5.5.3 THE BANKCLIENTACCESSUI INTERFACE OBJECT

The BankClientAccessUI provides clients access to the system by allowing them to enter their PIN for validation. The BankClientAccessUI is designed to work with a card reader device, where the user can insert the card and the card number should be displayed automatically in the card number field. In a situation where there is no card reader, such as on-line banking (e.g., user wants to log onto the system from home), the user must enter his or her card number. This is shown in figure 15.

Figure 15



5.5.4 THE MAINUI INTERFACE OBJECT

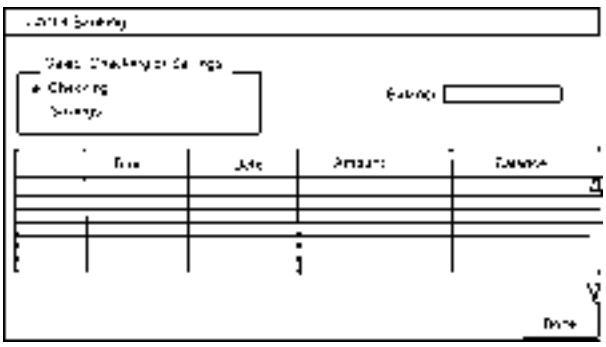
The MainUI provides the main control to the ATM services. Users can select to deposit money to savings or checking account, withdraw money from savings or checking account, inquire as to a balance or transaction history, or quit the session. This is shown in figure 16.

Figure 16

5.5.5 THE ACCOUNTTRANSACTIONUI INTERFACE OBJECT

The AccountTransactionUI interface object will display the transaction history of either a savings or checking account. The user must select the account type by pressing the radio buttons. Figure 17 below display the account balance inquiry and transaction history interface.

Figure 17



5.5.6 DEFINING THE INTERFACE BEHAVIOR

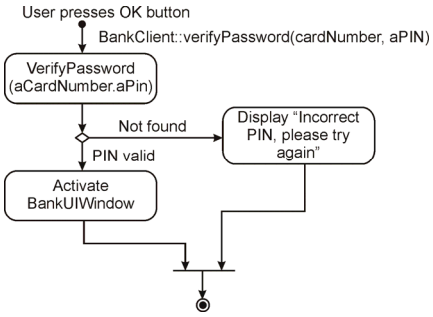
The role of a view layer object is to allow the users to manipulate the business model. The actions a user takes on a screen should be translated into a request to the business object for some kind of processing. When the processing is completed, the interface can update itself by displaying new information, opening a new window, or the like.

Defining behavior for an interface consists of identifying the events to which you want the system to respond and the actions to be taken when the event occurs. Both GUI and business objects can generate events when something happens to them (for example, a button is pushed or a client's name changes). In response to these events, one can take requisite actions. An action is a combination of an object and a message sent to it.

5.5.7 IDENTIFYING EVENTS AND ACTIONS FOR THE BANKCLIENTACCESSUI INTERFACE OBJECT

When the user inserts his or her card, types in a PIN, and presses the OK button, the interface should send the message BankClient::verifyPassword to the object to identify the client. If the password is found correct, the MainUI should be displayed and provide users with the ATM services; otherwise, an error message should be displayed. The UML activity diagram of BankClientAccessUI events and actions are shown in figure 18.

Figure 18



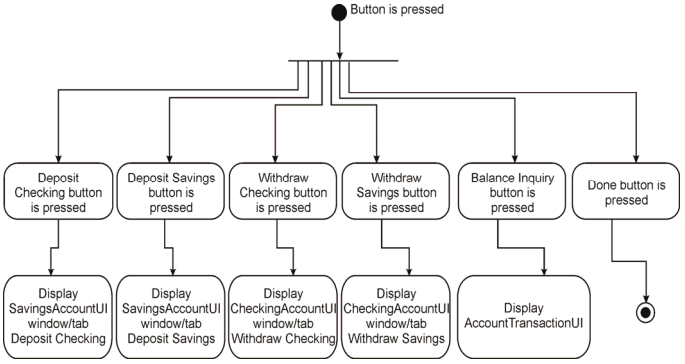
5.5.8 IDENTIFYING EVENTS AND ACTIONS FOR THE MAINUI INTERFACE OBJECT

From this interface, the user should be able to do the following:

- Deposit into the checking account by pressing the Deposit Checking button.
- Deposit into the savings account by pressing the Deposit Savings button.
- Withdraw from the savings account by pressing the Withdraw Savings button.
- Withdraw from the checking account by pressing the Withdraw Checking button.
- View balance and transaction history by pressing the Balance Inquiry button.
- Exit the ATM by pressing Done.

Figure 19 shows the MainUI events and actions.

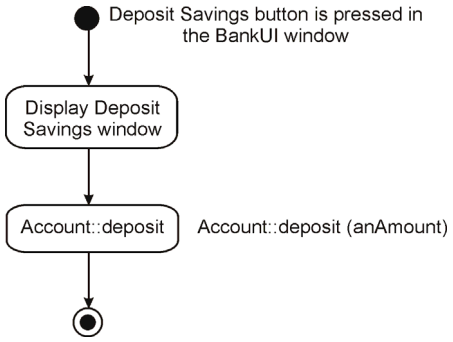
Figure 19



5.5.9 IDENTIFYING EVENTS AND ACTIONS FOR THE SAVINGSACCOUNTUI INTERFACE OBJECT

The SavingsAccountUI has two tabs. First, the SavingsAccountUI opens the appropriate tab. For example, if the user selects the Deposit Savings from the MainUI, the SavingsAccountUI will display the Deposit Savings tab. Figure 20 shows the deposit savings.

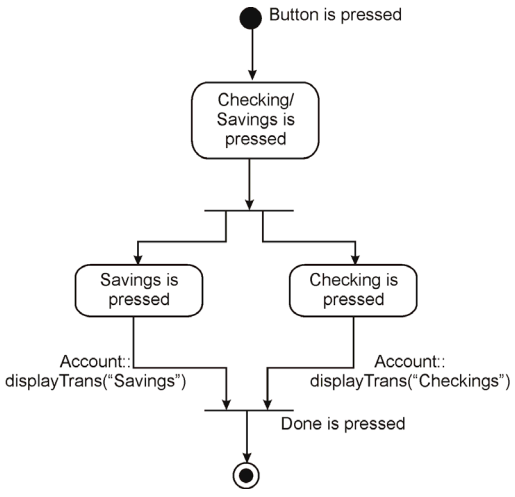
Figure 20



5.5.10 IDENTIFYING EVENTS AND ACTIONS FOR THE ACCOUNTTRANSACTIONUI INTERFACE OBJECT

A user can select either savings or checking account by pressing on the Savings or Checking radio button. The system then will display the balance and transaction history for the selected account type. The default is the checking account, so when the AccountTransactionUI window is opened for the first time, it will show the checking account history. Pressing on the savings account radio button will cause it to display the savings account balance and history. To close the display and get back to MainUI, the user presses the Done button. This is shown in figure 21.

Figure 21



Notice that here we are assuming that the account has a method called displayTrans, which takes a string parameter for type of account (Savings or Checking) and retrieves the appropriate transaction. Since we did not identify or design it, we need to develop it here. This occurs quite often during software development, which is why the process is iterative.

SUMMARY

- A model is a simplified representation of reality, simplified because reality is too complex or large and much of the complexity actually is irrelevant to the problem being described or solved.
- The unified modeling language was developed by Booch, Jacobson, and Rumbaugh. The UML encompasses the unification of their modeling notations.
- The UML class diagram is the main static structure analysis diagram for the system. It represents the class structure of a system with relationships between classes and inheritance structure. The class diagrams are developed through use-case, sequence and collaboration diagrams.
- The use-case diagram captures information on how the system or business works or how one wishes it to work. It is a scenario-building approach in which one models the processes of the system. It is an excellent way to learn the object-oriented analysis of the system.
- The UML sequence diagram is for dynamic modeling, where objects are represented as vertical lines and message passed back and forth between the objects are modeled by horizontal vectors between the objects.
- The UML collaboration diagram is an alternative view of the sequence diagram, showing in a scenario how objects interrelate with one another.
- State chart diagrams, another form of dynamic modeling, focus on the events occurring within a single object as it responds to messages; and activity diagram is used to model an entire business process. Thus, an activity model can represent several different classes.
- Implementation diagrams show the implementation phase of systems development, such as the source code and run-time implementation structures. The two types of implementation diagrams are component diagrams, which show the structure of the code itself, and deployment diagrams, which show the structure of the run time system.
- Stereotypes represent a built-in extensibility mechanism of the UML. User-defined extensions of the UML are enabled through the use of stereotypes and constraints.
- UML graphical notations can be used not only to describe the system's components but also to describe a model itself; this is known as a meta-model. It is a model of modeling elements. The purpose of the UML meta-model is to provide a single, common, and definitive statement of the syntax and semantics of the elements of UML.

Glossary

Actor	:	An actor is someone or something that interacts with the system i.e., the actor is a type (a class), not an instance.
Activity Diagram	:	Activity diagrams are in the form of flow charts that describe the dynamic nature of a system by modeling the flow of control from one activity to another.
Attribute	:	An attribute represents the state of an object.
Association	:	An association is a relationship between instances of the two classes.
Aggregations	:	Aggregations provide a means of showing that the entire object is composed of the sum of its parts.
Block Sequence Code	:	In a block sequence code, a series of consecutive numbers and/or letters is divided into blocks, each one reserved for identifying a group of items with a common characteristic.
Bottom-up Computer Program Development	:	This is the older method for the development and testing of computer programs and this method contains a hierarchical structure within which the lowest level programs are tested individually and then combined into higher level modules, which are tested next.
Booch Methodology	:	It is a widely used object-oriented method that helps to design the system using the object paradigm. It covers the analysis and design phases of an object-oriented system.
CASE tool	:	A CASE tool is a computer-based product aimed at supporting one or more software engineering activities within a software development process.
Changeover	:	Changeover is the process of putting the new security system online and phasing out the old system. In Parallel changeover, outputs can be compared to ensure that the new system is functioning correctly.
Charts	:	Charts are used to analyze data graphically.
Classes, Responsibilities and Collaborators (CRC)	:	CRC is a technique used for identifying classes' responsibilities and their attributes and methods. A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.
Closed Systems	:	Systems that do not interact with their environment are said to be closed systems.
Code Dictionary	:	Code dictionary provides for the translation of a code into the corresponding data item or can be used to determine of the code for a particular data item.
Code Plan	:	Code plan identifies the particular characteristic that needs to be incorporated within the code.
Cohesion	:	The term cohesion refers to the strength of dependencies within a subsystem.
Communication Process	:	Communication process involves sending and receiving messages i.e., it is the process of transferring information from one point to another point.

Computer Print Chart : A computer print chart is a part of computer printer output and the detailed description of outputs includes the identification of the print positions to be used for the title, column headings, detailed data, and total.

Computer Program Function : A function is a part of computer program consisting of a sequence of steps and it performs a given task and returns the result to the main program.

Computer Program Testing : Computer programs are tested in planned, top-down sequence that includes structured walk-throughs. The testing continues until the individual programs can be assembled as a component that can be tested as a unit.

Computer Output : Anything that comes out of a computer. Output can be meaningful information or gibberish, and it can appear in a variety of forms – as binary numbers, as characters, as pictures, and as printed pages. Output devices include display screens, loudspeakers and printers.

Context Diagram : It is a data flow diagram showing data flows between a generalized application within the domain and the other entities and abstractions with which it communicates.

Control : It is an action taken to bring the difference between an actual output and a desired output within an acceptable range.

Conversion : Conversion is the process of performing all of the operations that result directly in the turnover of the new system to user.

Corollary : A corollary is a proposition that follows from an axiom or another proposition that has been proven.

Coupling : The term coupling refers to the degree of dependency between two subsystems.

Data Dictionary : It is a catalog or a repository of data about data i.e., it describes about the elements in a system.

Database Design : Database design is the process of producing a detailed data model of a database.

Data Flow Diagram : It is a structured, diagrammatic technique for showing the functions performed by a system and the data flowing into, out of, and within it.

Data Stores : These are repositories of data in the system. A data store is depicted by two parallel lines or open-ended rectangles.

Database Management System : A Database Management System (DBMS) is defined as a collection of interrelated data and a set of programs to access the data.

Denormalization : It is a process of splitting or combining normalized relations into physical tables based on affinity of use of rows and fields.

Decision Table : It is a table of possibilities that are to be considered in the definition of a problem, together with the actions to be taken.

Decision Tree : It is a tree like structure that represents the various conditions and the subsequent possible actions. It also shows the priority in which the conditions are to be tested or addressed.

Decision Support System : A decision support system is a computer program application that analyzes business data and presents it so as to make the job for users much easier in dealing with any business decision.

Design Phase	:	Design phase is the life-cycle phase in which the detailed design of the system selected in the study phase is carried out.
Design Specification	:	Design specifications are the specifications that describe the features of a system and its components.
Desktop Publishing	:	Desktop publishing is the use of a personal computer or workstation to produce high-quality printed documents.
Development Phase	:	The development phase is the third of the four systems development life cycle phases. It is the life-cycle phase in which the system is constructed according to the design specifications.
Direct Access	:	Direct access is based on a disk model of a file. For direct access, the file is viewed as a numbered sequence of blocks or records.
Ergonomics	:	The word ergonomics refers to the physical factors of an information system that affect the performance, comfort, and satisfaction of direct users.
Error-handling procedures	:	Error-handling procedures are the actions that are taken when unexpected results occur.
Entity-Relationship (E-R) Diagram	:	It represents entities in the business environment, the relationships among the entities, and the attributes or properties of both the entities and their relationships.
Entity-Relationship (E-R) Model	:	It is a conceptual data model that views the real world as entities and relationships.
Expert System	:	An expert system is a computer program that simulates the judgment and behavior of an organization that has expert knowledge and experience in a particular field.
External Entity	:	A source or destination of data considered to be external to the system described.
Fact Finding Technique	:	The specific methods analysts use for collecting data about requirements are called fact-finding techniques. It is also known as information gathering or data collection.
Feasibility Analysis	:	Feasibility Analysis is the identification of candidate systems and the selection of the most feasible system.
Feedback	:	It is the process of comparing an actual output with a desired output for the purpose of improving the performance of the system.
Field	:	A physical part of a database that can be packed with several data items; the smallest unit of named application data recognized by system software.
First Normal Form (1NF)	:	The first step in normalizing a relation in data used in a database so that it contains no repeating groups.
Flow Chart	:	It is a pictorial representation that uses predefined symbols to describe data flow and processing in either a business system or the logic of a computer program.

Form : A form is the physical carrier of data. It can carry instructions for action. It is classified by what it does in the system.

Fourth Generation Programming (4GL) : A Fourth-Generation Programming Language (4GL) is a programming language or programming environment designed with a specific purpose in mind, such as the development of commercial business software.

Formal Test Planning : A formal test plan is a document that provides and records important information about a test project.

Functional Dependency : Functional Dependency (FD) is a constraint between two sets of attributes in a relation from a database. Given a relation *R*, a set of attributes *X* in *R* is said to functionally determine another attribute *Y*, also in *R*, (written $X \rightarrow Y$) if and only if each *X* value is associated with precisely one *Y* value. Customarily we call *X* the *determinant set* and *Y* the *dependent attribute*. T

Goal : It is a very broadly stated purpose. Some examples are, the goals of making profit; the goal of educating students.

Group Classification Codes : It designates major, intermediate and minor data classification by successively assigning lower orders of digits.

Identification Code : It is a collection of characters used to identify a record of data.

Implementation Planning : Implementation planning is the first activity of the development phase. After the initiation of the development phase is approved, implementation planning begins.

Indexed File Organization : An indexed file contains records ordered by a *record key*. For that reason, a second file is created that contains an index or key for every record and the record's location in the file.

Indexed Sequential File Organization : Indexed sequential files are a different kind of indexed files. The indexes are pointers to the data file and the data is stored sequentially in order by the key.

Information Resource Management : It is the concept that information is a major corporate resource and must be managed using the same basic principles used to manage other assets. This includes the effective management and control of data/information as a shared resource to improve its availability, accessibility and utilization.

Information Service Request (ISR) : An Information Service Request (ISR) is a formal request from a user group for support from the information services organization. It provides for statements of objectives and anticipated benefits and for the description of inputs and outputs.

Information Systems : Information systems are the computer systems designed to store, transmit, retrieve, manipulate and display information used in one or more business processes.

Interview : It is a common technique used to collect information from key stakeholders in a software project. It can be performed in small or large groups and in formal or informal settings.

Interface : A point at which the system meets its environment is called an interface.

Joint Application Design : A Joint Application Development (JAD) is a methodology that involves the end-users in the design and development of an application to work more efficiently within a short period of time.

Key	:	A key is one of the data items in a record used for identifying a record.
Management Information System	:	A computer system designed to help managers plan and direct business and organizational operations.
Normalization	:	It is a process of converting complex data structures into simple, and stable data structures.
Object Modeling Technique	:	A technique used for identifying and modeling all the objects making up a system.
Objectives	:	These are concrete and specific accomplishments necessary for the achievements of goals. For example, an automobile manufacturer must have as an objective the production of a competitive product in order to achieve a profit goal.
Organization Chart	:	It is a flowchart that identifies the organizational elements of a business and displays areas of responsibility and lines of authority.
Open Systems	:	The systems that interact with their surrounding i.e., receive input and produce output, are said to be open systems.
Performance Review Board (PRB)	:	A performance review board is established for ensuring system integrity. In the PRB, both the user and information systems are presented. A performance review board is a user oriented board and is headed by the principal user.
PERT	:	PERT is a management planning and analysis tool that uses a graphical display, called a network, to show relationships between tasks that must be performed to accomplish an objective.
Prototype	:	A prototype is a basic model of an information system, especially designed for demonstration purposes or as part of a development process.
Physical System	:	A physical system is an interconnection of physical components that perform a specific function.
Post Installation Review (PIR)	:	A Post Installation Review should be conducted when there are remaining discrepancy reports, deferred requirements for future maintenance release, or any issues related to the system operation.
Presentation	:	It is the process of presenting the content of a topic to an audience.
Principal User	:	The principal user is the person who will accept or reject the computer based business system. The principal user may be the person who issues the project directive.
Process	:	It represents activities in which data is manipulated by being stored or retrieved or transferred in some way.
Project Directive	:	Project directive is an authorized document that reflects the results of discussions and decisions made during the review.
Primary Key	:	A primary key is one that uniquely identifies a record.
Qualifier	:	A qualifier is an association attribute.
Questionnaire	:	Questionnaire is a form containing a set of questions, especially one addressed to a statistically significant number of subjects as a way of gathering information for a survey.

Relational Database : A relational database is a database that groups data using common attributes found in the data set.

Recurring Data Analysis Sheet : Recurring data analysis sheet is a form which is prepared with the document names and identifying numbers.

Response Time : Response time is the time that elapses between the release of input data by a user and receipt of computer output.

Run-time procedures : The run-time procedures are the steps and actions taken by the system operators or the end-users who are interacting with the system to achieve the desired results.

Second Normal Form (2NF) : When normalizing data for a database, the analyst ensures that all the non-key attributes are fully dependent on the primary key. All partial dependencies are removed and placed in another relation.

Secondary key : A key that cannot uniquely identify a record can be used to select a group of records that belong to a set.

Sequential Access : In sequential access, information in the file is processed in order, one record after the other.

Sequential File Organization : A sequential file contains records organized in the order they were entered (the first record written is the first record in the file, the second record written is the second record in the file, and so on).

Software Requirements Specification (SRS) Software Requirements Specification (SRS) is a complete description of the behavior of the software of the system to be developed. It includes a set of use cases that describe all the interactions the users will have with the software.

Stakeholder : A stakeholder is a person who has a share or an interest in an enterprise i.e., a person holding property or owing an obligation that is claimed by two or more adverse claimants and who has no claim to or interest in the property or obligation.

Standards Manual : Standards are rules under which analysts, programmers, operators and other personnel in information service organization operate.

Structured English : It is the modified form of the English language used to specify the logic of information system processes.

Structure Chart : Structure Chart in software engineering and organizational theory is a chart that shows the breakdown of the configuration system to the lowest manageable levels. In structure chart each program module is represented by a rectangular box. Modules at the top level of the structure chart call the modules at the lower levels.

Structured Programming : Structured programming (sometimes known as *modular programming*) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify.

Structured Analysis : It is a set of techniques and graphic tools that allow the analysts to develop a new kind of system specification that is easily understandable to the user.

Structured Walk-through : Structured reviews are a technique used in developing efficient and reliable systems. A ‘structured walk-through’ is a technical review to assist the technical people working on a project.

Study Phase Report : The study phase report is a comprehensive report prepared for the user-sponsor of the system and presented at the conclusion of the study phase.

Study Phase Review : The study phase review is a review for presenting the results of the study phase activities and determining future action. It is attended by the principal users and managers who will be affected by the system.

System : An organized relationship among functioning units or components.

Systems Analysis : It is the process of gathering and interpreting facts, diagnosing problems, and using the information to recommend improvements to the system.

System Analyst : System Analyst is a person responsible for studying the requirements, feasibility, cost, design, specification, and implementation of a computer based system for an organization/business.

System Design : System design is one of the phases of System Development Life Cycle (SDLC). It is the process or art of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements.

System Flowchart : It is a flowchart that describes the data flow for a data processing system and provides a logical diagram of how the system operates.

System Development Life Cycle (SDLC) : It is a logical process by which systems analysts, software engineers, programmers, and end-users build information systems and computer applications to solve business problems and needs.

System Documentation : System documentation is the detailed information, in either written or computerized form, about a computer system, including its architecture, design, data flow and programming logic.

System Performance Definition : System Performance definition is the transition from a logical performance requirement to a physical one. The process includes the statement of general constraints, identification of specific objectives, and description of the outputs to be provided.

System Verification : System Verification is the process of determining if the system meets the conditions set forth at the beginning.

Third Normal Form (3NF) : A Table is in 3NF if and Only if both of the following conditions hold: (i) The relation R (table) is in second normal form, and (ii) Every non-prime attribute of R is non-transitively dependent (i.e., directly dependent) on every key of R.

Transitive dependency : A transitive dependency is one in which non-key attributes are dependent on other non-key attributes.

Throughput Time : Throughput time is the time required for work to flow through the machine room.

Top-down Computer Program Development : The top-down computer program development and testing approach is a structured technique that starts with a general description of the system and expands into successively greater levels of detail.

Turnaround Time : Turnaround time is the time that elapses between data arrival at the computing center and the availability of output for pick-up.

Use Case : A use case is a methodology used in system analysis to identify, clarify and organize system requirements.

Use Case Diagram A use case diagram provides a graphical overview of the functionality provided by a system in terms of actors, their goals, and any dependencies between those use cases.

User Interface Design : User interface design is the design of computers, appliances, machines, mobile communication devices, software applications, and websites with the focus on the user's experience and interaction.

User Turnover : It is the stage in the operation phase when the data processing department assumes full responsibility for the system.

Unified Modeling Language (UML) : Unified Modeling Language (UML) is a standardized general-purpose modeling language that provides a set of tools to document the object-oriented analysis and design of a software system.

Validation Software : Software that checks whether data input to the information system is valid or not.

Visual Display Terminal (VDT) : VDT (video display terminal or sometimes visual display terminal) is a term used, especially in ergonomic studies, for computer display.

Bibliography

1. Elias, M. Awad. *Systems Analysis and Design.* 2nd ed. New Delhi: Galgotia Publications (P) Ltd., 2002.

2. Igor Hawryszkiewycz. *Systems Analysis and Design*. 4th ed. New Delhi: Prentice-Hall of India Private Limited, 2002.

3. James, A.O' Brien. *Management Information Systems. Managing Information Technology in the E-Business Enterprise*. 5th ed. New Delhi: Tata McGraw Hill, 2002.

4. Kenneth E. Kendall, and Julie E. Kendall. *Systems Analysis and Design.* 4th ed. New Delhi: Prentice-Hall of India Private Limited, 2007.

5. Kenneth C. Laudon, and Jane P. Laudon. *Management Information Systems, Managing The Digital Firm.* 8th ed. New Delhi: Prentice-Hall of India Private Limited, 2003.

6. Marvin Gore and John W. Stubbe. *Elements of Systems Analysis.* 4th ed. New Delhi: Galgotia Publications (P) Ltd., 2003.

7. Rajaraman, V. *Analysis and Design of Information Systems.* 2nd ed. New Delhi: Prentice-Hall of India Private Ltd., 2006.

8. Uma G. Gupta. *Management Information Systems, A Managerial Perspective.* 1st ed., New Delhi: Galgotia Publications (P) Ltd., 1998.

Websites

- http://books.google.co.in/
- http://www.ipipan.gda.pl/~marek/objects/TOA/OOMethod/mcr.html
- http://www.jqjacobs.net/edu/
- http://www.scribd.com/
- www.answers.com
- www.ce.sharif.edu
- www.cio.gov.bc.ca
- www.rabbit.eng.miami.edu
- www.saintmarys.edu
- www.ublib.boulder.ibm.com
- www.webopebia.com