BSC COMPUTER SCIENCE,

THIRD YEAR, SEMESTER TWO

ICS 2312:SOFTWARE SYSTEM DEVELOPMENT

LECTURER: Dr. Ann Kibe

annkibe@gmail.com

0727684270

COURSE OUTLINE

OBJECTIVES

At the end of the course students should be able to demonstrate understanding of;

- Systems and system development methodologies
- effective methods for gathering essential information during system development
- effective methods for designing systems to solve problems effectively

Topics

1. Introduction to System Development; System Components, Classification of Systems

2. Systems Development Lifecycle (SDLC); Stages of the Lifecycle.

3. System Analysis; Preliminary Investigation; Feasibility; Requirement Specification, Functional, Non-functional;

4. Continuous Assessment Test (CAT) 1 and Assignment 1

5. Structured Systems Development Methods (SSDM);

6. System Design: Data Flow Diagrams (DFDs), Entity Relationship Diagrams (E-RDs).

7. Testing; Strategies of testing; Types of testing.

8 Implementation; Different implementation strategies..

9. Continuous Assessment Test (CAT) 2 and Assignment 3

10. System review and maintenance; Types of maintenance.

11. Revision

Assessment:

• Continuous Assessment Tests	20%

• Assignments	10%
• <u>End of Semester Exam</u>	<u>70%</u>
Total	**100%**

References;

1. Yourdon Systems Method-Model Driven Development;- Yourdon Inc
2. Systems Development: Methodologies, Techniques and Tools; 4th Ed. David Avison.
3. Systems Analysis and Design With Modern Methods; FertuckL..
4. Systems Analysis and Design Methods, (5th Ed); DittmanK. C., Bentley D. L., Whitten J. L
5. The Internet

SYSTEMS DEVELOPMENT

1.0 Introduction

When any new system is to be developed, it is a prerequisite that a robust planning process is entered into. This is because even when the development is very small, there is a real danger that the project will be a disaster unless it is well managed, so as to avoid the development being piecemeal and fragmented and thereby causing serious problems. When the project has been given the go-ahead, it is necessary to plan all of the resources, constraints, time and costs involved in the process

Systems development can generally be thought of as having two major components: Systems analysis and Systems design.

System design is the process of planning a new business system or one to replace or complement an existing system. But before this planning can be done, we must thoroughly understand the old system and determine how computers can best be used to make its operation more effective.

System analysis, then, is the process of gathering and interpreting facts, diagnosing problems, and using the information to recommend improvements to the system.

A ***system*** is a set of components that interact to achieve a common goal. One uses, observes, and interacts with many systems during daily activities. A driver drives in a highway system to reach a destination. Businesses use many types of systems. A billing system (such KPLC), allows a company to send invoices (electricity bills) and receive payments from customers. An inventory system keeps track of the items in a warehouse. A manufacturing system produces the goods that customers order. Through a payroll system, employees receive paychecks.

Very often, these systems are referred to as information systems. An ***information system*** is a collection of hardware, software, data, people, communications, and procedures that work together to produce quality information. The goal of an information system is to provide users with high-quality information so they can make effective decisions. An information system supports daily, short-term, and long-range activities of users. Some examples of users include store clerks, sales representatives, accountants, supervisors, managers, executives, and customers.

The kinds and types of information that users need often change over time. As a system user in a business organisation, one may someday participate in the modification of an existing system or the development of a new system. Thus, it is important that one understands the system development in the business organisation. Systems development is the activity of creating a new system or modifying an existing system. It refers to all aspects of the process --from identifying problems to be solved or opportunities to be exploited to the implementation and refinement of the chosen solution. Whatever its scope and objectives, a new system is an outgrowth of a process of organizational problem solving. A new system is developed as a solution to some type of problem or set of problems the organization perceives it is facing. The problem may be one where managers and employees realize that the organization is not performing as well as expected, or it may come from the realization that the organization should take advantage of new opportunities to perform more successfully.

When requirements change, the information system must meet the new requirements. In some cases, the current information system is modified; in other cases, an entirely new information system is developed. Understanding systems development is important to all professionals, not just those in the field of information systems.

One important thing to know about systems development is that a system is a socio-technical entity, an arrangement of both technical and social elements. The development of a new information system not only involves hardware, software, data, programmers and communications, but also includes changes in jobs, knowledge, skills, management, policies, processes, and organization. Often new systems mean new ways of doing things. Building a new system will affect the organization as a whole and change the decision-making process.

1.1 Systems Fundamentals

Data is an extremely valuable resource as it translates into information. Information is the backbone of any decision-making process that supports decision levels ranging from technical aspects to policy instruments for sustainable practices in organizations. The availability of relevant information is a major management concern. Consequently, there is a need to develop a system to support collection, processing and analysis of data.

The purpose of (information) system is to collect, process, store and disseminate information.

Defining "System" A system may be defined in one of many ways such as;

(a) A collection of components that work together to achieve a given objective/goal.

(b) An integrated set of interoperable elements, each with explicitly specified and bounded capabilities, working synergistically to perform value-added task.

(c) System is an integrated set of interoperable elements, working synergistically together to perform or accomplish a given set of goals.

(d) An information system is a set of hardware, software, data, human, and procedural components intended to provide the right data and information to the right person at the right time.

- By "an integrated set," we mean that a system, by definition, is composed of hierarchical levels of physical elements, entities, or components. i.e A system has at least two components, and every system component is connected directly or indirectly to every other system component
- By "interoperable elements," we mean that elements within the system's structure must be compatible with each other in form, fit, and function, for example. System elements include equipment (e.g., hardware and system, system, facilities, operating constraints, support), maintenance, supplies, spares, training, resources, procedural data, external systems, and anything else that supports mission accomplishment.
- By "working in synergistically," we mean that the purpose of integrating the set of elements is to leverage the capabilities of individual element capabilities to accomplish a higher level capability that cannot be achieved as stand-alone elements. i.e the whole is greater than the sum of its parts. The

cooperative action of system components is such that the total effect is greater than the sum of the effects taken independently. This is a major impetus for systems development. Subsystems defined and designed independently may not produce optimum benefits even if the individual system components are optimal

1.2 System Environment

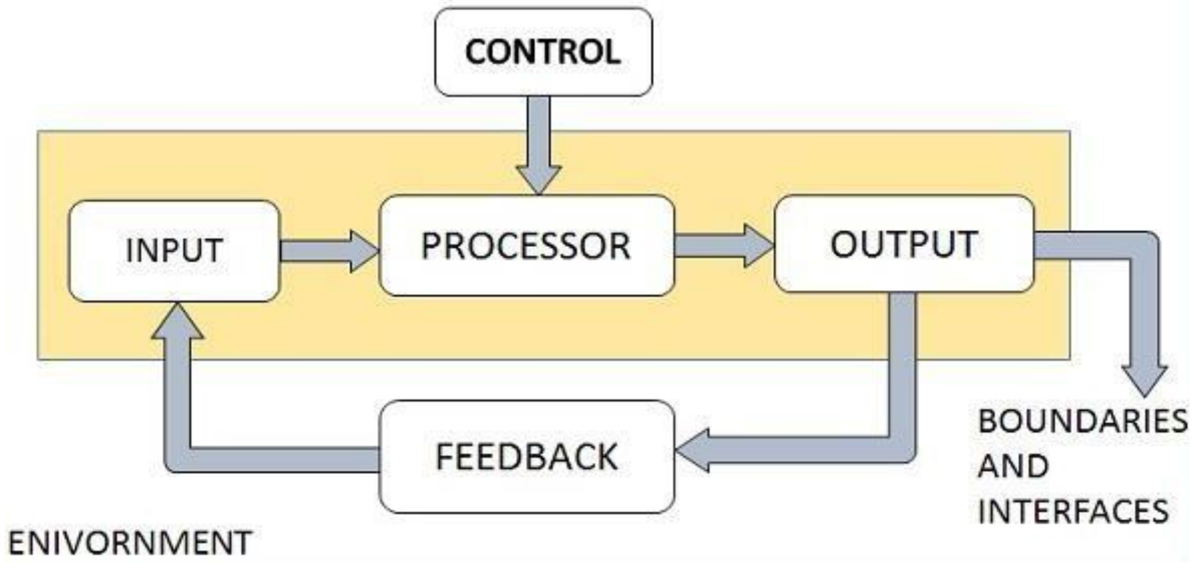
Organizations are open systems. They are not closed and self-contained and therefore the relationship between the organization and its environment is important. Organisations will exchange information with the environment, both influencing the environment and being influenced by it. The system, (the organization), will be affected by, for example, policies of the government, competitors, suppliers and customers, and unless these are taken into account, predictions regarding the organization will be incorrect;

Therefore, the system to be developed should be:

- compatible to any existing system and situations (implementation plans, system to support project management, procedures for project monitoring and evaluation and reporting),
- designed to be flexible to accommodate unpredictable decisions to resources cut, transfer etc. by supervising and funding agencies,
- designed taking into account the characteristics of the potential system users. These include mainly the technical capabilities of the potential users.

1.3 System Components

There are three major components in every system, namely input, processing and output, as illustrated below.



In a system the different components are connected with each other and they are interdependent. For example, human body represents a complete natural system. We are also bound by many national systems such as political system, economic system, educational system and so forth. The objective of the system demands that some output is produced as a result of processing the suitable inputs. A system also includes the 'control' that provides the output of the system.

A big system may be seen as a set of interacting smaller systems known as subsystems or functional units, each of which has its own defined tasks. All these work in coordination to achieve the overall objective of the system.

The basic elements of the system may be listed as:

a) Resources
b) Input (Data)/Output (Information)
c) Processes
d) Control/procedures
e) Feedback
f) Boundaries and interfaces

(a) Resources

Every system requires certain resources for the system to exist. Resources can be hardware, software or users. Hardware resources may include the computer, its peripherals, stationery etc. Software resources would include the programs running on these computers and the users would include the human beings required to operate the system and make it functional.

(b) Input (Data) / Output (Information)

Every system has some predefined goal. For achieving the goal the system requires certain inputs, which are converted into the required output. The main objective of the system is to produce some useful output. Output is the outcome of processing. Output can be of any nature e.g. goods, services or information.

(c) Processes

The systems have some processes (transformation) that make use of the resources to achieve the set goal under the defined procedures. These processes are the operational element of the system.

(d) Control/Procedures

Every system functions under a set of rules that govern the system to accomplish the defined goal of the system. This set of rules defines the procedures for the system to operate. For instance, the Banking systems have their predefined rules for providing interest at different rates for different types of accounts.

(e) Feed Back

Feedback is an important element of systems. The output of a system needs to be observed and feedback from the output taken so as to improve the system and make it achieve the laid standards.

(f) Boundaries and Interfaces

Every system has defined boundaries within which it operates. Beyond these limits the system has to interact with the other systems. For instance, Personnel system in an organization has its work domain with defined procedures. If the financial details of an employee are required, the system has to interact with the Accounting system to get the required details.

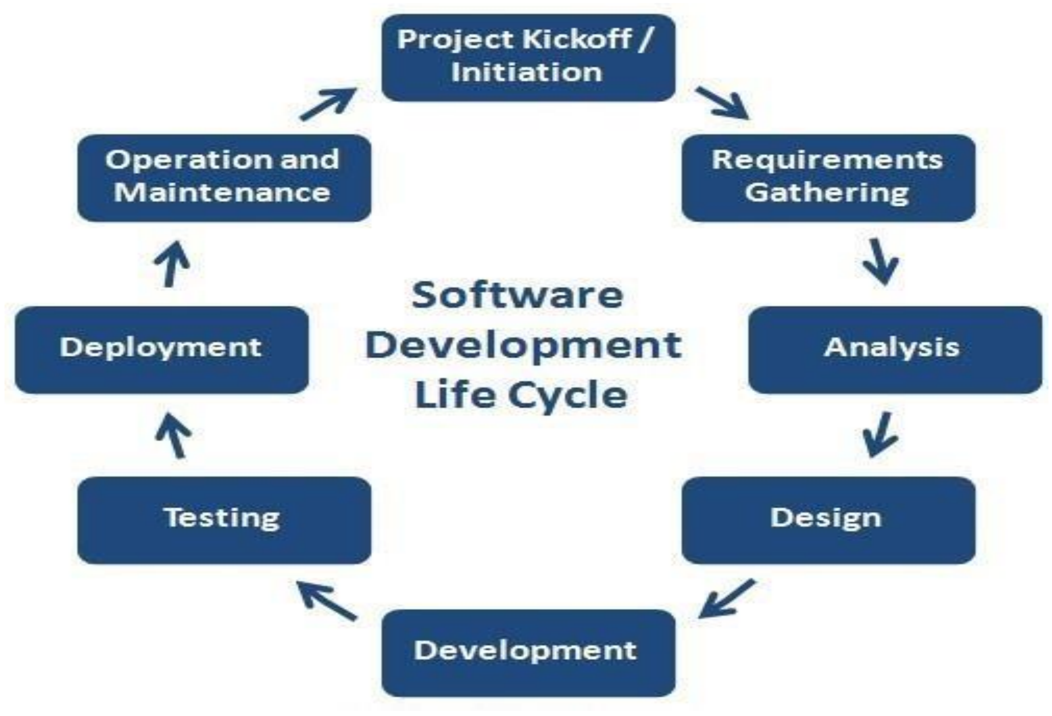
2.0 SYSTEMS DEVELOPMENT LIFECYCLE (SDLC)

Any (physical) product development can be expected to proceed as an organized process that usually includes the following phases:

- Problem
- Analysis
- Design
- Construction
- Implementation
- Production

So is with software development. The common software development phases are as follows:

1. Problem / Initiation: - Understand the problem
2. Requirements gathering:- Expand and amplify the statement of requirements
3. Analysis: --Getting the actual requirements/specification
4. Design: -- Coming up with logical solutions of the problem
5. Development: -- Encoding the design in a programming language and testing it
6. Implementation:-Putting the developed system into operation
7. Production and Maintenance: -- Running the system; fixing bugs and adding new features


Project Kickoff / Initiation
Requirements Gathering
Analysis
Design
Development
Testing
Deployment
Operation and Maintenance
Software Development Life Cycle

The early inspiration for systems lifecycle came from other engineering disciplines, where the above activities usually proceed in a sequential manner. This method is known as *waterfall process* because initially, developers used to build monolithic systems in one fell swoop. This method requires completing the pieces of the current phase before proceeding to the subsequent one.

However, over the years the developers noticed that systems development is unlike any other (physical) product development in the following aspects:

- Unlike most of other products, software is *intangible* and hard to visualize. Most people experience software through what it does: what inputs it takes and what it generates as outputs.
- Software is probably the most *complex* piece—a large software product consists of so many bits and pieces as well as their relationships, every single one having an important role—*one flipped bit can change the entire sense of a program.*
- Software is the most *flexible* piece—it can be easily and radically be modified at any stage of the development process.

3.0 SYSTEMS ANALYSIS

Introduction

Requirements engineering helps software engineer to better understand the problem they will work to solve. It involves activities that lead to an understanding of **what** the customer wants, how end-users will interact with the software, and what the business impact will be. Requirements engineering starts with the problem definition: *customer statement of requirements*. This is an informal description of what the customers think they need of a software system to do for them. The problem could be identified by management personnel, through market research, by ingenious observation, or some other means. Defining the requirements for the planned system includes both *fact-finding* about how the problem is solved in the current practice as well as *envisioning* how the planned system might work.

Requirements Definition

Given the customer statement of requirements, the first step in the software development process is the ***requirements analysis*** or ***systems analysis***. During this activity the developer attempts to expand and amplify the statement of requirements and produce the system specification document—the document that is an exact description of **what** a planned system will do. Requirements analysis state clearly the system and specifies the services it offers, identifies the types of users that will interact with the system, and identifies other systems that interact with it. The system is at first considered a black box, its services are identified, and typical interaction scenarios are detailed for each service.

This is simply a logical ordering of requirements engineering activities, regardless of the methodology that is used.

Requirements gathering (also known as "requirements elicitation") helps the customer to define what is required: what is to be accomplished, how the system will fit into the needs of the business, and how the system will be used on a day-to-day basis. This turns out to be very hard to achieve. The customer statement of requirements, is rarely clear and complete enough for the development team to start working on the software product.

Requirements analysis involves refining and modifying the requirements initially received from the customer during requirements gathering. Analysis is driven by the creation and elaboration of user scenarios that describe how the end-user will interact with the system. Negotiation with the customer will be needed to determine the priorities, what is essential, and when it is required. A popular tool is the use cases.

Requirements specification represents the problem in a semiformal or formal manner to ensure clarity, consistency, and completeness. It describes the function and performance of the planned software system and the constraints that will govern its development. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios (or, "use cases"), a prototype, or any combination of these. The developers could use UML or some other symbolic language for this purpose. It is important to ensure that the developers' understanding of the problem coincides with the customers' understanding of the problem.

Types of Requirements

Requirements specification is a precise and detailed descriptions of the system's functionality and constraints.
System requirements make clear the characteristics of the system that is to be developed. Requirements are usually divided into **functional** and **non-functional** requirements.

(a) Functional Requirements

- The objective of this process is to elicit, analyse, prioritise and validate the functional requirements that define the capabilities of the system.
- Functional requirements are specified by statements of functionality. It is highly recommended that Use Cases should be developed for all functional requirements, especially those that relate to user-system interaction and support a user-centric operation model.
- A complete set of functional requirements for the new system should be specified and stated in the form of a list. Requirements for the new system may include handling of existing functions, problem resolutions, provision of new facilities or functions etc.
- Functional requirements represent the functions or features that the system must have and the tasks (Use Cases) that the users must be able to perform.
- Functional requirements determine how the system is expected to behave and what kind of effects it should produce in the application domain.
- a specification is a document that clearly and accurately describes the essential technical requirements for; items, materials, services, procedures, etc, by which it can be determined that the requirements have been met
- A functional specification does not define the inner workings of the proposed system
- it does not include the specification of how the system function will be implemented
- it focuses on what various outside agents might "observe" when interacting with the system
- The functional specification is a kind of guideline
- It is also a continuing reference point as the developers write the programming code

Thus Functional requirements describe the services to be provided in detail, i.e. it is a structured document setting out detailed descriptions of the system services; what processing system must perform, details of inputs and outputs as well as details of data held by system amongst others.

Functional requirements can either be ***informal or formal***

Informal functional requirements

- can be considered as a blueprint or user manual from a developer's point of view
- usually written in less complex or technical language

Formal functional requirements

- have a definite meaning defined in mathematical or programmatic terms.
- may contain descriptions of user tasks, dependencies on other products, and usability criteria

- In practice, most successful requirements are written to understand and fine-tune applications that were already well-developed

The purpose for functional requirements is for

- Team projects achieve some form of team consensus on what the program is to achieve
- Done before making more time-consuming effort of writing source code and test cases
- This is followed by a period of debugging

Examples of Functional Requirements

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

(b) Non-functional requirements

Non-functional requirement is a constraint on the system or on the development process. Non-functional requirements define constraints on the system and the development process i.e. they describe system properties that do not relate to the system function. An example of a non-functional specification is: Maintain a persistent data backup, for the cases of power outages. Other non-functional system properties include the following amongst many others: Fault-tolerance, Usability, Reliability, Performance, Security, Usability, HCI issues, etc

NB: Although it may appear easy at first, the distinction between functional and non-functional requirements is often difficult to make. More often than not, these requirements are intertwined and satisfying a non-functional requirement usually necessitates modifications in the system function. One should be cautioned against regarding non-functional requirements *secondary* in importance relative to functional specification. The satisfaction of non-functional requirements must be as thoroughly and rigorously ensured as is the case for functional requirements.

Requirements Gathering Strategies

Requirements for a planned system should be devised based on **observing** the current practice and **interviewing** the stakeholders, such as end users, managers, etc. To put it simply, you can't fix it if you don't know what's broken. Structured interviews help in getting an understanding of what stakeholders do, how they might interact with the planned system, and the difficulties that they face with the existing technology. Agile methodologists recommend that the customers or users stay continuously involved throughout the project duration, instead of only providing the requirements initially and waiting for a completed system.

How to precisely specify what system needs to do is a problem, but sometimes it is even more difficult to get the customer to say what he or she expects from the system. Gathering domain knowledge by interviews is difficult because domain experts use terminology specific to their domain that is unfamiliar and hard for an outsider to grasp.

In addition, it is often difficult for the user to imagine the work *with* a yet-to-be-built system. *People can easily offer suggestions on how to improve the work practices in small ways, but very rarely can they think of great leaps, such as, to change their way of doing business on the Internet before it was around, or to change their way of writing from pen-and-paper when word processors were not around.*

Of great help in such situation is having a working instance, a prototype, or performing "funny" experiment with a mock-up system.

Writing Requirements Specifications

Write a requirements document that describes in precise detail what the customer wants. Find out what the client wants. This should include what the software should do and also:

- likely and possible enhancements;
- platforms (machines, OS, programming language, etc);
- cost; ,• delivery schedule;
- terms of warranty and maintenance;
- user training.

A report on software engineering practices once noted that requirements are hard. The big problems are actually managerial, rather than technical. Specific recommendations were:

- Use evolutionary [requirements] acquisition to reduce risk.
- Remove any dependence on the assumptions of the “waterfall” model.
- Provide the ability to do rapid prototyping in conjunction with users.

Note: The requirements document does **not** say how the software works. The major deliverable here is a System Requirements Document (SRD).

A well-written SRD will reduce development effort by avoiding (expensive) changes later, in design and implementation phases ("*Usipoziba Ufa, utajenga ukuta*").

Characteristics of a good SRD:

(a) The SRD should define all of the software requirements **but no more**. In particular, the SRD should not describe any design, verification, or project management details.

(b) The SRD must be **unambiguous**.

(c) There should be exactly one interpretation of each sentence.

(d) Avoid "variety" — good English style, but not good SRD style.

(e) The SRD must be **complete**. It must contain all of the significant requirements related to functionality (what the software does), performance (space/time requirements), design constraints ("must run in 640Kb"), and external interfaces. The SRD must define the response of the program to all inputs.

(f) The SRD must be **verifiable**. A requirement is *verifiable* if there is an effective procedure that allows the product to be checked against the SRD.

(g) The SRD must be **consistent**. A requirement must not conflict with another requirement.

(h) The SRD must be **modifiable**. It should be easy to revise requirements safely — without the danger of introducing inconsistency.

(i) The SRD must be **traceable**. i.e the origin of each requirement must be clear. (Implicitly, a requirement comes from the client; other sources should be noted.)

(j) Distinguish mandatory, desirable, and optional requirements.

NB: The SRD is prepared by both the supplier with help and feedback from the client.

- The client (probably) does not understand software development.
- The supplier (probably) does not understand the application.

4.0 STRUCTURED SYSTEMS DEVELOPMENT METHODS

Structured systems development methods (SSDM) follows the waterfall life cycle model starting from the feasibility study to the physical design stage of development. One of the main features of SSDM is the intensive user involvement in the requirements analysis stage. The users are made to sign off each stage as they are completed assuring that requirements are met. The users are provided with clear, easily understandable documentation consisting of various diagrammatic representations of the system. SSDM breaks up a development project into stages, modules, steps and tasks. The first and foremost model developed in SSDM is the data model. It is a part of requirements gathering and consists of well-defined stages, steps and products.

Some of the important characteristics of SSDM are:

- Dividing a project into small modules with well-defined objectives
- Useful during requirements specification and system design stage
- Diagrammatic representation and other useful modelling techniques
- Simple and easily understood by clients and developers
- Performing activities in a sequence

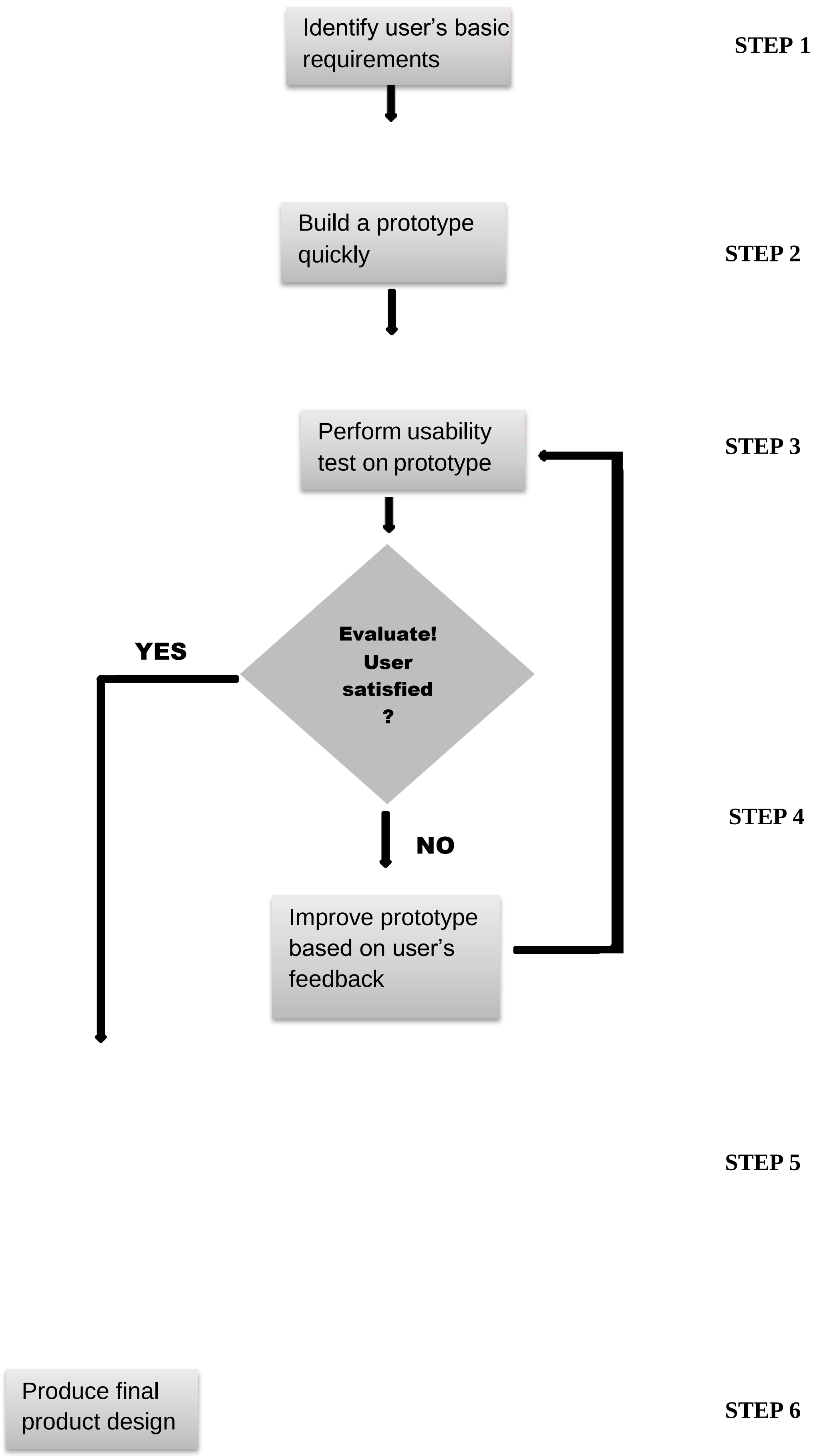
Software Process Models

A **process model** is a description of a way of developing software. A process model may also be a methodology for systems development.

1. Prototyping Model

The above insights led to adopting *incremental and iterative* (or, *prototyping)* development methods. Each iteration involves *progressing through* the design in more depth. Incremental and iterative process seeks to get to a working instance (**prototype**) as soon as possible. Having a working instance available lets the interested parties to have something tangible, to play with and make inquiries. Through this experimentation (by end users), unsuspected deficiencies are discovered that drive a new round of development using failures and the knowledge of things that would not work as a springboard to new approaches. This greatly facilitates the consensus reaching and building the understanding of all parties of what needs to be developed and what to expect upon the completion. The key of incremental and iterative methods is to progressively deepen the understanding or “visualization” of the target product, by both advancing and retracting to earlier activities to rediscover more of its features as indicated in the figure below.

Partitioning a problem into simpler ones, so called divide-and-conquer approach, is common when dealing with complex problems. In software development it is embodied in *modularity* (discussed in the next chapter): The source code for a module can be written and maintained independently of the source code for other modules. As with any activity, the value of a structured approach to software development becomes apparent only when complex tasks are tackled.


Identify user's basic requirements
STEP 1
Build a prototype quickly
STEP 2
Perform usability test on prototype
STEP 3
Evaluate! User satisfied ?
YES
NO
STEP 4
Improve prototype based on user's feedback
STEP 5
Produce final product design
STEP 6

1. Identify the users' basic requirements

2. Quickly build a working prototype

3. Perform a usability test on the prototype by giving it to the users to use it

4. Evaluate the prototype and check whether users are satisfied or not

5. If users are not satisfied, improve prototype based on users' feedback

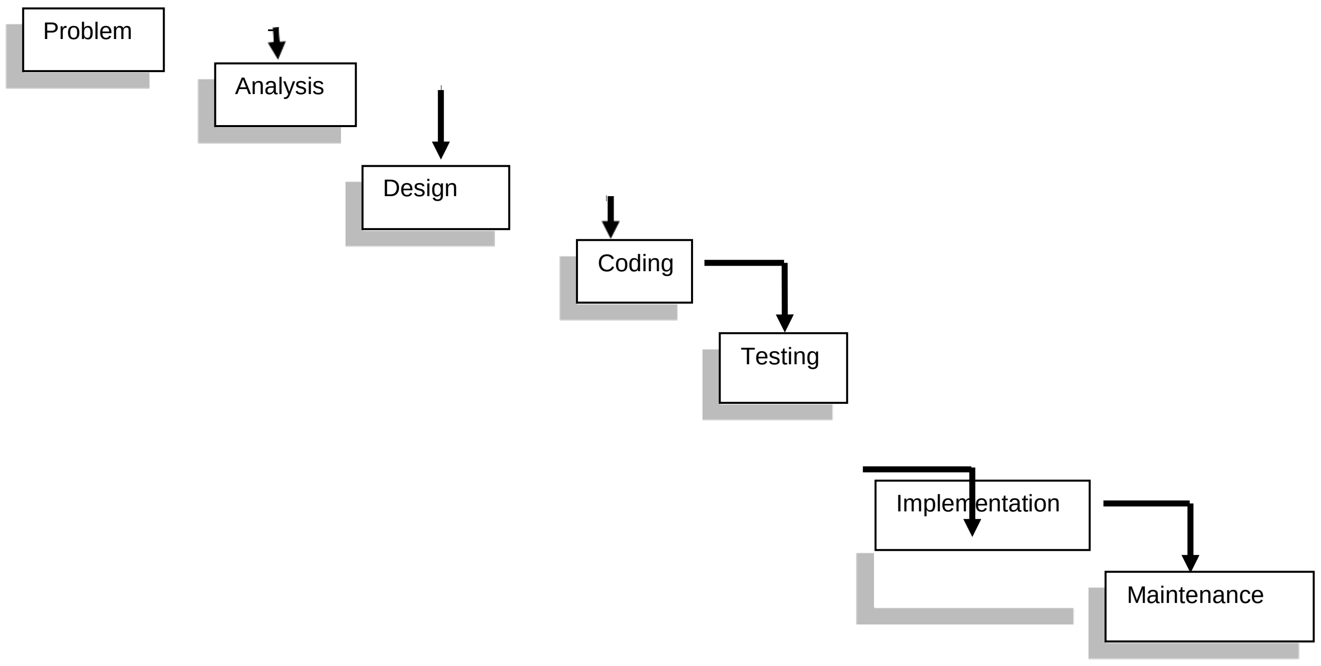
6. If users are satisfied, produce the final prototype

The objective of the Evolutionary development is to work with customers and to evolve a final system from an initial outline specification. It should start with well-understood requirements.
The objective of the Throw-away prototyping is to understand the system requirements. This should start with poorly understood requirements.

The objective of the Incremental developments is rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality. User requirements are prioritised and the highest priority requirements are included in early increments

2. The Waterfall Model

The Waterfall Model is an early lifecycle model. It is based on engineering practice; it works well if the requirements are well-understood and do not change. Ideally it is an old, document-driven model based on engineering practice but still used. A software project is divided into phases. There is feedback from each phase to the previous phase, but no further.



Waterfall model is **document driven** because requirements analysis yields a document that is given to the designers; design yields a document that is given to the implementers; and so on

Good features of the model include the following amongst others:

- simple to understand;
- phases are important even if their sequence is not;
- works for well-understood problems;
- keeps managers happy.

Bad features:

- does not allow for change;
- does not work for novel or poorly understood problems;

• does not allow for changing requirements;

• excessive number of documents lead to “bureaucratic” project management with more concern for the existence/size of documents than their meaning.

5.0 SYSTEM DESIGN

The design phase of the SDLC uses the requirements that were gathered during analysis to create a blueprint for the future system. A successful design builds on what was learned in earlier phases and leads to a smooth implementation by creating a clear, accurate plan of what needs to be done.

The purpose of the *analysis* phase is to figure out **what** the business needs. The purpose of the *design phase* is to decide **how** to build it. System design is the determination of the overall system architecture—consisting of a set of physical processing components, hardware, software, people, and the communication among them—that will satisfy the system's essential requirements.

Design is explaining the idea/concept of something, usually with *graphical diagrams*, with the intention to build from the explanation. It is a representation of a product or a system with sufficient detail for "construction". It is a meaningful development representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for 'good' design. In the systems development context, design focuses on four major areas of concern, data, architecture, interfaces, and components. It is the process of converting the system specification into an executable system

What Is Design?

Design is the first step in the development phase for any engineered product or system. Design can be defined as "*the process of applying various techniques and principles for the purpose of defining a system in sufficient detail to permit its physical construction*".

The goal of the designer is to produce a model or representation of an entity that will later be built.

Systems design is an iterative process through which requirements are translated into a blueprint/plan for constructing the system. Initially, the blueprint depicts a holistic view of system, i.e. the design is represented at a high level of abstraction. As the design iterations occur, subsequent refinements lead to design representations at much lower levels of abstraction.

The design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the ***thing*** to be built e.g. a three dimension plan of the house and slowly refines the ***thing*** to provide guidance for constructing each detail e.g. plumbing layout.

The design step produces a data design, an architectural design, an interface design etc.

(a) Data Design

The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity-relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity.

(b) Architectural design

The architectural design defines the relationship among major structural elements of the program. The design representation, the modular framework of the program, can be derived from the analysis model and the interaction of subsystems defined within the analysis model.

(c) Interface design

The interface design describes how the software communications within itself, to systems that interoperate with it and with humans who use it. An interface implies a flow of information e.g. data or control. Therefore the data or control flow diagrams provide the information required for interface design.

Systems Design and Quality

During design decisions are made that will ultimately affect the success of software construction and the ease with which software can be maintained. Importance of design can be stated with a single word; "**quality**".

Design is the place where quality is nurtured in software development. Design provides the representations of software that can be assessed for quality. Design is the only way one can accurately translate the customers' requirements into a finished software product or system. Software design serves as the foundation for all software engineering steps that follow.

Without design, one risks building unstable system. A system that will fail when small changes are made, one that may be difficult to test, one whose quality cannot be assessed until late in the software engineering process, when time is short and a lot of time and money have already been spent.

MODELING

Logical Data Modeling: The process of identifying, modeling and documenting the data requirements of the system being designed. The data are separated into entities (things about which a business needs to record information) and relationships (the association between the entities).

This involves the process of identifying, modelling and documenting data as a part of system requirements gathering. The data are classified further into entities and relationships.

Data Flow Modeling: The process of identifying, modeling and documenting how data moves around an information system. Data Flow Modeling examines processes (activities that transform data from one form to another), data stores (the holding areas for data), external entities (what sends data into a system or receives data from a system), and data flows (routes by which data can flow).

Process Modeling

Modeling clarifies the requirements definition from the analysis phase by use of graphical representation or models. (*A picture is worth/speaks a 1,000 words*).

A *process model* is a graphical way of representing how a business system should operate. It illustrates the processes or activities that are performed and how data move among them. A process model can be used to document the current system or the new system being developed whether computerized or not. There are many different process modeling techniques in use today such as DFDS, ERD, Gantt charts, PERT, etc. One of the most commonly used technique is Data flow diagramming which is a technique that diagrams the business processes and the data that pass among them.

Although the name *data flow diagram* (DFD) implies a focus on data, this is not the case. The focus is mainly on the processes or activities that are performed. Data modeling, presents how the data created and used by processes are organized. Process modeling—and creating DFDs in particular—is one of the most important skills needed by systems analysts.

The focus on *logical process models*, which are models that describe processes, without suggesting how they are conducted. When reading a logical process model, one is not able to tell whether a process is computerized or manual, whether a piece of information is collected by paper form or on line, or whether information is placed in a filing cabinet or a large database.

These physical details are defined during the design phase when these logical models are refined into *physical models*, which provide information that is needed to ultimately build the system. By focusing on logical processes first, analysts can focus on how the business should run, without being distracted by implementation details.

Data Flow Diagram (DFD)

The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions. It is a diagrammatic representation of the information (data) flow within a system. That is DFDs track any information entering or leaving the system and also represents how and in what form the information is getting stored.

Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system. A DFD model uses a very limited number of primitive symbols to represent the functions performed by a system and the data flow among these functions.

NB:

- DFDs show the way that a system processes data. They illustrate how the data flows from input through various processes to the output, showing every step the system takes to process the data.

- DFDs are a common way of modeling data flow for software development. For example, a DFD for a word-processing program might show the way the software processes the data that the user enters by pressing keys on the keyboard to produce the letters on the screen.

- DFDs are popular for software design because the diagram makes the flow of data easy to understand and analyze. DFDs represent the functions a system performs hierarchically, starting with the highest-level functions and moving through various layers or levels of sub-functions. As a modeling technique, DFDs are useful for performing a structured analysis of software problems, allowing developers to spot and pinpoint issues in software development.

- Data flow diagram is a graphical tool used to analyze the movement of data through a system. DFDs view a system as a function that transforms the inputs into desired outputs. Any complex system will not perform this transformation in a "single step", and data will typically undergo a series of transformations before it becomes the output. The DFD aims to capture the transformation that takes place within a system to the input data so that eventually the output data is produced. The agent that performs the transformation of data from one state to another is called a process (or a bubble). So a

DFD shows the movement of data through the different transformation or process in the system.

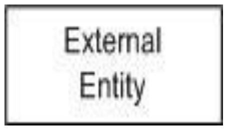
Notations in DFD’s

Data Flow Diagrams are composed of the four basic symbols shown below.

- The External Entity symbol represents sources of data to the system or destinations of data from the system.
- The Data Flow symbol represents movement of data.
- The Data Store symbol represents data that is not moving (delayed data at rest).
- The Process symbol represents an activity that transforms or manipulates the data (combines, reorders, converts, etc.).

Any system can be represented at any level of detail by these four symbols.

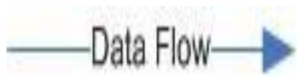
External Entities



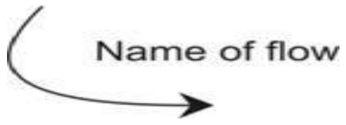
External entities determine the system boundary. They are external to the system being studied. They are often beyond the area of influence of the developer.

These can represent another system or subsystem. These go on margins/edges of data flow diagram. External entities are named with appropriate name.

Data Flow



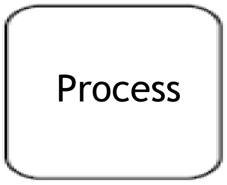
Or



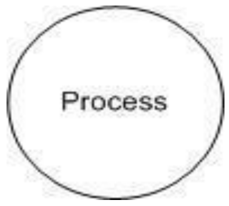
Data flow represents the input (or output) of data to (or from) a process ("data in motion"). Data flows only data, not control. Data flows must begin and/or end at a process.

Data flows are always named. Name is not to include the word "data". Should be given unique names. Names should be some identifying noun. For example, order, payment, complaint.

Processes



OR



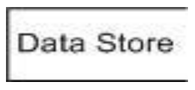
Processes are work or actions performed on incoming data flows to produce outgoing data flows. These show data transformation or change. Data coming into a process must be "worked on" or transformed in some way. Thus, all processes must have inputs and outputs. In some (rare) cases, data inputs or outputs will only be shown at more detailed levels of the diagrams. Each process in always "running" and ready to accept data.

Major functions of processes are computations and making decisions. Each process may have dramatically different timing: yearly, weekly, daily.

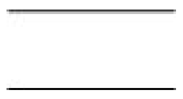
Naming Processes

Processes are named with one carefully chosen verb and an object of the verb. There is no subject. Name is not to include the word "process". Each process should represent one function or action. If there is an "and" in the name, you likely have more than one function (and process). For example, get invoice, update customer, create order etc. Processes are numbered within the diagram as convenient. Levels of detail are shown by decimal notation. For example, top level process would be Process 14, next level of detail Processes 14.1-14.4, and next level with Processes 14.3.1-14.3.6. Processes should generally move from top to bottom and left to right.

Data Stores



OR



Data Stores are repository for data that are temporarily or permanently recorded within the system. It is repository of data, that is, a file or database. It is an "inventory" of data. These are common link between data and process models. Only processes may connect with data stores.

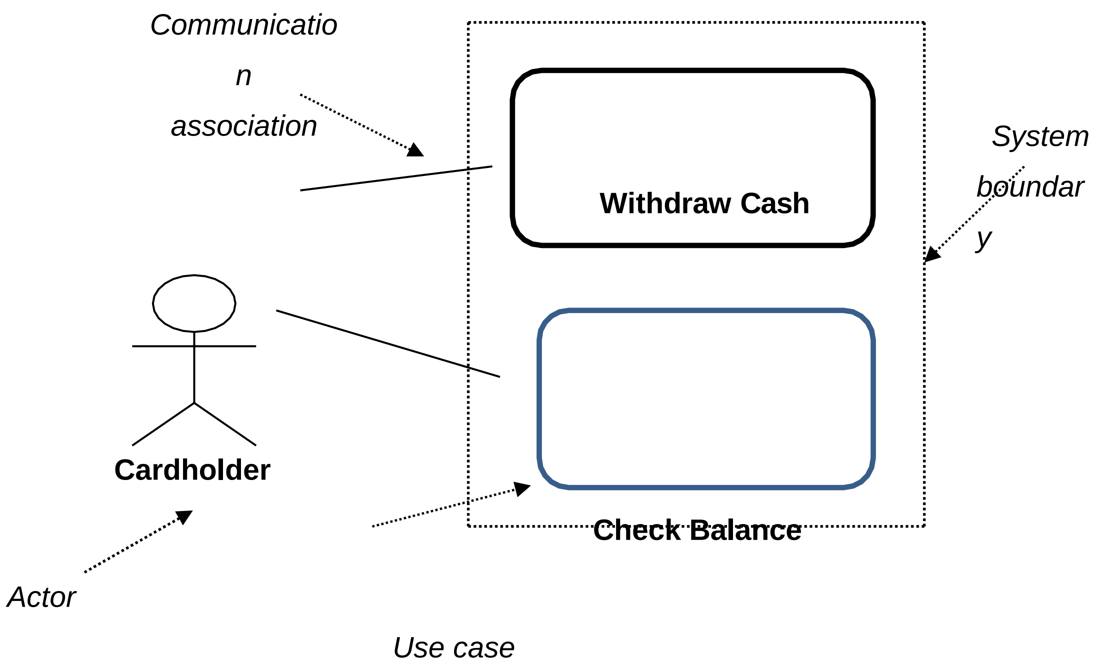
There can be two or more systems that share a data store. This can occur in the case of one system updating the data store, while the other system only accesses the data.

Data stores are named with an appropriate name, not to include the word "file", Names should consist of plural nouns describing the collection of data. Like customers, orders, and products.

USE CASE DIAGRAM

ATM Case

Use cases are used to model functionality; i.e. <u>what</u> system does, and not <u>how</u> it does. Focus is on functionality from users' perspective and thus it is not appropriate for non-functional requirements.



This is a use case diagram example for an automated teller machine (ATM). The system provides customer, bank and technicians with access to core functions like check balance, deposit funds, withdraw cash, transfer funds, maintenance and repair.

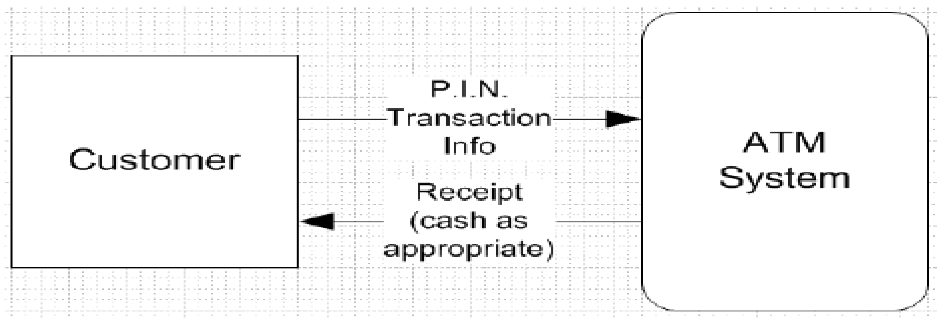
Various levels of Data Flow Diagrams

The highest level DFD is a **“context diagram,”** and consists of one and only one process, and absolutely zero data stores. It gives a very high level view of the system. A context diagram is followed by a **level 0 diagram**, then a **level 1 diagram**, and so forth. The lowest level DFD is known as a “primitive” data flow diagram.

Example“An ATM is an electronic device designed for automated dispensing of money. A User can withdraw money quickly and easily after authorization. The user interacts with the system through a card reader and a numerical keypad. A small display screen allows messages and information to be displayed to the user. Bank members can access special functions such as ordering a statement”

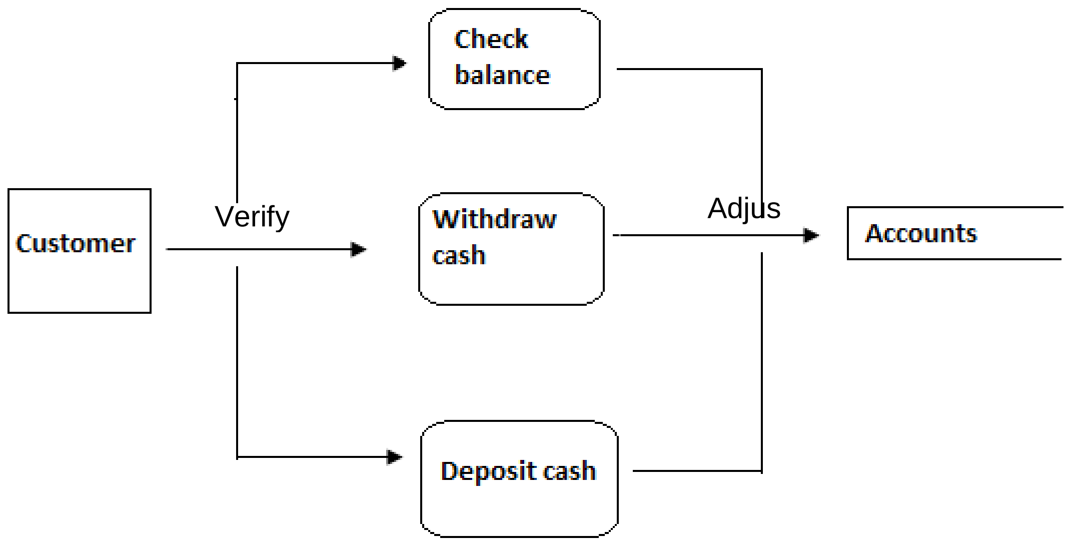
U s e

Consider the context diagram for an ATM System below



Note:

- ✓ The diagram shows both **Input(s)** and **Output(s)**
- ✓ Only 1 process is shown on a Context Diagram –this process represents the System in its entirety
- ✓ A Data Store is ***never*** shown on a Context Diagram
- ✓ The inputs to a process are different from the outputs (processes transform inputs into outputs
- ✓ All Communication must flow through a process
- ✓ An entity cannot communicate directly with another entity (since an entity is **external** to the system)
- ✓ An entity cannot access a data store without going through a process (since a data store is part of the system, and an entity is external to the system).
- ✓ Data cannot move directly from one data store to another; data movement must occur through a process. Data flows in only one direction at a time (data flow arrows are uni-directional)
- ✓ All entities, processes, data flows, and data stores are labeled
- ✓ There may be as many entities, processes, data flows, and data stores as needed.



The Level-0 diagram is followed by a Level-1 diagram

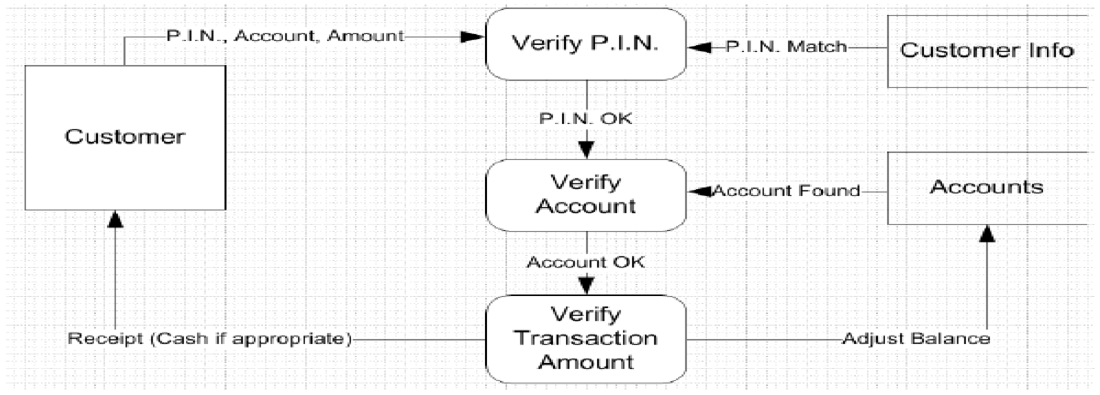
- ✓ One Level-2 diagram is drawn for each *process* appearing on the Level-1 diagram
- ✓ Inputs and outputs from the process should remain balanced with what is depicted on higher level diagrams
- ✓ When the process is fully understood, diagramming stops (this rule applies to the system as a whole, as well)
- ✓ The final set of data flow diagrams is referred to as primitive data flow diagrams
- ✓ All processes must have input; if a process is making outputs without inputs, there something terribly wrong.
- ✓ All processes must produce outputs; if a process is taking inputs but not producing outputs, there is a "black hole" (if an object only ever receives inputs without producing outputs, perhaps it should be modeled as a sink (entity) rather than as a process).
- ✓ Processes typically have a verb (action) name, while entities and data stores typically have a noun name
- ✓ A data flow leading into a data store is an update (i.e., write, modify, delete)
- ✓ A data flow leading out of a data store is a use (i.e., read, retrieve)
- ✓ Data may not flow directly from one data store to another data store. All movement of data must occur through a process
- ✓ Data may not move from a source into a data store. Data must be sent from the source into a process

and from the process into the data store

- ✓ Data may not move from a data store into a sink. Data must be sent from the data store into a process and from the process into the sink
- ✓ Data may not move directly between a source and a sink. All data must flow through the system. If data does not flow through a process (i.e., through the system), it is external to the system, and therefore does not appear on a data flow diagram.
- ✓ Note also that the inputs to a process must be sufficient to produce the outputs.
- ✓ Several iterations of drawings will likely occur (software drawing tool use becomes a must)
- ✓ Consider stopping drawing when each process captures a single decision, and when each data store represents a single data entity

Consider the problem of obtaining money from an ATM cash machine. Suppose you need to determine that the user of the ATM is a valid user, and that the user of the ATM has enough money in their account to complete the transaction.

The data flow diagram (level 2) below illustrates the logical flow of data and the processes necessary for the transaction



Importance of DFDs in a good software design

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism – it is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub-functions. In fact, any hierarchical model is simple to understand. Human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem, but also for several other applications such as showing the flow of documents or items in an organization.

Data Modeling

Data modelling mainly discuss **how** the data that flow through the processes are organized and presented.

A *data model* is a formal way of representing the data that are used and created by a system; it illustrates people, places, or things about which information is captured and how they are related to each other (entity-relationships E-RDs).

During analysis, analysts draw a logical data model, which shows the logical organization of data without indicating how data are stored, created, or manipulated. Because this model is free of any implementation or technical details, the analysts can focus more easily on matching the diagram to the real requirements of the system without being distracted by technical details.

Later, during the design phase, the data model is changed to reflect exactly how the data will be stored in databases and files, i.e. the analysts draw a *physical data model* to reflect how the data will physically be stored in databases and files. At this point, the analysts investigate ways to store the data efficiently and to make the data easy to retrieve.

There are several ways to model data, but one commonly used techniques is the entity relationship diagramming (E-RD), a graphic drawing technique that shows all the data components of a system.

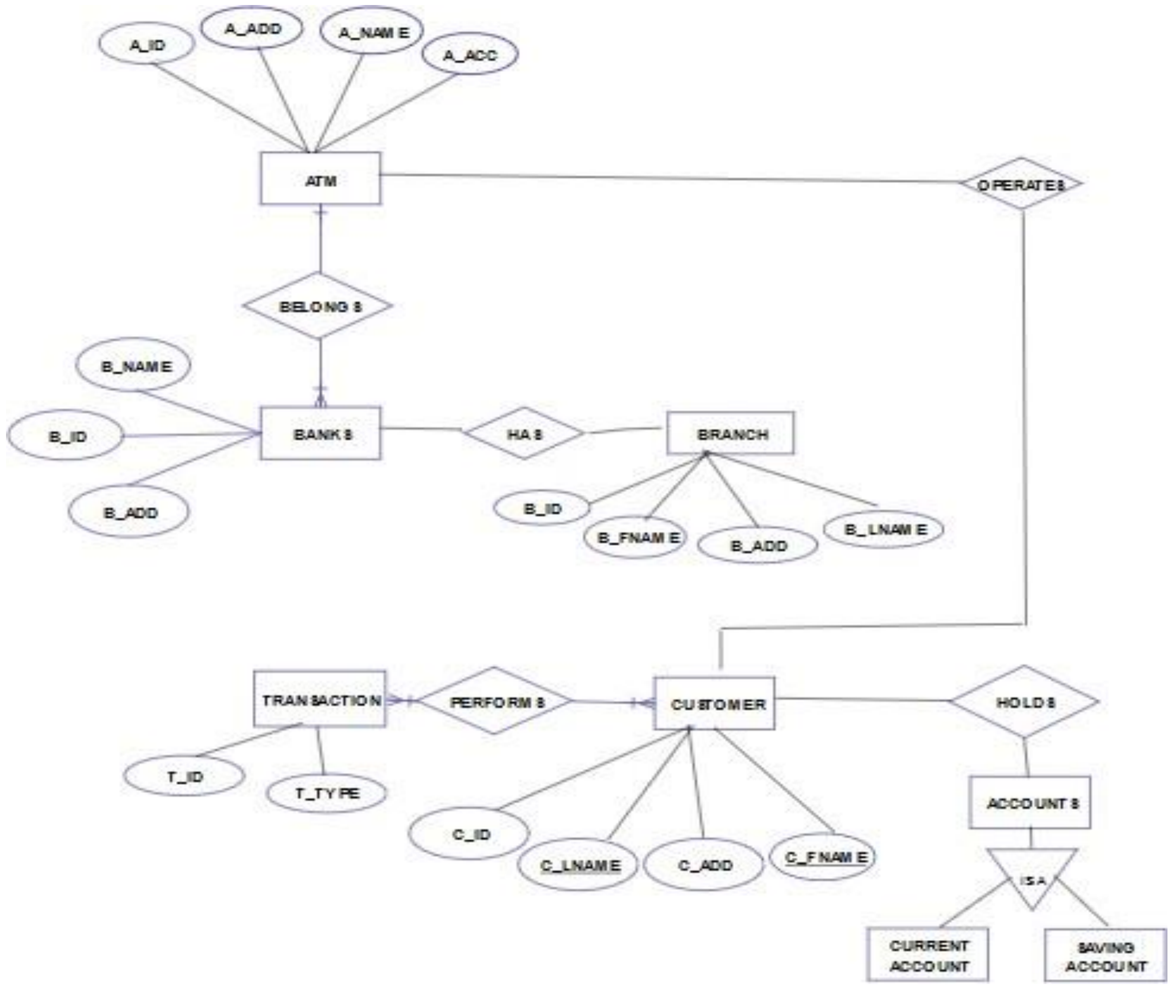
Entity Relationship Diagram (ERD)

An Entity Relationship (ER) Diagram is a type of flowchart that illustrates how “entities” such as people, objects or concepts relate to each other within a system. ER Diagrams are most often used to design or debug relational databases in the fields of software engineering, information systems development, etc. Also known as ERDs or ER Models, they use a defined set of symbols such as rectangles, diamonds, ovals and connecting lines to depict the interconnectedness of entities, relationships and their attributes. They mirror grammatical structure, with entities as nouns and relationships as verbs.

An entity relationship diagram (*ERD*) is a picture which shows the information that is created, stored, and used by a system. An analyst can read an ERD to discover the individual pieces of information in a system and how they are organized and related to each other. On an ERD, similar kinds of information are listed together and placed inside boxes called entities. Lines are drawn between entities to represent relationships among the data, and special symbols are added to the diagram to communicate high-level business rules that need to be supported by the system. The ERD implies no order, although entities that are related to each other are usually placed close together.

E-RDs aid in requirements gathering by determining the requirements of an information system by drawing a conceptual ERD that depicts the high-level objects of the system. Such an initial model can also be evolved into physical database model that aids the creation of relational database, or aids in the creation of process map and data flow model.

Consider the ATM system. A possible E-RD resembles the following


A_ID
A_ADD
A_NAME
A_ACC
ATM
OPERATES
BELONGS
B_NAME
B_ID
BANKS
B_ADD
HAS
BRANCH
B_ID
B_FNAME
B_ADD
B_LNAME
TRANSACTION
PERFORMS
CUSTOMER
HOLDS
T_ID
T_TYPE
C_ID
C_LNAME
C_ADD
C_FNAME
ACCOUNTS
ISA
CURRENT ACCOUNT
SAVING ACCOUNT

The components and features of an ER diagram

ER Diagrams are composed of entities, relationships and attributes. They also depict cardinality, which defines relationships in terms of numbers, such as one-to-one, one-to-many, and many-many.

Entity

It specifies distinct real world items in an application. A definable thing—such as a person, object, concept or event—that can have data stored about it. Think of entities as nouns. Examples: a customer, student, course, car, vendor, bank, product, account etc. Typically shown as a **rectangle**.

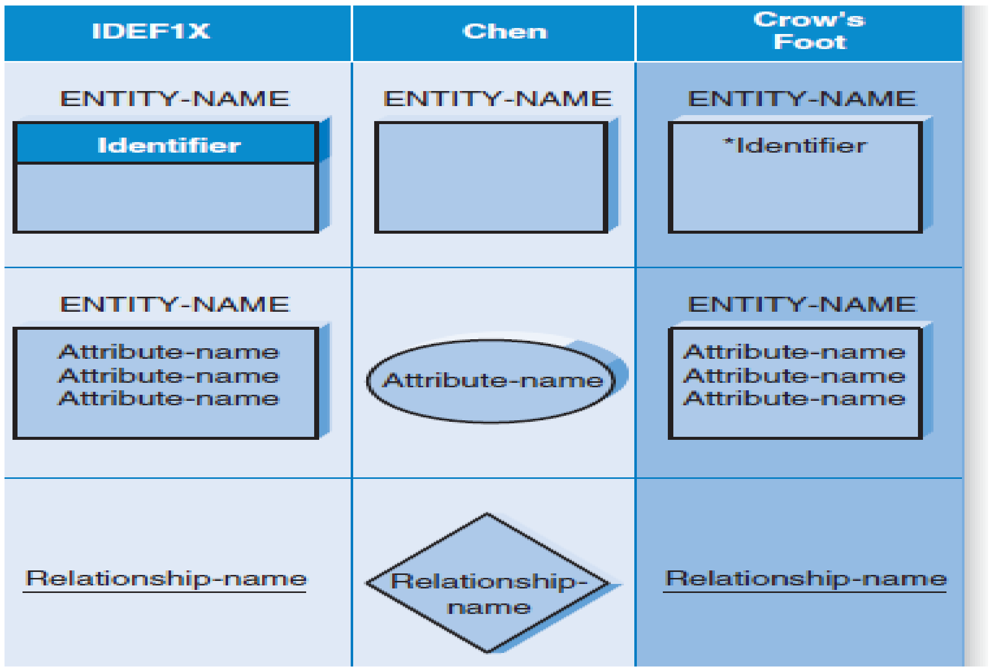
Relationship

How entities act upon each other or are associated with each other. Think of relationships as verbs. For example, the named student might register for a course. The two entities would be the student and the course, and the relationship depicted is the act of enrolling, connecting the two entities in that way. Relationships are typically shown as **diamonds** or **labels** directly on the connecting lines.

Attribute

A property or characteristic of an entity. Often shown as an oval or circle.

There are many different sets of symbols that can be used on an ERD. No one set of symbols dominates industry use, and none is necessarily better than another. (*Chen's format widely used though*)


IDEF1X
Chen
Crow's Foot
ENTITY-NAME
Identifier
ENTITY-NAME
ENTITY-NAME
*Identifier
ENTITY-NAME
Attribute-name
Attribute-name
Attribute-name
Attribute-name
ENTITY-NAME
Attribute-name
Attribute-name
Attribute-name
Relationship-name
Relationship-name
Relationship-name

6.0 CODING PHASE

Once the design is complete, most of the major decisions about the system have been made. The goal of the coding phase is to translate the design of the system into code in a given programming language. For a given design, the aim of this phase is to implement the design in the best possible manner. The coding phase affects both testing and maintenance profoundly. A well written code reduces the testing and maintenance effort. Since the testing and maintenance cost of software are much higher than the coding cost, the goal of coding should be to reduce the testing and maintenance effort. Hence, during coding the focus should be on developing programs that are easy to write. Simplicity and clarity should be strived for.

An important concept that helps the understandability of programs is structured programming. The goal of structured programming is to arrange the control flow in the program. That is, program text should be organized as a sequence of statements, and during execution, the statements are executed in the sequence in the program.

For structured programming, a few single-entry-single-exit constructs should be used. These constructs includes selection (if-then-else), and iteration (while - do, repeat - until etc). With these constructs it is possible to construct a program as sequence of single - entry - single - exit constructs. There are many methods available for verifying the code. Some methods are static in nature that is, that is they do not involve execution of the code. Examples of such methods are data flow analysis, code reading, code reviews, testing (a method that involves executing the code, which is used very heavily). In the coding phase, the entire system is not tested together. Rather, the different modules are tested separately (unit testing). Consequently, this phase is often referred to as "coding and unit testing". The output of this phase is the verified and unit tested code of the different modules.

Advancements in programming techniques which include the concept of "object-oriented programming" centers on the development of reusable program routines (modules) and the classification of data types (numbers, letters, dollars, etc.) and data structures (records, files, tables, etc.). Linking pre-scripted module objects to predefined data-class objects reduces development times and makes programs easier to modify.

Organizations should complete testing plans during the coding phase. Additionally, they should update conversion, implementation, and training plans and user, operator, and maintenance manuals.

7.0 SYSTEM TESTING

Testing is a process to detect errors in the software product. Before going into the details of testing techniques one should know what errors are. In day-to-day life we say whenever something goes wrong, there is an error. This definition is quite vast. When we apply this concept to software products then we say whenever there is difference between what is **expected** out of software and what is being **achieved**, there is an error.

For the output of the system, if it differs from what was required, it is due to an error. This output can be some numeric or alphabetic value, some formatted report, or some specific behavior from the system. In case of an error there may be change in the format of output, some unexpected behavior from system, or some value different from the expected is obtained. These errors can be due to wrong analysis, wrong design, or some fault on developer's part.

All these errors need to be discovered before the system is implemented at the customer's site.

Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V).

- Verification refers to the set of activities that ensure that software correctly implements a specific function.
- Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.

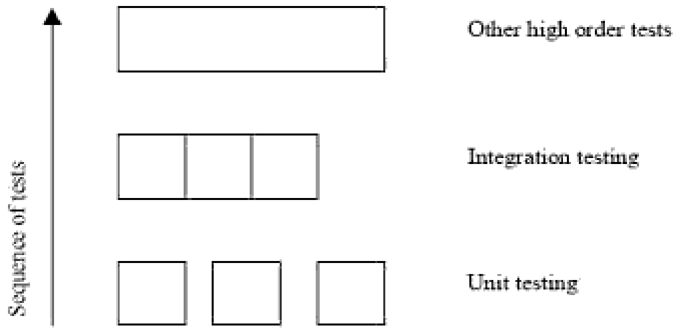
Boehm States like this

Verification: "Are we building the product **right**?"

Validation: "Are we building the **right** product?"

Types of Testing

Testing Strategies decide in which manner testing should be performed. As shown in fig. below unit testing is performed first. Unit testing focuses on the individual modules of the product. After that integration testing is performed. When modules are integrated into bigger program structure then new errors arise often. Integration testing uncovers those errors. After integration testing, other high order tests like system tests are performed. These tests focus on the overall system. Here system is treated as one entity and tested as a whole.



Unit Testing

It is known that the smallest unit of software design is a module. Unit testing is performed to check the functionality of these units. It is done before these modules are integrated together to build the overall system. Since the modules are small in size, individual programmers can do unit testing on their respective modules. So unit testing is basically white box oriented. Procedural design descriptions are used and control paths are tested to uncover errors within individual modules. Unit testing can be done for more than one module at a time.

Integration Testing

Unit testing ensures that all modules have been tested and each of them works properly individually. Unit testing does not guarantee if these modules will work fine if these are integrated together as a whole system. It is observed that many errors crop up when the modules are joined together. Integration testing uncovers errors that arise when modules are integrated to build the overall system.

Integration testing is a systematic technique for constructing the program structure while conducting tests to uncover errors associated with interfacing. The objective is to take unit tested modules, integrate them, find errors, remove them and build the overall program structure as specified by design.

System Testing

Software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements and a series of system integration and validation tests are conducted. These tests fall outside the scope of software engineering process and are not conducted solely by the software developer.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that all system elements have been properly integrated and perform allocated functions.

Recovery Testing

Many computer-based systems must recover from faults and resume operation within a pre-specified time. In some cases, a system may be fault tolerant; i.e, processing faults must not cause overall system failure. In other cases, a system failure must be corrected within a specified period or severe economic damage occurs.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If the recovery is automated (performed by system itself), re-initialization mechanisms, data recovery, and restart are each evaluated for correctness. If the recovery requires human intervention, the mean time to repair is evaluated to determine whether it is within acceptable limits.

Stress Testing

Stress tests are designed to confront program functions with abnormal situations. Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate 10 interrupts per seconds, when one or two is the average rate; (2) input data rates may be increased by an order of magnitude to determine how input functions will respond; (3) test cases that require maximum memory or other resources may be executed; (4) test cases that may cause excessive hunting for disk resident data may be created; or (5) test cases that may cause thrashing in a virtual operating system may be designed. The testers attempt to break the program.

Security Testing

Any computer-based system that manages sensitive information or causes actions that can harm or benefit individuals is a target for improper or illegal penetration.

Security testing attempts to verify that protection mechanism built into a system will protect it from unauthorized penetration. During security testing, the tester plays the role of the individual who desires to penetrate the system. The tester may attack the system with custom software designed to break down any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to find the key to system entry; and so on.

Alpha and Beta testing

For a software developer, it is difficult to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combination of data may be regularly used; and the output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Acceptance test is conducted by customer rather than by developer.

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

Customer conducts the alpha testing at the developer's site. The software is used in a natural setting with the developer. The developer records errors and usage problem. Alpha tests are conducted in a controlled environment.

The beta test is conducted at one or more customer sites by the end user(s) of the software. Here, developer is not present. Therefore, the beta test is a live application of the software in an environment that cannot be controlled by the developer. The customer records all problems that are encountered during beta testing and reports these to the developer at regular intervals. Because of problems reported during beta test, the software developer makes modifications and then prepares for release of the software product to the entire customer base.

Regression Testing

This is an important type of test that is performed when some corrections or changes are made to an existing system. Changes are fundamental and inevitable in software. Any software must undergo changes. Usually, a change is made either to correct detected errors or to add new features and functionality. It has been found that whenever such corrections or changes are made, somehow they result in the propagation of unexpected and obviously undesirable side effects. Therefore the purpose of regression testing is to ensure that changes made to the software have not altered the behaviors or functionality. It has been found that whenever such corrections or changes are made, somehow they result in the propagation of unexpected and obviously undesirable side effects. Therefore the purpose of regression testing is the need for adequate and accurate documentation at every stage.

White Box and Black Box Testing

White box testing focuses on the internal functioning of the product. For this, different procedures are tested. White box testing tests the following

- Loops of the procedure
- Decision points
- Execution paths

Black box testing tests the overall functional requirements of product. Inputs are supplied to product and outputs are verified. If the outputs obtained are same as the expected ones then the product meets the functional requirements. In this approach internal procedures are not considered. It is conducted at later stages of testing.

Black box testing uncovers the following types of errors.

1. Incorrect or missing functions
2. Interface errors
3. External database access
4. Performance errors
5. Initialization and termination errors.

8.0 IMPLEMENTATION

The implementation stage of any project is a true display of the defining moments that make a project a success or a failure. The implementation stage is defined as "the system or system modifications being installed and made operational in a production environment. The phase is initiated after the system has been tested and accepted by the user. This phase continues until the system is operating in production in accordance with the defined user requirements.

This stage "rolls out" the solution to the people who will be using it. It is necessary to:

- **train users** - who needs training on what topics? How, where and when will it be done?
- **install the system** and bring it online, using direct, phased, parallel and/or pilot changeover schemes.
- **use the solution** for real, and monitor its performance

Changing over from the old to the new system can be anything from trivial to traumatic, depending on the size of the visible differences between the two systems.

There are four main styles of changing from system to system :(the first 2 are complete strategies; the final 2 are optional accessories. Remember - if you choose pilot and/or parallel, you still need to use direct or phased implementation to finish the job!)

Different Implementation Strategies

Direct conversion:

The old system is stopped and the new system replaces it immediately. It is a quick transition, which may be unnerving if the changes are large. If used, it should not be done in a peak period where transition "hiccups" will upset the organisation.

Phased conversion:

If the new system has several components, they can be introduced one at a time. For example, an organisation introducing modern communications may first introduce the internet, then email, then videoconferencing, then electronic ordering. This takes more time, but each small step is less traumatic than one huge one, and staff can become accustomed to one change before facing the next. Also, problems with any step can be detected and fixed before the next step is taken. Phased conversion is impossible if the system is "one piece", for example when introducing new hard disks you can't add a little piece of a hard disk at a time: it has to be "all or nothing."

Pilot scheme:

This is not an alternative to the first 2 methods: it is an optional extra. If the organisation has several branches or departments where the new system will be implemented, it may decide to trial the new system in one location first and see how it goes. Any faults and problems will be limited to that one location and will not cripple the whole organisation. Pilot conversion may be impossible if the introduction of the new system in the test location makes it incompatible with other unchanged locations. (e.g. if every branch used MS Word and the pilot branch converted to Brand X Word Processor, it might not be able to share documents with the other branches or head office.)

Parallel operation:

This is not an alternative to the first 2 methods: it is an optional extra. If it is possible to keep the old system in place and running while the new system is installed, you gain certain benefits.

- You can directly compare the effectiveness and efficiency of the new and old systems
- If the new system fails, the old system is still there chugging away as normal so no harm is done.

Parallel conversion is only possible if the old and new systems are completely independent. It is impossible to achieve if the new system is built on the old system.

9.0 SYSTEM MAINTENANCE

It is impossible to produce systems of any size, which do not need to be changed. Over the lifetime of a system, its original requirements will be modified to reflect changing user and customer needs. The system's environment will change as new hardware is introduced. Error, undiscovered during system validation, may merge and require repair.

The process of changing of a system after it has been delivered and is in use is called Software maintenance. The changes may involve simple changes to correct coding errors, more extensive changes to correct design errors or significant enhancement to correct specification errors or accommodate new requirements. Maintenance therefore, in this context, really means evolution. It is the process of changing a system to maintain its ability to survive.

The purpose is to preserve the value of software over time. The value can be enhanced by meeting additional requirements, becoming easier to use, more efficient and employing newer technology. *Maintenance may span for 500 years, whereas development may be 1-2 years.*

There are four major types of software maintenance with very blurred distinction between them.

1. Corrective maintenance
2. Adaptive maintenance
3. Perfective maintenance
4. Preventive maintenance

Corrective Maintenance

Corrective Maintenance is concerned with fixing reported errors in the software. Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve the rewriting of several programs components. Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.

Adaptive maintenance

Adaptive maintenance means changing the software to new environment such as different hardware platform or for use with different operating systems. The software functionality does not radically change.

Perfective maintenance

Perfective maintenance involves implementing new functional or non-functional system requirements. These are generated by software customers as their organizations and business changes.

Preventive Maintenance

There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software, which reflects deteriorating structure. The work is required to be done to maintain it or to reduce it, if possible.

This work may be named as preventive maintenance. This activity is usually initiated from within the maintenance organization with the intention of making program easier to understand and hence facilitating future maintenance work. This includes code restructuring, code optimization and documentation updating. After a series of quick fixes to software, the complexity of its source code can increase to an unmanageable level, thus justifying complete restructuring of the code. Code optimization can be performed to enable the programs to run faster or to make more efficient use of storage.