Mount Kenya University

DEPARTMENT OF PURE AND APPLIED SCIENCES

BBIT 3101

UNIT: SOFTWARE ENGINEERING

CHAPTER 1: FUNDAMENTALS OF SOFTWARE ENGINEERING

Chapter Objectives

At the end of this chapter, student should understand:

1. The Evolution of Software and the Software Crisis
2. Concepts of Software Engineering
3. Skills necessary for Software Engineering
4. Software Engineering Components:
   - Methods,
   - Tools and Procedures
5. Software Engineering Paradigms
   - The Classical Life Cycle
   - Prototyping
   - Fourth Generation Techniques
   - Combination of Paradigms
6. Changing Nature of Software Development

1.1 The Evolution of Software

The Early Years (50's - mid 60's)

⇒ This generation was characterized by Batch orientation, limited distribution, and customization of software.

⇒ In Batch Processing, the system handles an entire sequence of jobs together, often with little or no human intervention.

⇒ Also, as computers were not widely used at that time, only in scientific and military institutions, software could be highly customized since distribution was limited. Job mobility was low, and the software was basically designed this way:

You wrote it, you got it working, and if it failed, you'll be the one to get it working again.

Second Era (60's - mid 70's)

⇒ This era saw the growth of software houses and the use of multiprogramming and multi-user systems, introducing with it new concepts of human-machine interaction.

⇒ Software started to be distributed in a multidisciplinary market. At this point of time, the software crisis began.

Third Era (70's - mid 80's)

⇒ This period was characterized by widespread growth and the use of personal computers.

⇒ Similarly, the use of microprocessors saw its way into use of intelligent products.

⇒ This led to the greatly increased usage of software and subsequent mushrooming of software companies.

Fourth Era (80's and beyond)

⇒ This period saw the use of increasingly powerful desktop systems, object-oriented technologies, Expert Systems, Artificial neural networks, and Parallel Computing.

⇒ The software crisis intensifies!

⇒ As we move into the fourth era, the problems associated with computer software continue to intensify:

- Hardware sophistication has outpaced our ability to build software to tap the hardwares potential.
- Our ability to build new programs cannot keep pace with the demand for new programs.
- Our ability to maintain existing programs is threatened by poor design and inadequate resources.

⇒ In addition, the quick pace in which consumers demand new software have resulted in impossible deadlines and tough schedules that software designers have to deal with. This has also resulted in other problems:

- Project overruns: i.e. the project takes far longer than expected.
- Poor quality products

1.2 Software Crisis

⇒ The problems faced in these periods of software evolution have been described as a crisis in software development. Managers responsible for software development have actually summarized these problems into several core issues:

- Schedule and cost estimates are often grossly inaccurate
- The productivity of software development people has not kept pace with the demand for their services.
- The quality of software is sometimes less than adequate.

⇒ Other related problems are that there is little data on the software development process, forestalling improvements that can be made by looking at past experiences. Similarly, poorly defined customer requirements at the start of a software development cycle has resulted in customer dissatisfaction with the completed software at the end of the cycle. Lastly, existing software itself can be very difficult to maintain.

⇒ These questions and problems faced in the history of software development have thus been instrumental to the creation and adoption of modern software engineering practices.

1.3 Software Engineering Paradigms

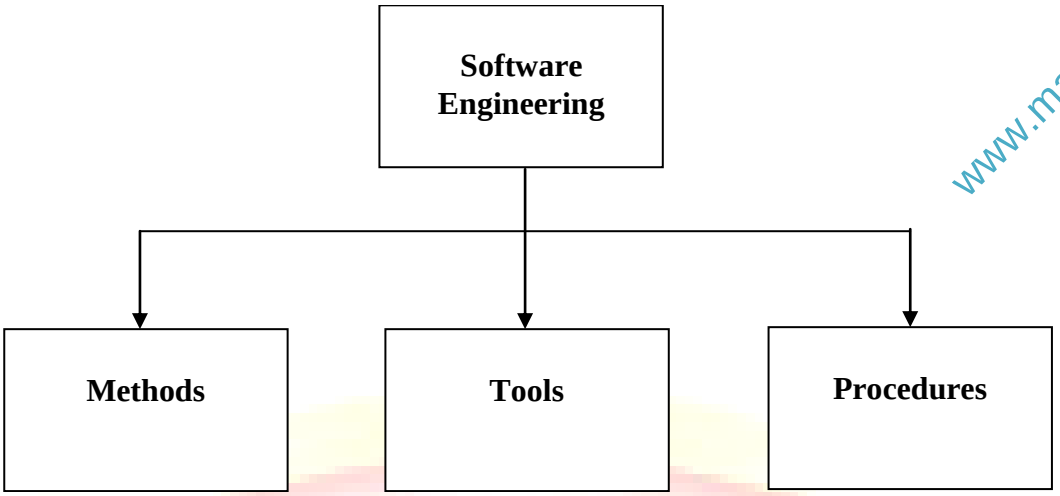
⇒ The definition of Software Engineering can thus be said to be the following:

⇒ Software Engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

⇒ The skills necessary for software engineers to adequately perform the roles in software development can be said to be of two kinds:

- Software Skills: this is the nuts and bolts of actually writing the software.
- Project Management Skills: This refers to the how-to control the development and subsequent maintenance of software.

In Software Engineering concepts, there are three key elements (also known components): methods, tools and procedures.



⇒ **Methods:** these are the how-to for building the software. Examples of these are (I) Project Planning; (2) System and Software requirement analysis; (3) Coding, testing and maintenance.

⇒ **Tools:** these are the automated or semi-automated support for methods. Examples of these are CASE, or Computer-Aided-Software-Engineering, the software equivalent of hardware design.

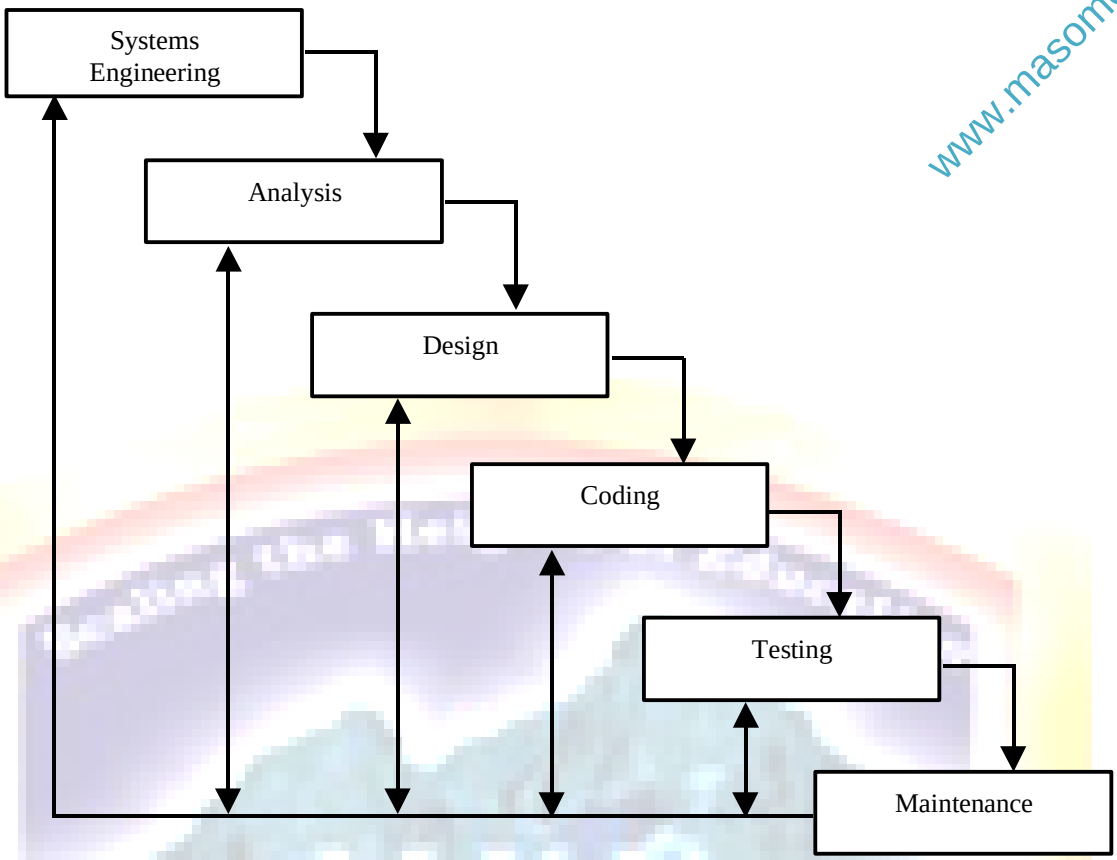
⇒ **Procedures:** this is the glue that holds the methods and tools together. It defines the sequence in which methods are applied, and makes sure that the development of software is logical (i.e. flows in correct order) and is on time too.

⇒ Software Engineering thus comprises of a set of steps that encompass each of the three elements above. These steps are often referred to as Software Engineering Paradigms. Four such paradigms are of interest to us:

- Classical Life Cycle
- Prototyping
- Fourth Generation Techniques
- A combination of all three techniques

1.3.1 The Classic Life Cycle

Also known as the Waterfall model, this life-cycle paradigm demands a systematic and sequential approach to software development. It presents a highly structured method of software development that starts at the system level, and progresses through analysis, design, coding, testing and finally maintenance.



⇒ **Systems engineering:** establish requirements for all system elements and then allocating some subset of these requirements to software. Essential for interfacing correctly with external components, e.g. databases, hardware

⇒ **Analysis:** analysis of requirements is now focused on software alone. Requirements for both system and the software are documented and reviewed with the customer.

⇒ **Design:** the multi-step process that focuses on four distinctive attributes of the program: (a) data structures; (b) software architecture; (3) Procedural detail; (4) Interface characteristics.

⇒ **Coding:** design is translated to machine readable form

⇒ **Testing:** focuses on the logical internals of the software.

⇒ **Maintenance:** errors/changes will invariably occur because software must accommodate changes in the real and external environment.

⇒ Common problems faced with the Classic Life-Cycle

- Real projects rarely follow sequential flow of the "waterfall" model, i.e. they jump to-and-fro. And iterations by itself always produces problems.
- It is difficult to determine requirements so explicitly and so early in the development. There is always a lot of uncertainty at the start of the project.
- The customer has to wait for a really long time before he even gets a feel of the real project, i.e. working version of prototype is not available early in the cycle.

⇒ Despite these problems, the Classic Life-Cycle still remains the most widely used procedural model for software engineering.

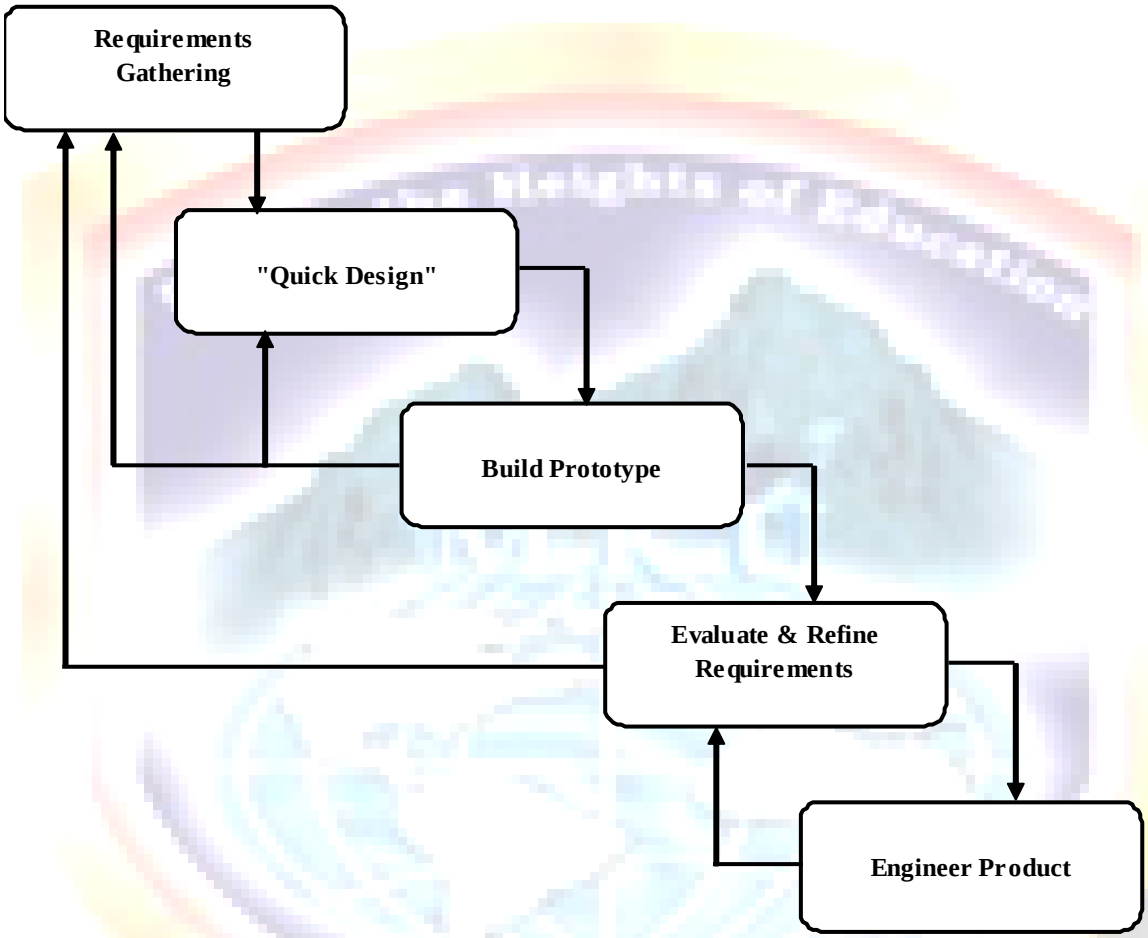
1.3.2 Prototyping Paradigm

⇒ It often happens in a typical software development cycle that the customer has defined a set of general objectives for software, when much more detailed input, processing, or output requirements are needed by the software developer. Or that the developer could be unsure of the efficiency of an algorithm, the adaptability

of an operating system, or the form that human-machine interaction should take. In these cases, a prototyping approach could be considered.

⇒ A prototype can be defined as the first or original type or model in which anything is copied. The prototyping process thus is a series of steps that enables the developer to create a model of software that must be built. The model can take one of three forms:

- Paper/PC model that depicts how human interactions will occur
- A working prototype that implements some subset of the functions required for the desired software
- An existing program that performs part of or all of the functions desired but has other features that will be improved later.



⇒ **Requirements gathering:** developer and customer meet to define overall objectives of software.

⇒ **Quick Design:** quick implementation of these aspects of the software that will be visible to user.

⇒ **Build prototype:** leads to design of prototype.

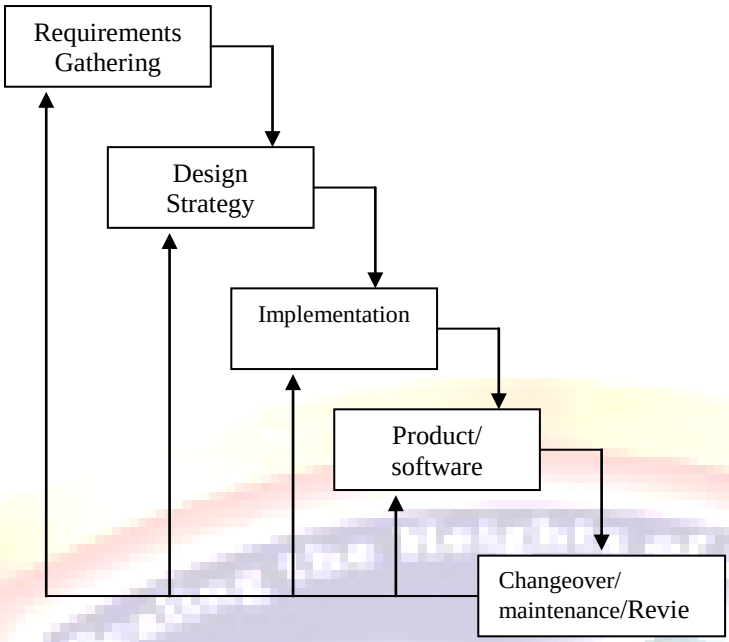
⇒ **Evaluate and Refine Requirements:** analysis of prototype; refinement.

⇒ **Engineer product**

1.3.3 Problems in Prototyping

⇒ Prototype is often rushed, and long term aspects of overall software quality and maintainability not considered.

⇒ Developer often makes implementation promises in order to get prototype working quickly.



A possible solution to this problem then is that the software developer and customer must both agree that the prototype is only meant to define requirements. It is then discarded and actual software is engineered with an eye towards quality and maintainability.

1.3.4 Fourth Generation Techniques

⇒ 4th Generation Techniques actually encompasses a broad array of software tools which have one thing in common: each of them enables the software developer to specify some characteristic of software at a high level. The tool then automatically generates source code based on the developer specification. One of the special characteristics about this technique is that the high level language used is often very close to our natural language.

⇒ Features of 4L Tools:

- Database Query
- Report Generation
- Data Manipulation
- Screen Interaction
- Code Generation
- Graphics/Spreadsheets

⇒ **Requirements Gathering.** Ideally, the customer could actually describe the requirements and these would be directly translated into an operational prototype. However, this is generally unworkable, since the customer may be unsure of what exactly is required, or introduce a degree of ambiguity in these requirements.

⇒ **Design Strategy.** Often, for smaller scale software applications, it is possible to move directly from the requirements gathering step to implementation using a non-procedural 4th Generation Language. However, for large projects, the design phase is crucial to avoid poor quality, poor maintainability, and poor customer acceptance problems later.

⇒ **Implementation Using 4GL.**

⇒ **Product**

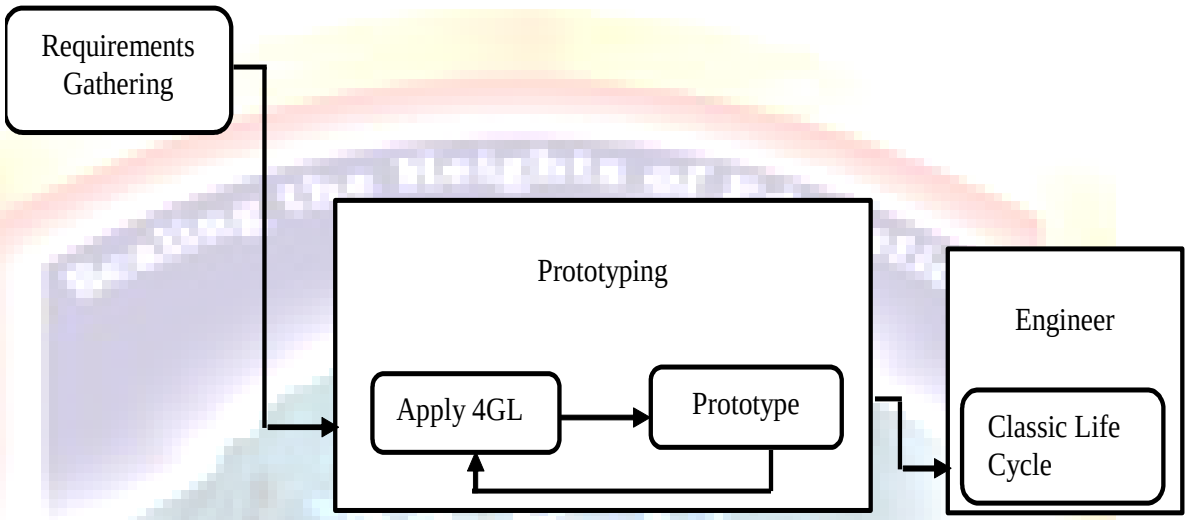
⇒ Possible Problems faced in 4GT

- Current Application domain for 4GT is limited to business information systems, for example information analysis and reporting that is keyed to large databases.
- Rapid system development is true only for small systems

- 4GTs are not substitutes for good design planning necessary particularly for large software development efforts.

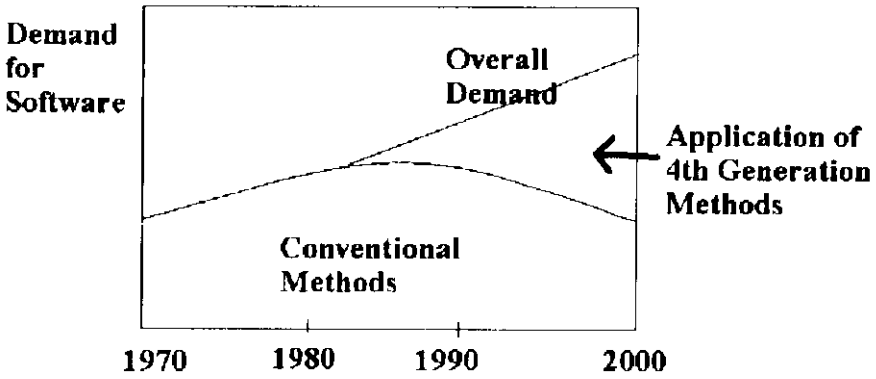
1.3.5 Combining Paradigms

⇒ In many cases, paradigms can be combined and the strengths of each can be utilized in a single project. By looking at the diagram, requirements gathering is still essential, and interaction between the developer and customer must still occur. To create the prototype, 4GL can be applied to develop the prototype quickly. Once the prototype has been evaluated and refined, the design and implementation steps of the classic life cycle can be applied to engineer the software formally.



1.4 The Changing Nature of Software Development

⇒ In the graph, a number of observations can be seen:



- The overall demand for software will continue to rise in the future.
- However, the ratio of software products developed using conventional methods and 4th generation methods will change.

Chapter Review Questions

1. Explain the concept of software engineering
2. What is the difference between a software process model and a software process? Suggest two ways in which a software process might be helpful in identifying possible process improvements.

References for Further Reading

1) Peters, J.F and Pedrycz (2000) Software Engineering: An Engineering Approach, John Wiley and sons.
2) Pressman R.S (1997)software engineering: a practitioner's Approach, McGraw Hill
3) Somerville Ian (2002) software engineering ,Pearson's education

CHAPTER 2: REQUIREMENTS ANALYSIS FUNDAMENTALS

Chapter Objectives

At the end of this chapter, student should understand: to :

⇒ Requirement Analysis

⇒ Analysis Tasks

- Problem Recognition
- Evaluation and Synthesis
- Modeling
- Specification
- Review

⇒ The Analyst

⇒ Problems in Requirement Analysis

⇒ Communication Techniques

- Preliminary Meeting or interview
- Facilitated Application Specification Technique (FAST)

⇒ Analysis Principles

- Information Domain
- Partitioning
- Essential and Implementation Views

2.1 Requirements Analysis

A formal definition for Requirements Analysis would be:

⇒ Requirements Analysis the process of discovering, refinement, modeling and specification in a software project.

⇒ Such a process will always involve both the customer and system engineer, allowing the system engineer to:

- Specify software function and performance;
- Indicate softwares interface with other system elements;
- Establish design constraints that the software must meet.The three points above can also be thought of as the objectives of Requirements Analysis

⇒ Requirement Analysis provides the software designer with a representation of information and function that can be translated to data, architectural and procedural design.

⇒ Finally, Requirement Analysis is also concerned with the preparation of the Software Specification, a formal document that specifies clearly the functional and performance requirements of the software. This Specification document in turn will allow the developer and customer to assess quality once the software itself has been built.

⇒ The software developer performing Requirement analysis would often be known as the **analyst.**

2.2 Analysis Tasks

⇒ Software requirement analysis can be divided into an ordered sequence of five main areas of effort, namely:

- Problem Recognition
- Evaluation and Synthesis
- Modeling
- Specification
- Review

⇒ **Problem Recognition**

- Goal of analyst here is recognition of the basic problem elements as perceived by user and customer
- Understand software in the system context
- Define software scope
- Analyst will also need to establish contact with management and technical staff of the customer and software development organization

⇒ **Evaluation and Synthesis**

- The analyst must now 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 uncover design constraints.
- Throughout this step, the emphasis is on what must be done, not how it will be done.
- This step will continue until both the analyst and customer feels confident that software can be adequately specified for subsequent development steps.
-

⇒ **Modeling**

- The analyst will then create models of the system that will enable better understanding of data and control flow.
- These models describe the data and control flow, functional processing, behavioral operation, and information content.
- Models serve a number of important roles:
  - Aids in understanding the information, function and behavior of a system.
  - Makes requirement analysis task easier and more systematic
  - It serves as a basis for creating specification for the software.
  - Becomes the focal point for review.
  - Becomes the foundation for design.

⇒ **Specification**

- The Specification document is now developed.
- The specification is a representation of software that can be reviewed and approved by the customer.
- Usually developed as a joint effort between the developer and the customer.

2.2.1 Specification Principles

⇒ Requirements need to be represented in a manner that ultimately leads to successful software implementation. Eight principles have been developed by Balzer and Goldman in order to help in such development of specifications.

1. **Separate functionality from implementation**
   - A specification is a description of what you want (i.e. you specify), as opposed to how it is realized (implementation).
2. **A process-oriented systems specification language is required**
   - In a dynamic environment, the behavior of the entity cannot be expressed as a mathematical function of its input.
   - Rather, a process-oriented description must be employed, in which the **"what"** specification is achieved by specifying a model of the desired behavior in terms of functional responses to various stimuli from the environment.
3. **A specification must encompass the system of which the software is component**
   - System is made up of interacting components
   - Specifications must be made in context of entire system and interaction between its parts
4. **A specification must encompass the environment in which the system operates**
   - i.e., the environment in which the system operates and how it interacts with must be specified
5. **A specification must be a cognitive model**
   - Basically, it means that the specification must describe a system as perceived by its user community.
6. **A specification must be operational**
   - i.e., the specification must be complete and formal enough such that it can be satisfy an implementation of some arbitrary test cases.
7. **A specification must be tolerant of incompleteness and augmentable**
   - i.e., the specification must be robust enough to undergo changes, expansion.
8. **A specification must be localized and loosely coupled**
   - Localized means to affect one piece only
   - Loosely coupled means that parts can be removed or added easily
   - i.e., the specification must be such that its content and structure should be able to accommodate dynamic changes, and that whatever information changes there are, it should affect one component only.

2.2.2 Review

⇒ Review of analysis documents like specification.

⇒ Review should first be conducted at a macroscopic level.

⇒ Conducted by customer and developer.

⇒ Results in modifications to

- Functions;
- Performance;
- Information representation;
- Constraints; and
- Validation criteria

2.3 The Analyst

⇒ The basic responsibility of the analyst can be said to be the following:

- To analyse and define systems of optimum performance, i.e. an output that fully meets management objectives

⇒ The analyst must also exhibit the ability to:

- Grasp abstract concept, partition them and generate solutions based on each division
- Understand implicit information, separate them and treat them individually
- Absorb pertinent facts from conflicting sources
- Understand the customer environment
- Apply hardware and/or software system elements to the customer environment
- Communicate well in written and verbal form

2.4 Problems in Requirements Analysis

⇒ Requirements analysis is a communication-intensive activity. i.e. where communication is concerned, noise, i.e. miscommunication, will always occur.

⇒ Thus problems like miscommunication and omission often occur between analyst and customer. This is often because of different levels of communication between an analyst and customer.

⇒ Successful acquisition of information cannot be guaranteed. This is because when communication fails, information can be wrongly put forward, and misinformation then occurs. And we know that accurate requirements analysis is highly dependent on getting the correct information.

⇒ Analyst have difficulties:

- In getting pertinent (appropriate) information
- Handling large and complex problems, i.e. as complexity increases, effort increases
- Accommodating changes that occur during and after analysis

2.4.1 Causes for the Problems

⇒ Poor communication that makes information acquisition difficult

⇒ Inadequate techniques and tools for developing specification

⇒ Tendency to take short-cuts during requirements analysis tasks, leading to unstable design

⇒ Failure to consider alternative solutions before software is specified on both parts of analyst and customer. i.e. are there better ways to do this?

2.5 Communication Techniques

⇒ Because of these problems, Requirements Analyze must be concerned with how to address these problems. There are two techniques that are available to tackle these problems:

- Preliminary meeting or interview
- Facilitated Application Specification Technique (FAST)

2.5.1 Preliminary Meeting or Interview

⇒ Proposed by Gause and Weinberg

⇒ The most commonly used analysis technique to bridge the communication gap between the customer and developer. Though effective for a first meeting, it should not be used for subsequent meetings between the customer and software developer.

⇒ The technique comprises of 3 different sets of questions

First Set

- These questions should lead to a basic understanding of the problem
- Focuses on the customer and overall goals and benefits.

Second Set

- These questions should enable the analyst to gain a better picture of the problem.
- Allow the customer to voice his/her perceptions about a solution.

Third Set

- Focuses on the effectiveness of the meeting itself.

2.5.2 FAST (Facilitated Application Specification Techniques)

⇒ This can be thought of a technique that is used after the first meeting is completed and basic understanding has been achieved. It proposes a meeting format that combines elements of problem solving, negotiation, and specification.

⇒ The FAST mentality encourages the working together mentality rather than working individually. Hence, the technique is always a team-oriented approach, the team jointly made up of customers and developers.

⇒ The basic goals of the FAST meeting are:

- Identify the problem
- Propose elements of the solution
- Negotiate different approaches
- Specify a preliminary set of solution requirements.

⇒ There are different approaches to FAST, but all of them have the following basic guidelines:

- Meeting is conducted at a neutral site;
- Rules for preparation and participation are established;
- An agenda is suggested to cover all the important points. Agenda must be formal enough to cover all important points, but informal enough to encourage the free flow of ideas.
- A "facilitator" is appointed to control the meeting. A facilitator is the controller, overall chairman of the meeting.

2.6 Analysis Principles

⇒ Over the years of software development, a number of analysis and specification methods have been developed. But analysis methods are related by a set of fundamental principles:

- The information domain of a problem must be represented and understood;
- Models that depict system information, function, and behavior should be developed;
- The models (and the problem) must be partitioned in a manner that uncovers detail in a layered (or hierarchical fashion);
- The analysis process should move from essential information toward implementation detail.

⇒ The Information Domain

- The information domain contains three different views of the data and control as a processed by a computer system, namely: (1) Information Flow; (2) Information Content; (3) Information Structure.

Information Flow

- Represents the manner in which data and control change as each moves through a system

Information Control

- Represents the individual data and control items that comprise some larger item of information.

Information Structure

- Represents the internal organization of various data and control items
- Ask ourselves questions like are data and control items to be organized as an n-th dimension table or as a hierarchical structure?

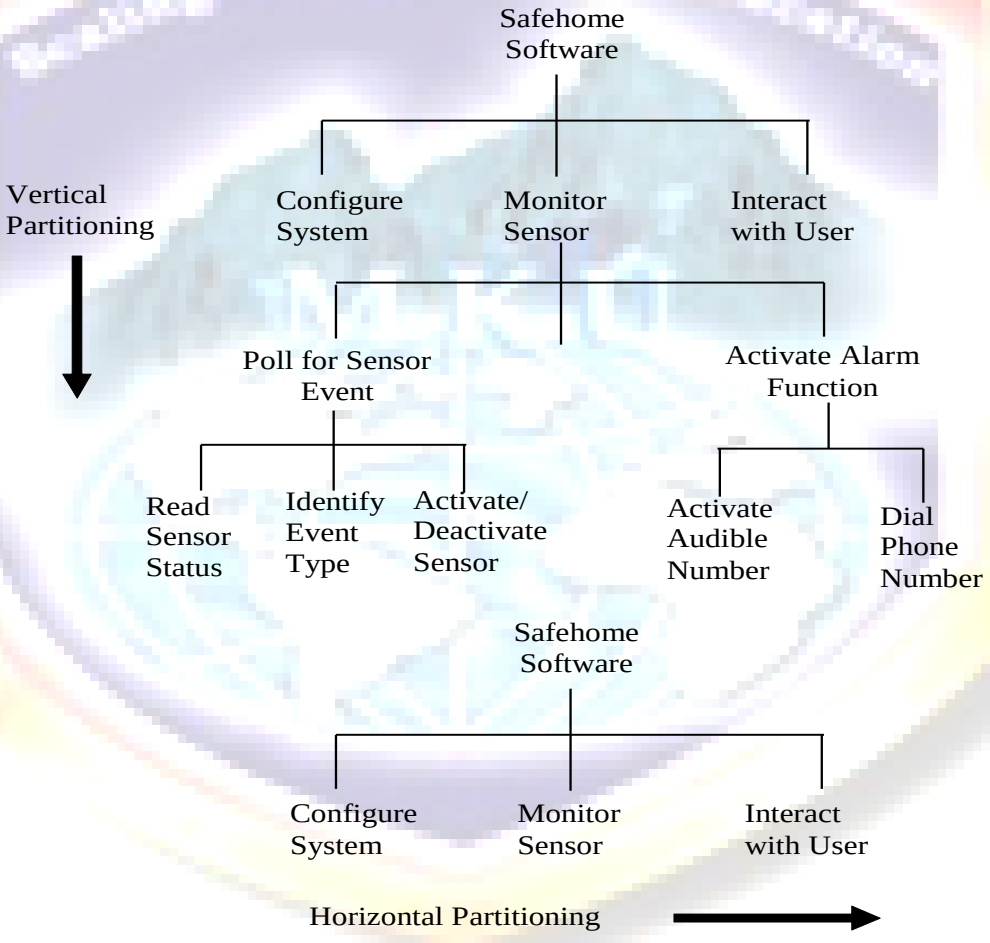
2.7 Partitioning

⇒ Problems are often too large and complex to be understood as a whole, so we partition them. The word partition means to divide into parts.

⇒ Interfaces between these divided parts must be established so that the overall function can be accomplished.

⇒ Information, functional, and behavioral domains can be partitioned.

⇒ A hierarchical representation of the function or information can then be established.



2.7.1 Partitioning Types

⇒ Two types:

- **Horizontal partitioning:** greater detail as we move from left to right
- **Vertical partitioning:** greater detail as we move from top to down

2.7.2 Essential and Implementation Views

⇒ **Essential view** presents the functions to be accomplished and information to be processed without regard to implementation details.

⇒ **Implementation view** presents the real world manifestation of processing functions and information structures.

⇒ Concerned with the physical how-is-it-going-to-be-done aspect.

Chapter Review Questions

1. outline the qualities of a good system analyst
2. Describe the various different types of non functional requirements which may be paced on a system. Give examples of these requirements.
3. write a set of non functional requirements for a an examination management system
4. Suggest how a software engineer responsible for drawing up a system requirements specification might keep track of the relationships between functional and non functional requirements of a system.

References for Further Reading

1. Kotonya.G and Somerville, I (1998), Requirements engineering: Processes and Techniques, Wiley.
2. Somerville Ian (2002) software engineering ,Pearson's education

CHAPTER 3: REQUIREMENTS ANALYSIS METHODS

Chapter Objectives

At the end of this chapter, stdent should understand::

Requirement Analysis Methods

- Definition
- Common characteristics in Requirement Analysis Methods
- Differences in Requirement Analysis Methods

Data Structure-Oriented Methods

- Data Structured Systems Development (DSSD)
- Jackson Systems Development (JSD)

Formal Methods

- Current status of Formal Methods
- Attributes of Formal Specification Languages
- Case Study: A Formal Specification in Z
- The Road Ahead of Formal Specification

Automated Techniques for Requirement Analysis

3.1 Requirements Analysis Methods

⇒ Definition of Requirement Analysis Methods:

- Requirement analysis methods enable an analyst to apply fundamental analysis principles in a systematic fashion.

⇒ Requirement analysis methods enable an analyst to apply fundamental analysis principles in a systematic fashion. But all methods share a number of fundamental and common characteristics. They:

- each supports the fundamental requirements analysis principles
- each creates a hierarchical representation of a system
- each demands a careful consideration of external and internal interfaces
- each provides a foundation for the design and implementation steps that follow

3.1.1 Common Characteristics of Requirement Analysis Methods

Although each method introduces new notation and analysis heuristics, all methods can be evaluated in the context of the following common characteristics:

⇒ Mechanism for information domain analysis, i.e. all analysis methods addresses (either directly or indirectly) information flow, information content, and information structure.

⇒ Approach for functional and/or behavioral representations

- All functions/behaviors are typically represented by specific notation.

⇒ Definition of interfaces

- Interfaces are derived from an examination of information flow.

⇒ Mechanisms for problem partitioning

- Problem partitioning is accomplished by a layering process that allows for representation of information and function domain at different levels of abstraction.

⇒ Support for abstraction: abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details.

- All methods provide mechanisms for partitioning a function into a set of sub-functions.

⇒ Representation of essential and implementation views.

3.1.2 Differences in Requirement Analysis Methods

⇒ Each method for the analysis of computer-based systems has its own point of view, its own notation, and its own approach to modeling. Also, each method has its own jargon and terminology.

⇒ The degree to which the method establishes a firm foundation for design differs greatly too. In some cases, the analysis model can be mapped directly into a working program. In other cases, the analysis method establishes a starting point only and the designer is left to derive the design with little help from the analysis model.

⇒ In this chapter, three broad categories of Requirement Analysis methods will be discussed:

- Data Structure-Oriented Methods
- Formal Methods
- Automated Techniques for Requirement Analysis

3.2 Data Structure-Oriented Methods

Data Structure-oriented methods represent software requirements by focusing on data structures rather than data flow.

⇒ Although each data structure-oriented methods has a distinct approach and notation, all have some characteristics in common:

- Each assist the analyst in identifying key information objects (also called entities or items) and operations (also called actions or processes).
- Each assumes that the structure of information is hierarchical;
- Each requires that the data structure be represented using the sequence, selection, and repetition constructs for composite data;
- Each provides a set of steps for mapping a hierarchical data structure in to a program structure.

⇒ Data structured-oriented analysis methods are:

- Data Structured Systems Development
- Jackson System Development

3.2.1 Data Structured Systems Development (DSSD)

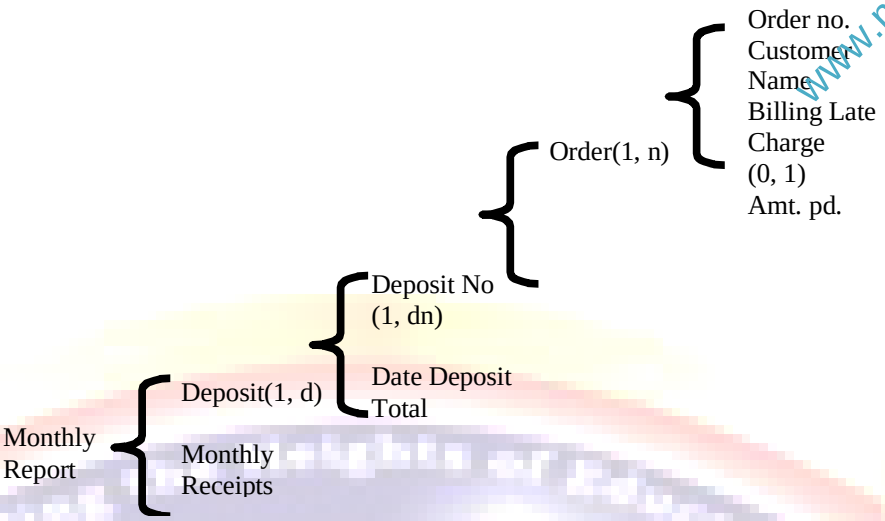
⇒ Also known as the Warnier-Orr Methodology: J.D.Warnier developed a notation for representing an information hierarchy using the three constructs for sequence, selection, and repetition and demonstrated that the software structure could be derived directly from the data structure. Ken Orr extended Warnier work to encompass a broader view of the information domain that eventually evolved into a data structured systems development. DSSD considers information flow and functional characteristics as well as data hierarchy.

3.2.2 The DSSD Approach

⇒ Rather than examining the information hierarchy, DSSD first examines the **application context**, that is, how data moves between producers and consumers of information from the perspective of one of the producers or consumers.

⇒ Next, **application functions** are assessed with a Warnier-like representation that depicts information items and the processing that must be performed on them.

⇒ Finally, **application results** are modeled using the Warnier diagram.



Jackson System Development (JSD)

⇒ Focuses on models of the "real-world" information domain

3.3.3 The JSD Approach

⇒ Entity action step:

- Identify entities (people, objects, or organizations that a system needs to produce or use information) and actions

⇒ Entity structure step:

- Using Jackson diagrams, order by time the actions affecting each entity

⇒ Initial modelling step:

- Represent entities and actions as process model; define connections between the model and the real world

⇒ Function step:

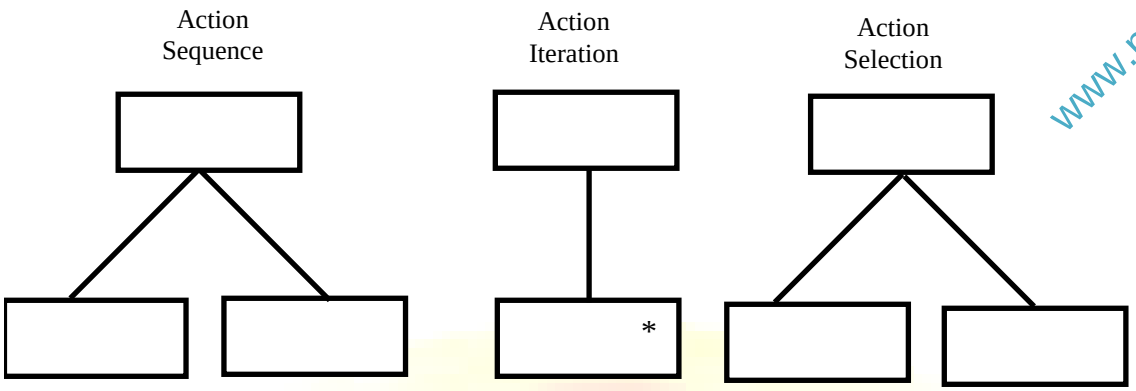
- Specify functions that correspond to defined actions

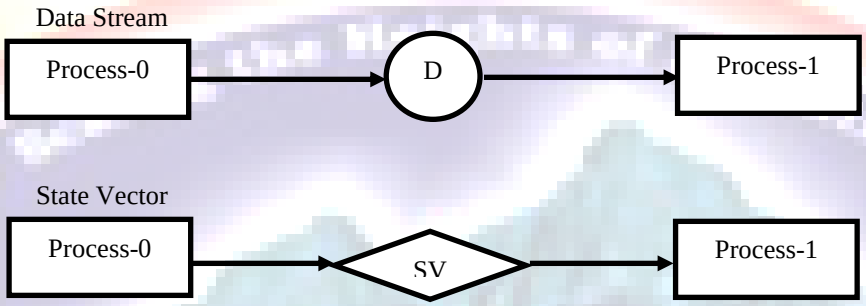
⇒ System timing step:

- Assess process scheduling characteristics

⇒ Implementation step:

- Specify hardware and software as a design





3.3 Formal Specification Techniques

⇒ Categorized on a "formality" spectrum

⇒ A specification is described using a formal syntax and semantics to specify system function and behavior

3.3.1 Current Status of Formal Methods

⇒ A formal specification is mathematical in form. For e.g. predicate calculus is used as the basis for a formal specification

⇒ Ambiguity, incompleteness and inconsistency can be discovered and corrected easily through mathematical analysis

⇒ When used during design, formal methods consider a software problem in a manner that is analogous to an algebraic derivation or a proof in analytical geometry

3.3.2 The Attributes of Formal Specification Languages

⇒ Consists of three components

- A syntax that defines the specific notation with which the specification is represented
- Semantics that helps to define a "universe of objects"
- A set of relations that defines the rules that indicate which objects satisfy the specification
- Syntactic domain is based on a syntax that is derived from standard set theory notation and predicate calculus
- For e.g. x,y,z to represent a set of objects; logic symbols like A, E V that mean for all, there exists and, respectively
- Semantic domain indicates how the language represents system requirements
- Comparison between programming language and formal specification language

Programming Language	Formal Specification Language
Specify algorithms that transform input to output	Describe the syntax of the programming language
Does not make a good specification language because it represent only computable forms	Must be capable of expressing ideas such as "For all x in an infinite set A. there exists a y in an infinite set B such that the property P holds for x and y

- Other specification languages can also develop syntax and semantics to specify states and state transition, events and their affect on state transition and synchronization and timing

Case Study: A Formal Specification in Z

⇒ The Z specification language has been used to illustrate the practical use of a specification language

⇒ A case study is reported to have made use of Z notation, a mathematical specification language, to specify the kernel for a diagnostic x-ray machine

⇒ The goal is to produce a precise specification that could be implemented on different hardware

⇒ This case study in specification reflected a flaw in the kernel design which is proven by the mathematical property of its specification

⇒ Hence, formal techniques can help to avoid error especially in embedded systems which are very difficult to test effectively

3.3.3 The Road Ahead of Formal Specification

⇒ Although formal techniques have advantages, there do exists problems

⇒ Formal specification emphasizes only on function and data

⇒ Timing, control and behavioural aspect of a problem more difficult to represent

⇒ Other elements such as human-machine interface better specified using graphical techniques or prototyping

⇒ Formal specification techniques are difficult to learn

3.4 Automated Techniques for Requirement Analysis

⇒ Automated Techniques can be categorized as follows:

- Manual method that has been completed by an automated CASE tool. These tools can produce diagrams, aid in problem partitioning, maintain a hierarchy of information about the system, and applies heuristics to uncover problems with the analyst.
  - E.g. DEC Design (Digital Equipment Corp), DesignAid (Transform Logic Corp.)
- Another class makes use of special notation that has been explicitly designed for a processing using an automated tool. Requirements are described with a specification language that combines keyword indicators with a natural language. The specification language is fed to a processor that will produce a requirements specification, and diagnostic reports indicating the consistency and organization of the specification.
  - E.g. SREM: Software Requirements Engineering Methodology, PSL/PSA: Program Statement Language/Problem Statement Analyzer).

⇒ Benefits of Automated Techniques:

- Improved documentation through standardization and reporting
- Easier detection of gaps, omissions, and inconsistencies.
- Easier tracing of the impact of modifications
- Reduction in maintenance costs for the specification.

Chapter Review Questions

1. Explain why it is almost inevitable that requirements from different stakeholders in a system development will conflict in some ways.
2. Discuss the concept of formal specification and outline its major weaknesses

References for Further Reading

1. Kotonya.G and Somerville, I (1998), Requirements engineering: Processes and Techniques, Wiley.
2. Somerville Ian (2002) software engineering ,Pearson's education

CHAPTER 4: SOFTWARE DESIGN

Chapter Objectives

At the end of this chapter,student will explain:

Software Design

Data Design Principles

Architectural Design

Procedural Design

- Graphical Design Notation
- Tabular Design Notation
- Program Design Language

Software Design Fundamentals

- Modularity
- Abstraction
- Software Architecture
- Control Hierarchy
- Data Structure
- Software Procedure
- Information Hiding
- Functional Independence (Cohesion andCoupling)

Criteria for Good Design

4.1 Software Design

The Software Design process can be defined as a process through which requirements are translated into a representation of software. From a project management point of view, software design can be conducted in two main steps:

Preliminary Design
Concerned with the transformation of requirements into data and software architecture.

Detail design
Focuses on refining the architectural representation, and lead to detailed data structure and algorithmic representations of software.

Within the preliminary and detail design, a number of different design activities occur. Besides the three main design activities, i.e. Data design, Architectural design, and Procedural design, there is also the Interface design, which establishes the layout and interaction mechanisms for human-machine interaction.

The three main design activities concerned in the Design phase are: Data design, Architectural design and Procedural design. In addition, many modern applications have a distinct interface design activity. Interface design establishes the layout and interaction mechanisms for human-machine interaction.

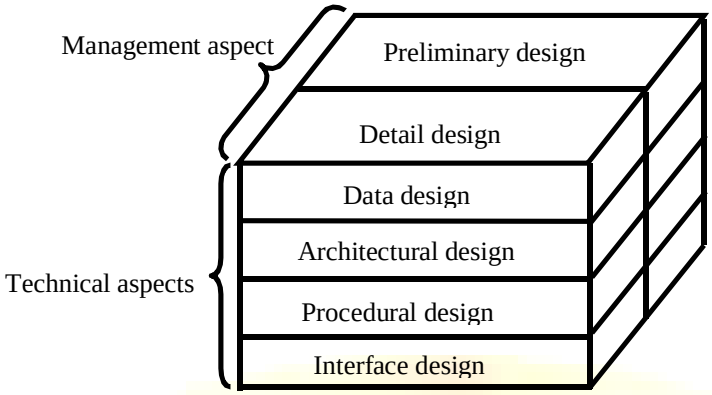


Diagram: Relationship between technical and management aspects of design

4.2 Data Design

⇒ Data design is the first (and sometimes the most important) of the three design activities that are conducted in software engineering. The objective of Data Design is to transform the information domain into data structures.

⇒ A set of principles was proposed by Wasserman that may be used to specify and design data. This set of principles are quite similar to the set of principles studied in requirement analysis.

(1) The systematic analysis principles applied to function and behavior should also be applied to data

- The same good principles that are followed in analysis principles should also be applied to help us develop data flow and content, identify data objects, and consider alternative data organization.

(2) All data structures and the operations to be performed on each should be identified.

- The design of an efficient data structure should consider the operations that will be performed on the data structure itself.

(3) A data dictionary should be established and used to define both data and program design.

- A data dictionary explicitly represents the relationships among data objects and the limitations on the elements of a data structure.

(4) Low-level data design decisions should be deferred until late in the design process.

- In data design, a top-down approach should be used.

(5) The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.

- Information Hiding must be practiced.

(6) A library of useful data structures and the operations that may be applied to them should be developed.

- Data structures should be designed for reusability.

(7) A software design and programming language should support the specification and realization of abstract data types.

- The programming language for use should support the creation of complex data structures.

4.3 Architectural Design

⇒ The primary objective in Architectural Design is to develop a modular program structure and represent the control relationships between modules.

⇒ In addition, architectural design combines program structure and data structure, thus defining interfaces that will enable data to flow throughout the program.

4.4 Procedural Design

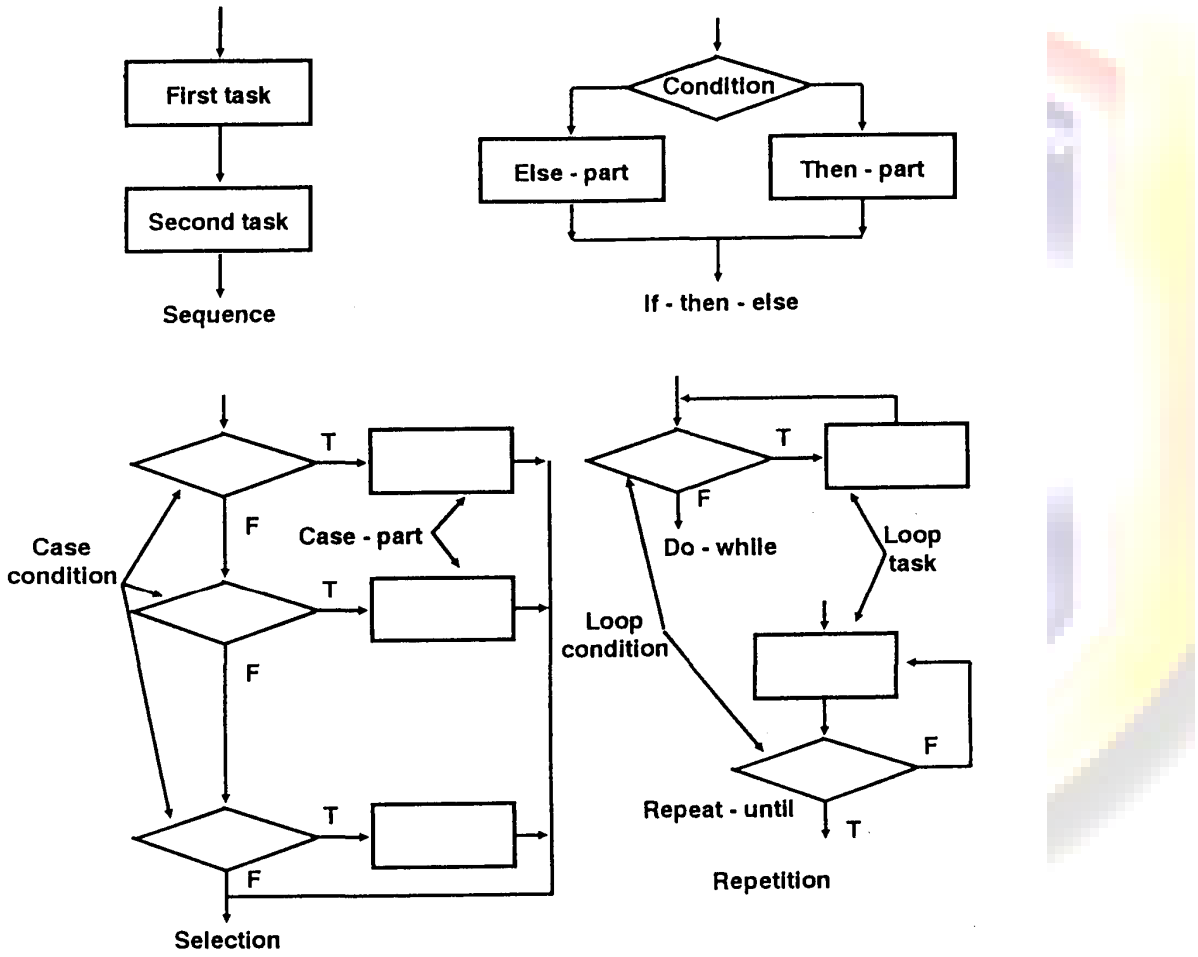
⇒ The objective in Procedural Design is to transform structural components into a procedural description of the software.

⇒ This step occurs after the data and program structures have been established, i.e. after architectural design. Procedural details can be represented in different ways:

- Graphical Design Notation
- Tabular Design Notation
- Program Design Language (PDL)

⇒ Graphical Design Notation

- The most widely used notation is the flowchart. Some notation used in flowcharts are (I) Boxes to indicate processing steps; (II) Diamond to indicate logical conditions; (III) Arrows to indicate



flow of control; (IV) Two boxes connected by a line of control will indicate a Sequence.

⇒ Tabular Design Notation

- Decision tables provide a notation that translates actions and conditions (described in a processing narrative) into a tabular form.
- The upper left-hand section contains a list of all conditions. The lower left-hand section lists all actions that are possible based on the conditions. The right-hand sections form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination.

Conditions	Fixed rate account	1	2	3	4	5
	Variable rate account	T	T	F	F	F
	Consumption <100KWH	T	F	T	F	

	Consumption ≥100KWH	F	T	F	T	
Actions	Minimum monthly charge	X				
	Schedule A billing		X	X		
	Schedule B billing				X	
	Other treatment					X

⇒ Program Design Language

- Program Design Language (PDL) is also called structured English, or Pseudocode.
- The main difference between PDL and its nearest neighbor, 4th Generation Languages, is that in PDL, the use of narrative text (e.g. English) is embedded directly within PDL statements.

PDL have the following characteristics:

⇒ A fixed syntax of keywords that provide for all structured constructs, data declaration, and modularity characteristics

⇒ A free syntax of natural language that describes processing features

⇒ Data declaration facilities that should include both simple (scalar, array) and complex (linked list or tree) data structures.

⇒ Subprogram definition and calling techniques that support various methods of interface description.

4.5 Software Design Fundamentals

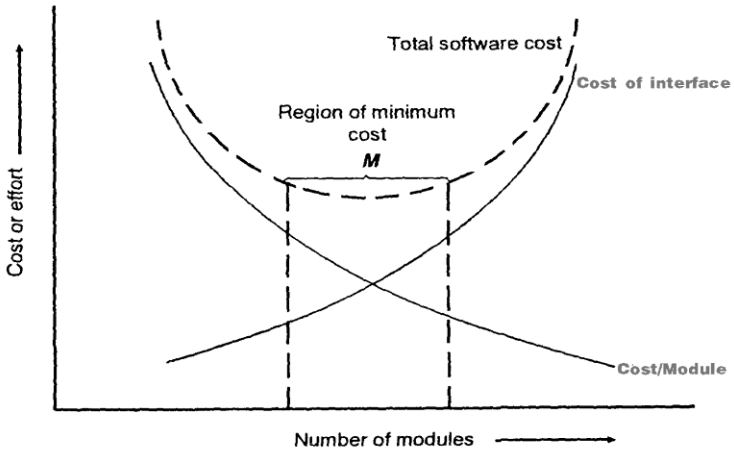
⇒ In this part of the chapter, the following Software Design Fundamentals will be discussed.

- Modularity
- Abstraction
- Software Architecture
- Control Hierarchy
- Data Structure
- Software Procedure
- Information Hiding
- Functional Independence

4.5.1 Modularity

⇒ Definition of Modularity:

- Software is divided into separately named an addressable components, called modules, that are integrated to satisfy problem requirements.



- The graph shown refers to the question of Modularity and Software Cost, and is a useful tool to consider when modularity is to be implemented. In this graph, the cost or effort drops as more modules are considered, but the cost to interface modules increases as we have more modules. Thus, we must take care to stay in the region of minimum cost, i.e. a balance between the number of modules and the cost or effort.

4.5.2 Effective Modular Design

⇒ Modularity is an accepted approach in all engineering disciplines

⇒ Reduces complexity and facilitates changes in modules

⇒ Easier implementation

4.5.3 Desirable Characteristics of Module

⇒ The attributes of a good module are as follows:

- A small program that can be invoked by the operating system, or it could be a sub-program invoked by another module(shareable)
- The statement are collectively referred to by a descriptive name called the module name
- A module must return to its caller i.e. have a single entry and exit; (single entry and exit module to ensure that modules are closed and simplify program maintenance)
- The module should be relatively small in size (small modules allow for more easily amended programs, estimating and project control more accurate and exhaustive testing are easier)
- It should be easy to read, modify and use
- A module should preferably have a single function

4.5.4 Advantages and Disadvantages of Modularity

⇒ Rationale for Modularity

- Allow large program to be written by several or different people
- Encourage creation of commonly used routines to be placed in library and/or be used by other programs
- Simplify overlay procedure of loading large program into main storage
- Provide more check point to measure progress

- Simplify design, making program easy to modify and reduce maintenance costs
- Provide a framework for more complete testing, easier to test
- Produces well-designed and more readable program

⇒ Rational Against Modularity

- Execution time may be, but not necessarily, longer
- Storage size may be, but is not necessarily, increased
- Compilation and loading time may be longer
- Intermodule communication problems may be increased
- Demands more initial design time
- More linkage required, run-time may be longer, more source lines must be written and more documentation has to be done

4.5.5 Abstraction

⇒ Abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details.

⇒ Use of abstraction also permits one to work with concepts and terms that are familiar, in the problem environment without having to transform them to an unfamiliar structure.

4.5.6 Software Architecture

⇒ Software Architecture can be said to refer indirectly to two important characteristics:

- The hierarchical structure of procedural components (modules) and
- the structure of data

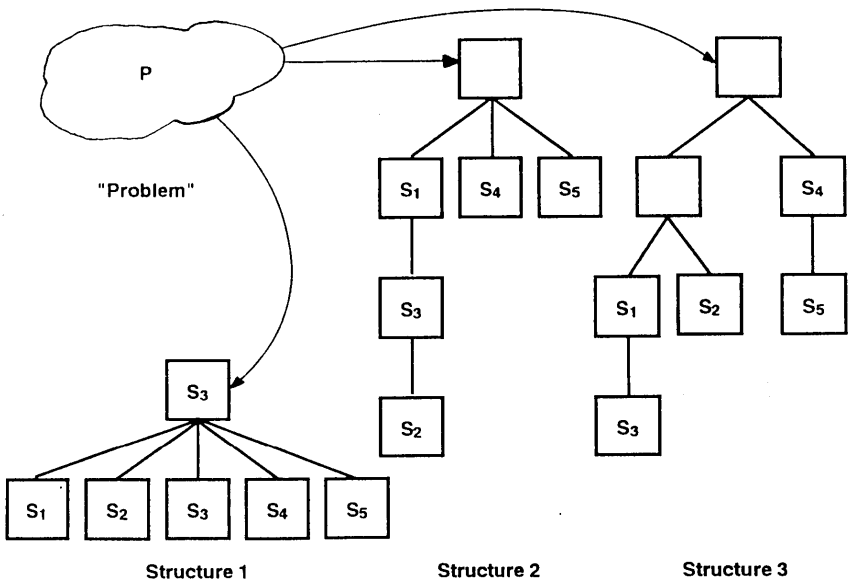
⇒ Software architecture is derived through a partitioning process that relates elements of a software solution to parts of a real-world problem implicitly defined during requirements analysis.

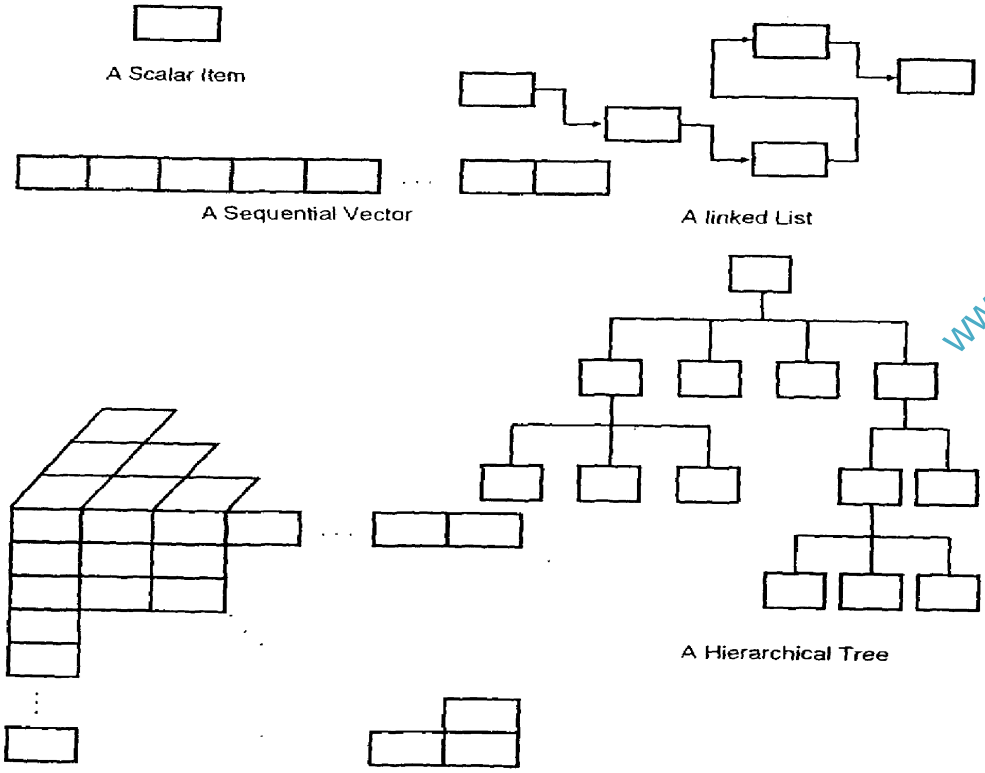
⇒ Thus, in the partitioning process, a big problem is broken up into different software solutions, and the problem may thus be satisfied by many different candidate structures.

4.5.7 Control Hierarchy

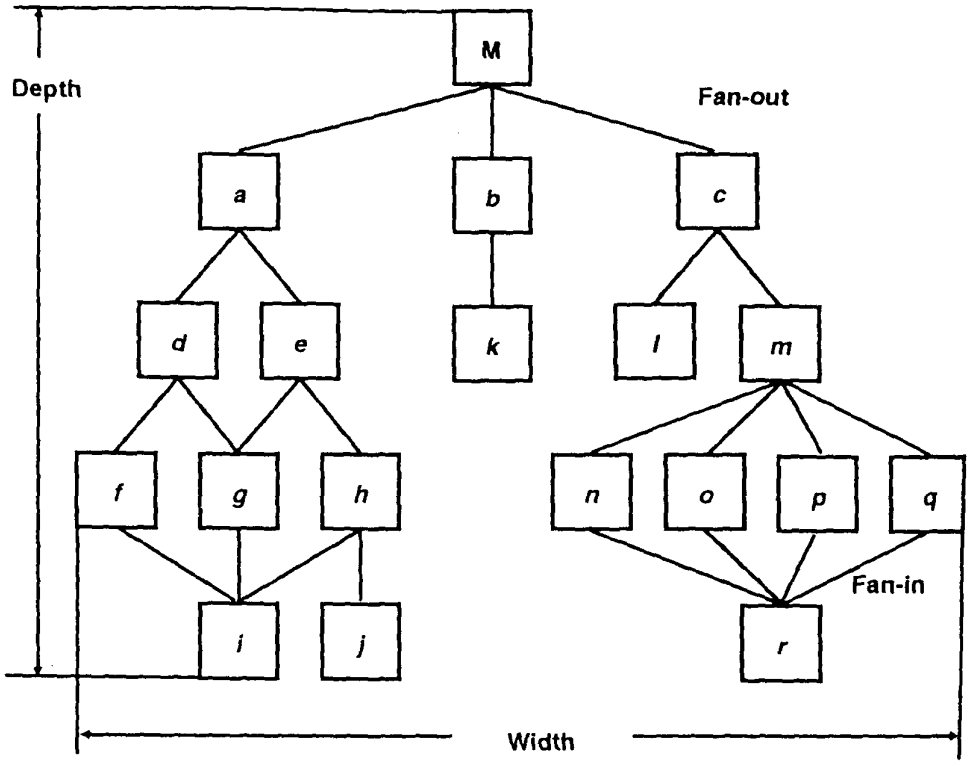
⇒ Control Hierarchy, also called program structure, represents the organization which is often hierarchical, of program components (modules) and implies a hierarchy of control.

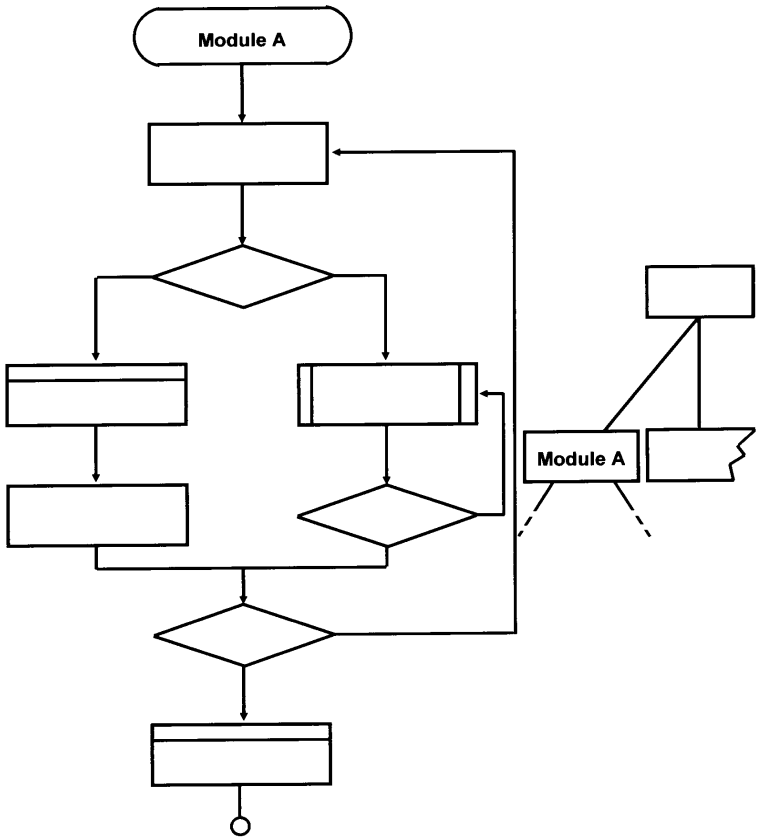
⇒ There are many different notations used to represent control hierarchy, for example the





An **n** - dimensional Space



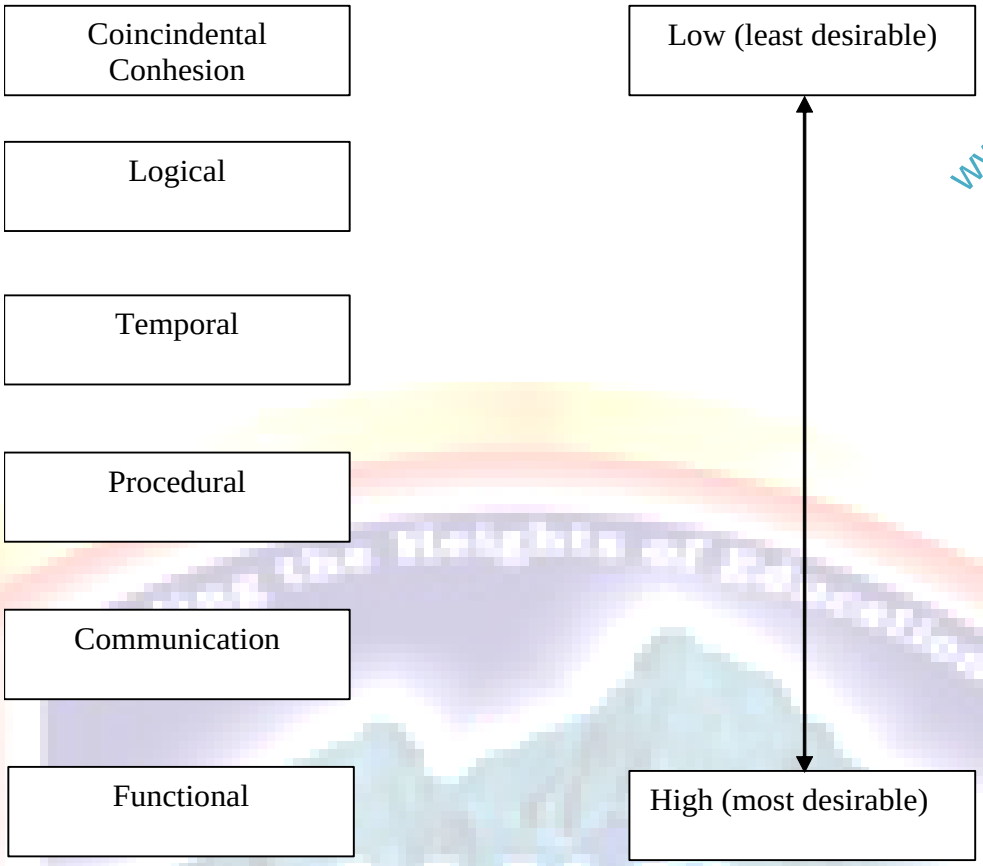


- N-dimensional array: result when the sequential vector is extended into an n-dimensional space.
- Linked list: a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner (called nodes) that enables them to be processed as a list.
- Hierarchical data structure: implemented using multilinked lists that contain scalar items, vectors, and possibly, n-dimensional spaces.

4.5.9 Software Procedure

⇒ Software procedure focuses on the processing details of each module individually.

⇒ Software procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization/structure.



4.6 Information Hiding

⇒ The principle of information hiding suggests that modules be characterized by design decisions that (each) hides from the others.

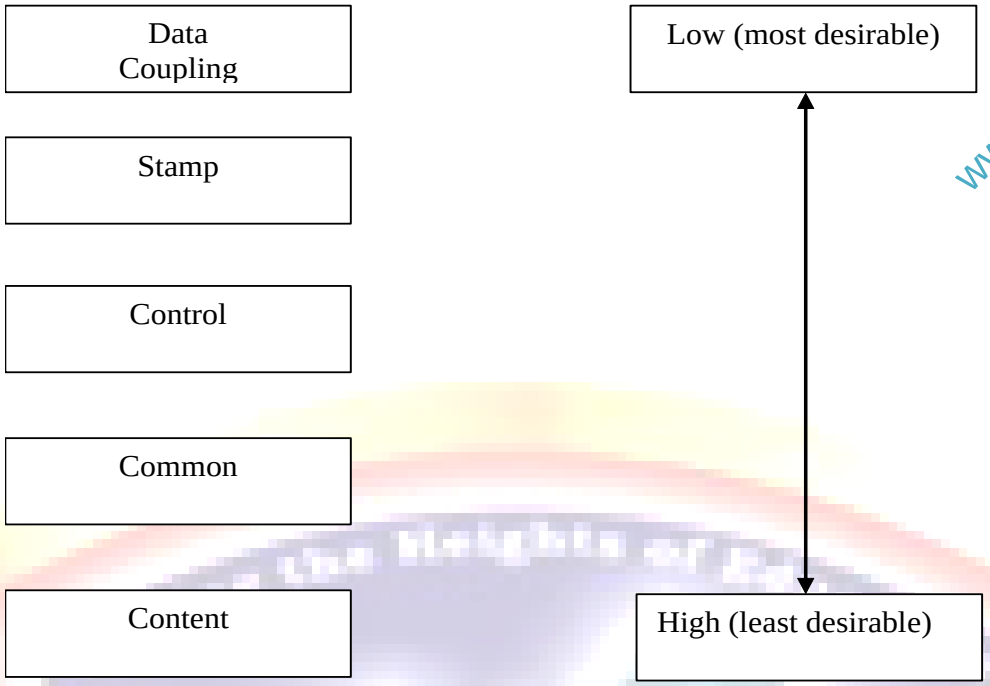
⇒ In other words, modules should be specified that information (procedure and data) contained within a module are inaccessible to other modules that have no need for such information.

⇒ The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. This is because as most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to other locations within the software.

4.7 Functional Independence

⇒ The concept of functional independence comes from a direct outgrowth of modularity and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with single-minded function and an aversion to excessive interaction with other modules. In other words, design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure.

- Independence is measured using two qualitative criteria: cohesion and coupling.
  - Cohesion: measures the relative functional strength of a module
  - Coupling: measures the relative interdependence among modules.



4.7.1 Cohesion

⇒ A module with high cohesion is able to perform a single task within a software procedure, requiring little interaction with procedures being performed in other parts of the program.

⇒ In other words, a cohesive module does just one thing and sticks with it.

⇒ Low Levels

- Coincidental Cohesion: a module that performs a set of tasks that relate to each other loosely, if at all.
- Logical Cohesion: a module that performs tasks that are related logically (e.g., a module that produces all output regardless of type)
- Temporal Cohesion: a module that performs tasks that are related by the fact that all must be executed with the same span of time.

⇒ Moderate Levels

- Procedural Cohesion: this happens when the processing elements of a module are related and must be executed in a specific order.
- Communication Cohesion: when all processing elements concentrate on one area of a data structure.

⇒ High Levels

- High cohesion is characterized by a module that performs one distinct procedural task.

4.7.2 Coupling

⇒ Coupling is a measure of interconnection among modules in a software structure.

- Low Coupling Levels (desirable)
- Data coupling: for example passing of simple data like an argument list from one module to another module.
- Stamp coupling: a portion of data structure, rather than a argument list, is passed via a module interface.

⇒ Moderate Coupling Levels

- Control Coupling: most common in software. Where control is passed via a flag on which decisions are made in a subordinate or superordinate module.

⇒ High Coupling Levels

- Common Coupling: occurs when a number of modules reference a global data area.
- Content Coupling: occurs when one module makes use of data or control information maintained within the boundary of another module.

4.8 Criteria for Good Design

⇒ A design should:

- Exhibit a hierarchical organization that makes intelligent use of control among components of software
- Be modular; that is, the software should be logically partitioned into components that perform specific functions and subfunctions
- Contain distinct and separable representation of data and procedure
- Lead to modules that exhibit independent functional characteristics
- Lead to interfaces that reduce the complexity of connections between modules and with the external environmental
- Be derived using a repeatable method that is driven by information obtained during software requirements analysis

Chapter Questions

1. Distinguish between coercion and cohesion
2. Explain the benefits of modular system design
3. What is user interface design? Outline any three types of user interfaces. Name any three types of user interface errors
4. Identify major technical and non technical factors that may hinder software re-use

References for further reading

1) Pressman R.S (1997)software engineering: a practitioner's Approach, McGraw Hill
2) Somerville Ian (2002) software engineering ,Pearson's education

CHAPTER 5: PROGRAMMING LANGUAGES

Chapter Objectives

At the end of this chapter, stdent should understand::

⇒ Programming Languages
- Definition
- The Translation Process

⇒ Programming Language Characteristics
- Psychological View
- Engineering View

⇒ Choosing a Language

⇒ Programming Language Fundamentals
- Data Types and Data Typing
- Subprograms
- Control Structures
- Support for Object-Oriented Approaches

5.1 Programming Languages

⇒ Definition: a form that can be "understood" by the computer.

5.1.1 The Translation Process

⇒ The coding step translates a detail design representation of software into a programming language realization.

⇒ The translation process continues when a compiler accepts source code as input and produces machine-dependent object code as output.

⇒ Compiler output is further translated into machine code, and these are the actual instructions that will actually be used by the central processing unit.

5.2 Programming Language Characteristics

⇒ The coding process can be viewed firstly as communication via a programming language- i.e. it is a human activity. Therefore, attention must be paid to the psychological characteristics of a language.

⇒ The coding process may also be viewed as one step in the software engineering process. The engineering characteristics of a language therefore, also have an important impact on the success of a software development project.

5.2.1 Psychological View

⇒ The role of the software psychologist is to focus on human concerns. Some of these concerns are:
- Ease of use
- Simplicity in learning
- Improved reliability

- Reduced error frequency
- Enhanced user satisfaction

⇒ At the same time maintaining an awareness of machine efficiency. Software capacity, and hardware constraints. These human aspects of computer-based system development must thus be taken into account.

5.2.2 Psychological View Characteristics

⇒ A number of psychological characteristics occur as a result of programming language design.

⇒ Uniformity

- Indicates the degree to which a language uses consistent notation, applies arbitrary restrictions.

⇒ Ambiguity

- Refers to the situation where a programming language is perceived by the programmer in one way, but the compiler always interprets the language in another way.

♦ Compactness

- Indicates the amount of code-oriented information that must be recalled from human memory.

⇒ Locality

- Measure of how much of a language that can be implemented as a "whole". Locality is enhanced when statements can be combined into blocks, and when design and resultant code are highly modular and cohesive.

⇒ Linearity

- A psychological characteristic that is closely associated with the concept of maintenance of functional domain, i.e. human perception is facilitated when a linear sequence of logical operations is encountered. A programming language that does extensive branching violates the linearity of processing.

⇒ Tradition

- Refers to a human trait of familiarity. In other words, a programmer with experience in one form of language will find it easy to pick up another language that has the same sort of constructs in the former.

⇒ The psychological characteristics of programming languages have an important bearing on our ability to learn, apply and maintain them.

5.2.3 Engineering View

⇒ The engineering view of programming language characteristics focuses on the needs of specific software development project. A general set of engineering characteristics can be said to be as follows:

- Ease of design to code translation
- Compiler efficiency
- Source code portability
- Availability of development tools
- Maintainability

⇒ Ease of design to code translation

- Indicates how closely a programming language can represent a design representation.

⇒ Compiler efficiency

- Many applications today still require fast, "tight" (i.e. low memory requirements) programs. Languages with optimizing compilers may be attractive if software performance is a critical requirement.

⇒ Source code portability

- This generally refers to whether source code may be transported from processor to processor and compiler to compiler with little or no modification.

⇒ Availability of development tools

- These can shorten the time required to generate source code and can improve the quality of code.

⇒ Maintainability of source code

- Source code must be easy enough to understand so as to allow modifications according to changes in design.

5.3 Choosing a Language

⇒ The choice of a programming language for a specific project must take into account both engineering and psychological characteristics.

⇒ The criteria that can be applied during an evaluation of available languages can be

- General application area
- Algorithmic and computational complexity
- Environment in which software will execute
- Performance considerations
- Data structure complexity
- Knowledge of software development staff
- Availability of a good compiler or cross-compiler.

⇒ C is often the language of choice for the development of systems software, while languages such as Ada and Modula-2 are often used in Real-time applications. In the engineering/scientific area, FORTRAN remains the predominant language. Widely-used object oriented programming languages are C++, Small Talk.

⇒ Although there are many "new and better" programming languages, sometimes it could also be better to choose a "weaker" (old) language that has solid documentation and support software, is familiar to everyone on the software development team and has been successfully applied in the past.

5.4 Programming Languages and Software Engineering

⇒ Regardless of the software engineering lifecycle in consideration, programming language will have an impact on project planning, analysis, design, coding, testing and maintenance. The quality of the end result and product is often closely tied to the software engineering activities that precede and follow coding.

⇒ During the project planning, consideration of the technical characteristics of a programming language is rarely undertaken. After software requirements have been established, the technical characteristics of the candidate programming language becomes more important. If complex data structures are required, languages with sophisticated data structure support (e.g. PASCAL) would be necessary. If high-performance, real-time capability is paramount, ADA would be appropriate. If memory-speed efficiency is in consideration, C would be more appropriate.

⇒ The quality of software design is established in a manner that is often independent of programming language characteristics. However, language attributes do play a role in the quality of an implemented design and can affect the way that design is specified. In some instances, a complex data structure in design can only be satisfied by specific programming languages.

5.5 Programming Languages Fundamentals

⇒ The technical characteristics of programming languages span an enormous number of topics. This section introduces a brief discussion of programming language fundamentals.

5.5.1 Data Types and Data Typing

⇒ Data types and data typing can be described as a class of data objects together with a set of operations for creating and manipulating them. Simple data types are often numeric types (e.g. integer, complex, floating point numbers), enumerated types (user defined data types), Boolean types (e.g. true or false), and string types (e.g. alphanumeric data). More complex data types could be from simple one-dimensional arrays to list structures and multi-dimensional arrays.

⇒ The operations that can be performed on a particular data type and the manner in which different types can be manipulated in the same statement is controlled by ***type checking.***

⇒ There are five levels of type checking.

- Level 0: typeless
- Level 1: automatic type coercion
- Level 2: mixed mode
- Level 3: Pseudostrong type checking
- Level 4: strong type checking

⇒ Typeless:

- programming languages have no explicit means for data typing and therefore, do not enforce type checking.

⇒ Automatic-type coercion:

- a type checking mechanism that allows the programmer to mix different data types, but then converts operands of incompatible types, thus allowing requested operations to occur.

⇒ Mixed-mode type conversion:

- similar to automatic type coercion. Different data types within the same type category are converted to a single target type so that a specified operation can occur.

⇒ Pseudostrong-type checking:

- similar to strong-type checking, but is implemented in a manner that provides one or more loopholes.

⇒ Strong-type checking:

- the programming language will only permit operations to be performed on data objects that are of the same data type.

5.5.2 Subprograms

⇒ A separately compatible program component that contains a data and control structure. A subprogram exhibits a number of generic characteristics:

- a **specification section** that includes its name and interface **characteristics**
- an **implementation section** that includes data and control structures
- an **activation mechanism** that enables the subprogram to be invoked from elsewhere in the program.

5.5.3 Control Structures

⇒ All modern programming languages enable the programmer to represent sequence, condition, and repetition- the structured programming logical constructs.

5.5.4 Support for Object-oriented Approaches

⇒ Support for object-oriented approaches should be built directly into the programming language that will be used to implement an object-oriented design.

- Definition: identifies the class
- Private data: attributes whose values are private to individual instances of the class
- Shared data: attributes whose values are shared by all instances of the class
- Pool data: attributes whose values are shared across multiple classes
- Instance methods: the procedures that implement messages that can be sent to an instance of a class
- Class methods: the procedures that implement message that can be sent to a class.

5.6 Language Classes

⇒ First Generation

- The first language generation represents machine code and its more human-readable equivalent-assembly language.
- ***MACHINE LANGUAGE***: is basically the only language that the computer directly understands. Often functions as the object language of higher-level language programs, since all high-level languages must be translated into machine language in order for the computer to execute them. This coding form is also often in octal or hexadecimal codes, since it is extremely tedious to code in 0s and 1s.
  - Advantage: most efficient in terms of storage area use and execution speed. Allows programmer to utilize the computer's potential for processing data.
  - Disadvantage: extremely difficult to program, remember and use.
- ***ASSEMBLY LANGUAGE:*** programmer uses symbolic names, or mnemonics, to specify machine codes. Mnemonics are English-like abbreviations for the machine-language opcodes.
  - Advantage: can be used to develop programs highly efficient in terms of storage space use and processing time.
  - Disadvantage: cumbersome to use, as one assembly-language instruction is translated into one machine-language instruction. Also difficult to program effectively. Also machine-dependent, i.e. programs written on one computer generally cannot work on another.

5.6.1 Second Generation Languages

⇒ These languages were developed in the late 1950's and the early 1960's, and served as the foundation for all third-generation languages.

⇒ Second generation languages are characterized by:

- Broad usage
- Enormous software libraries
- Widest familiarity and acceptance.

⇒ Some examples of these are FORTRAN (30 years old), COBOL and BASIC.

5.6.2 Third Generation Languages

⇒ Also called modern or structured programming languages. There are three broad categories of these languages:

- General-Purpose high order languages
- Object-Oriented high order languages
- Specialized languages

⇒ General Purpose High-Order Languages

- Languages used for general programming purposes, e.g. software products, embedded applications and systems software.
- Examples: PASCAL, ALGOL, ADA and C.

⇒ Object Oriented Languages

- Object-oriented programming languages enable a software engineer to implement analysis and design models created using Object-oriented analysis and object-oriented design.
- Examples: dialects of C, i.e. C++, and Smalltalk.

⇒ Specialized languages

- Characterized by unusual syntactic forms that have been especially designed for a distinct applications.
- Examples: LISP, PROLOG, APL and FORTH.

5.6.3 Fourth Generation Languages

⇒ 4GL can be said to combine procedure and non-procedure languages

⇒ In other words, the language enables the user to specify conditions and corresponding actions (the procedural component) while at the same time encouraging the user to indicate the desired outcome (the nonprocedural component) and then applying its domain-specific knowledge to fill in the procedural details.

⇒ 4GL can be divided into several categories:

- Query Languages
- Program Generators
- Other Categories of 4GLs

5.6.4 Query Languages

⇒ Vast majority of 4GLs have been developed for use in conjunction with database applications.

5.6.5 Program Generators

⇒ Program generators enable the user to create complete third-generation language programs. However, most program generators today focus extensively on business information systems applications and generate programs in COBOL.

5.6.6 Other Categories of 4GLs

⇒ Some of these other categories are Prototyping languages have been developed to assist in the creation of prototypes and a means for data modeling. Formal specification languages can also be considered as a 4GL when such languages produce machine-executable software.

Chapter review questions

1) highlight the benefits of using 4GLs for implementation
2) what are the factors to consider when selecting a good programming language for use

References for Further Reading

1. Pressman R.S (1997)software engineering: a practitioner's Approach, McGraw Hill
2. Somerville Ian (2002) software engineering ,Pearson's education

CHAPTER 6: DATA FLOW ORIENTED DESIGN

Chapter Objectives

At the end of this chapter, student learn :

⇒ Design Process Considerations

⇒ Transform Flow and Transaction Flow

- Transform Flow
- Transaction Flow
- Process Abstract

⇒ Transaction Analysis

⇒ Design Heuristics

⇒ Design Post processing

⇒ Data Flow-oriented Design

- The design phase in software development is the multi-step process in which representations of data structure, program structure, and procedure are synthesized from information requirements.
- The objective of Data Flow-Oriented methods is then to provide a systematic approach for the derivation of program structure.

6.1 Design Process Considerations

⇒ Data Flow-Oriented design allows a convenient translation from information requirements (e.g. the data flow diagram) contained in a Software Requirements Specification to a design description of program structure.

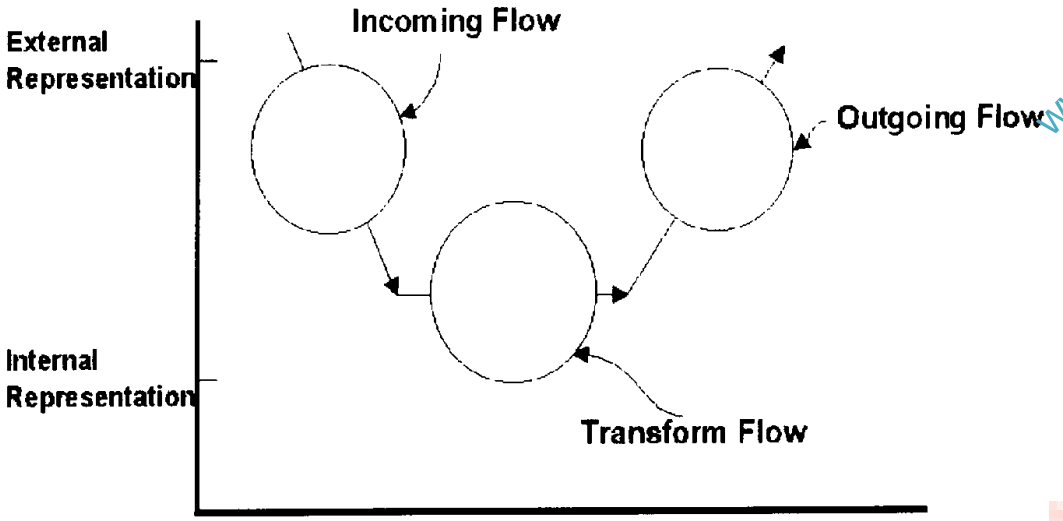
⇒ Five-Step Process

- Step 1: The type of information flow is established
- Step 2: Flow boundaries are indicated.
- Step 3: The DFD is mapped into program structure
- Step 4: The control hierarchy is defined by factoring
- Step 5: The resultant structure is refined using design measures and heuristics.

6.2 Transform Flow and Transaction Flow

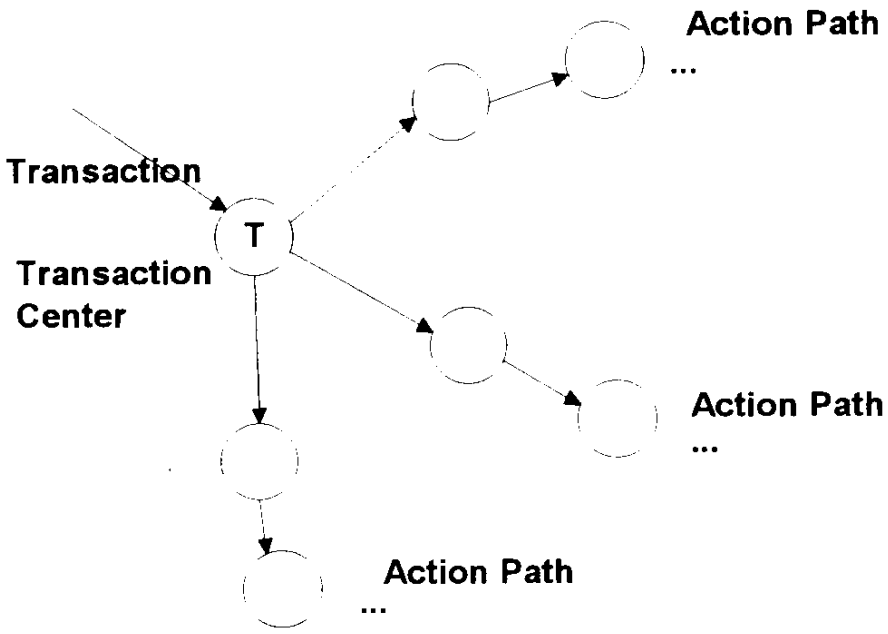
⇒ Transform Flow

- Data coming from external forms must be converted into an internal form for processing.
- Information entering the system along paths that transform external data into an internal form are identified as incoming flow.
- At the kernel of the software, a transition occurs: Incoming data are passed through a transform centre and begin to move along paths that lead out of the software.
- Data moving along these paths are called outgoing flow.



⇒ Transaction Flow

- Information flow is characterized by a single data item, called a transaction, that triggers other data flow along one of many paths.
- The transaction is evaluated and, based on its value, flow along one of many action paths is initiated.
- The hub of information flow from which many action paths flow from is called a transaction centre.



6.2.1 A Process Abstract

⇒ Design begins with an evaluation of the level 2 or level 3 DFD

⇒ The information flow category is established

⇒ Flow boundaries that delineate the transform or transaction centre are defined

⇒ Based on the location of boundaries, transforms are mapped into program structure as modules

⇒ The precise mapping and definition of modules is accomplished by distributing control top-down in the structure

6.3 Transform Analysis

⇒ Transform analysis refers to a set of design steps that will allow a DFD with transform flow characteristics to be mapped into a predefined template for program structure.

Design Steps:

- Step 1: Review the fundamental system model (refer to Figure 6.0 and 6.1)
- Step 2: Review and refine DFD for the software (refer to Figure 6.1 and 6.2)
- Step 3: Determine whether the DFD has transform or transaction flow characteristics (refer to Figure 6-4)
- Step 4: Isolated the transform centre by specifying incoming and outgoing flow boundaries (refer to Figure 6.4)
- Step 5: Perform "First-Level Factoring" (refer to Figure 6.5 and 6.6"
- Step 6: Perform "Second-Level Factoring" (refer to Figure 6.7, 6.8 and 6.9)
- Step 7: Refine the "First-Cut" program structure using design heuristics for improved software quality (refer to Figure 6.10)

⇒ Review the Fundamental System model

- The Context diagram, also known as the level 0 DFD, the Systems Specification, and the Software Requirements Specification are reviewed.

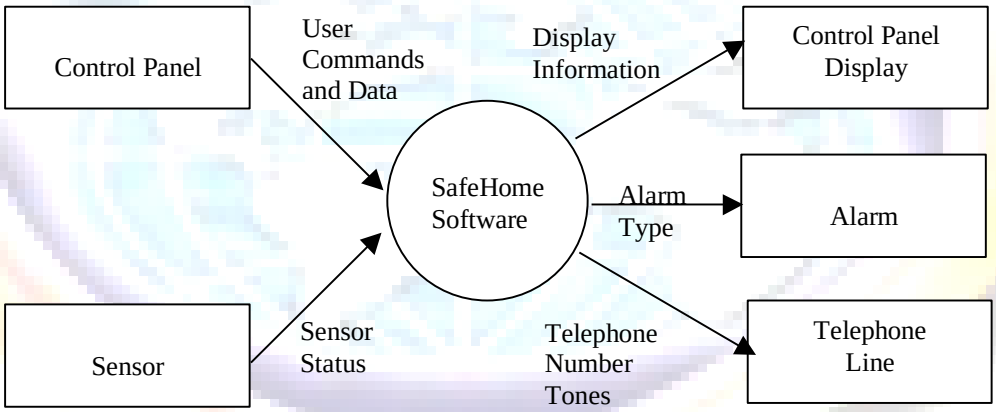


Figure 6-0 : Context Level DFD for SafeHome

⇒ Review and Refine Data Flow Diagrams for the Software

- From the Systems Specification and the Software Requirements Specification, information is obtained from the analysis models, and refined to greater detail.

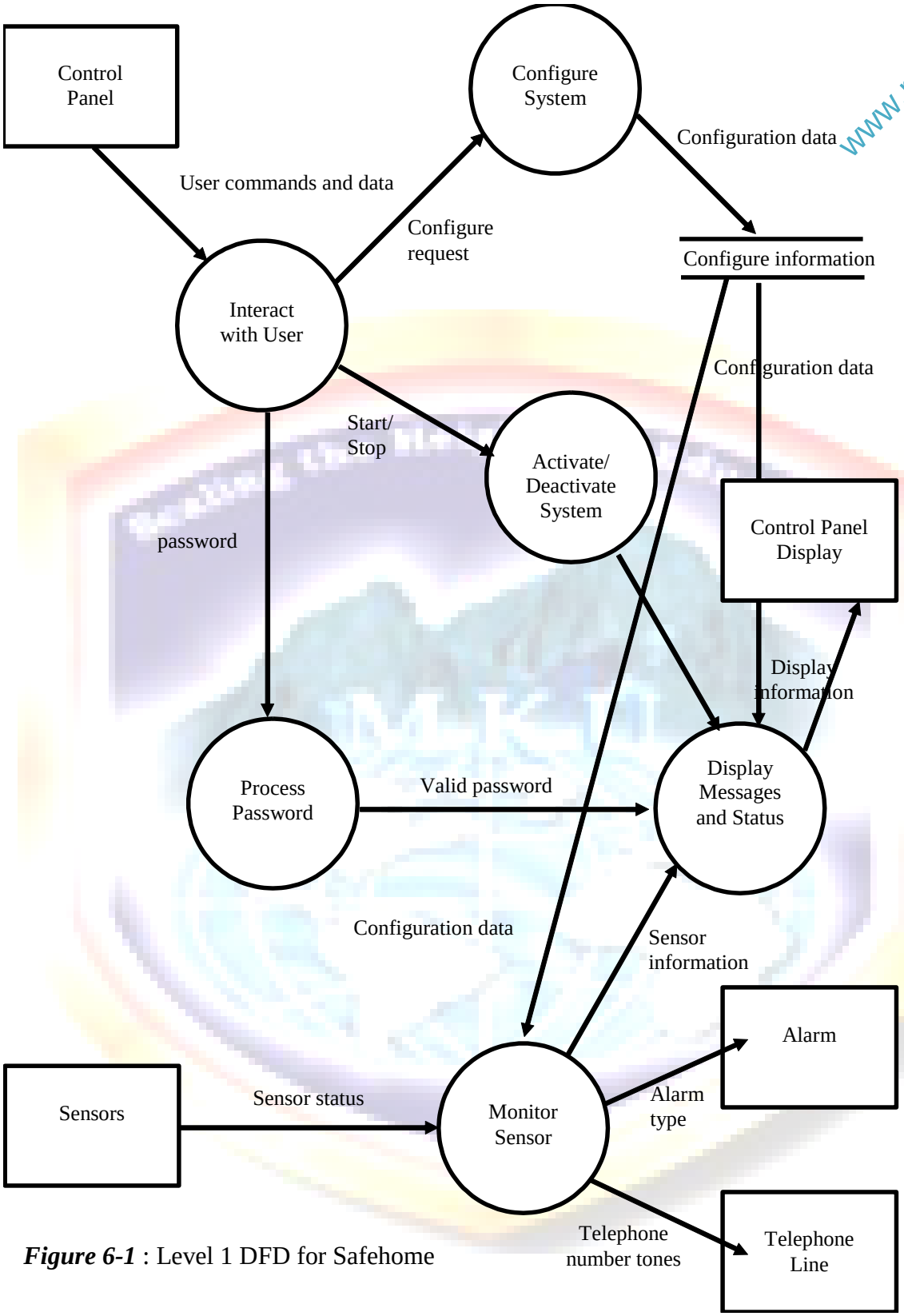


Figure 6-1 : Level 1 DFD for Safehome

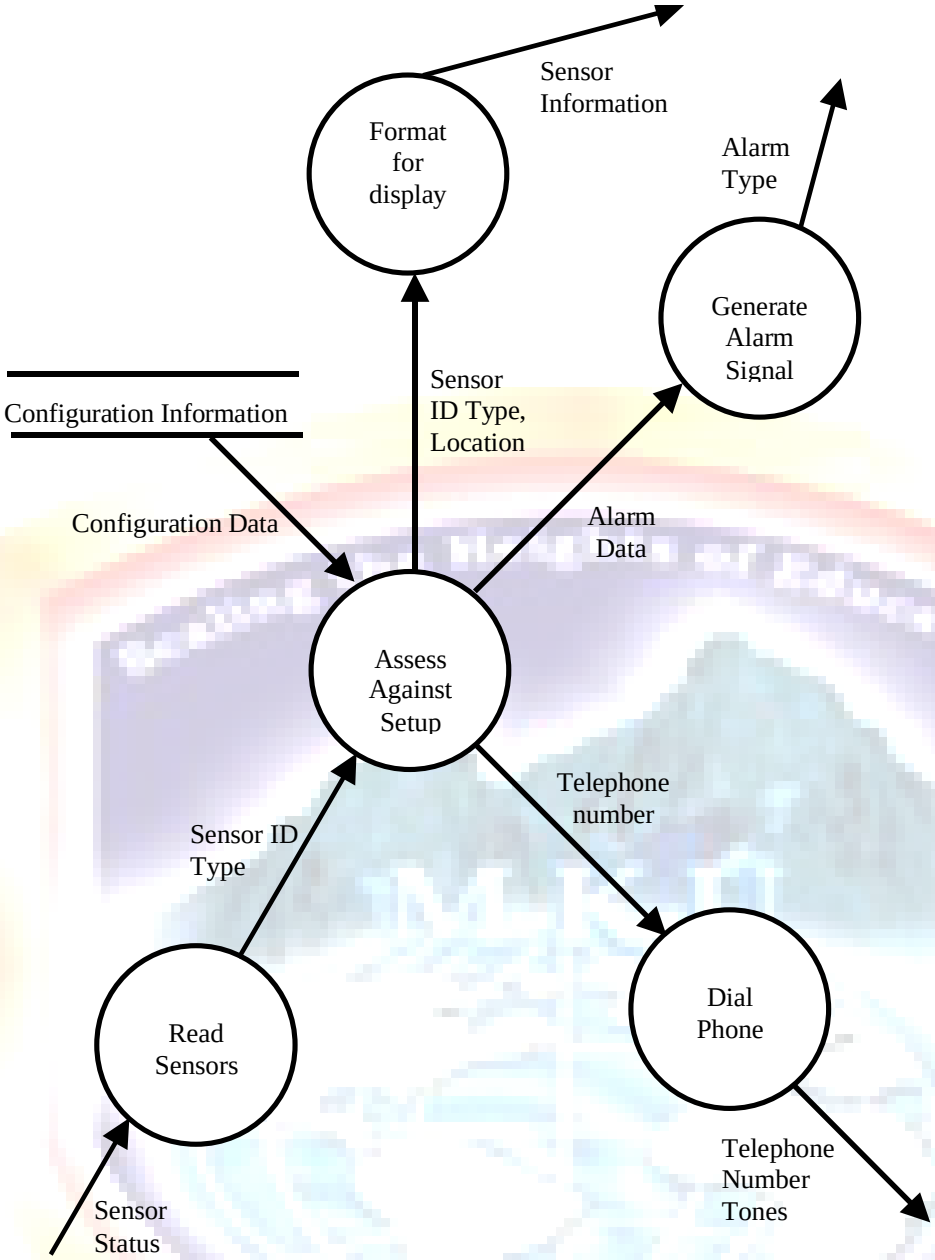


Figure 6-2 : Level 2 DFD That Refines the Monitors Sensors Process

⇒ Determine whether the DFD has transform or transaction flow characteristics

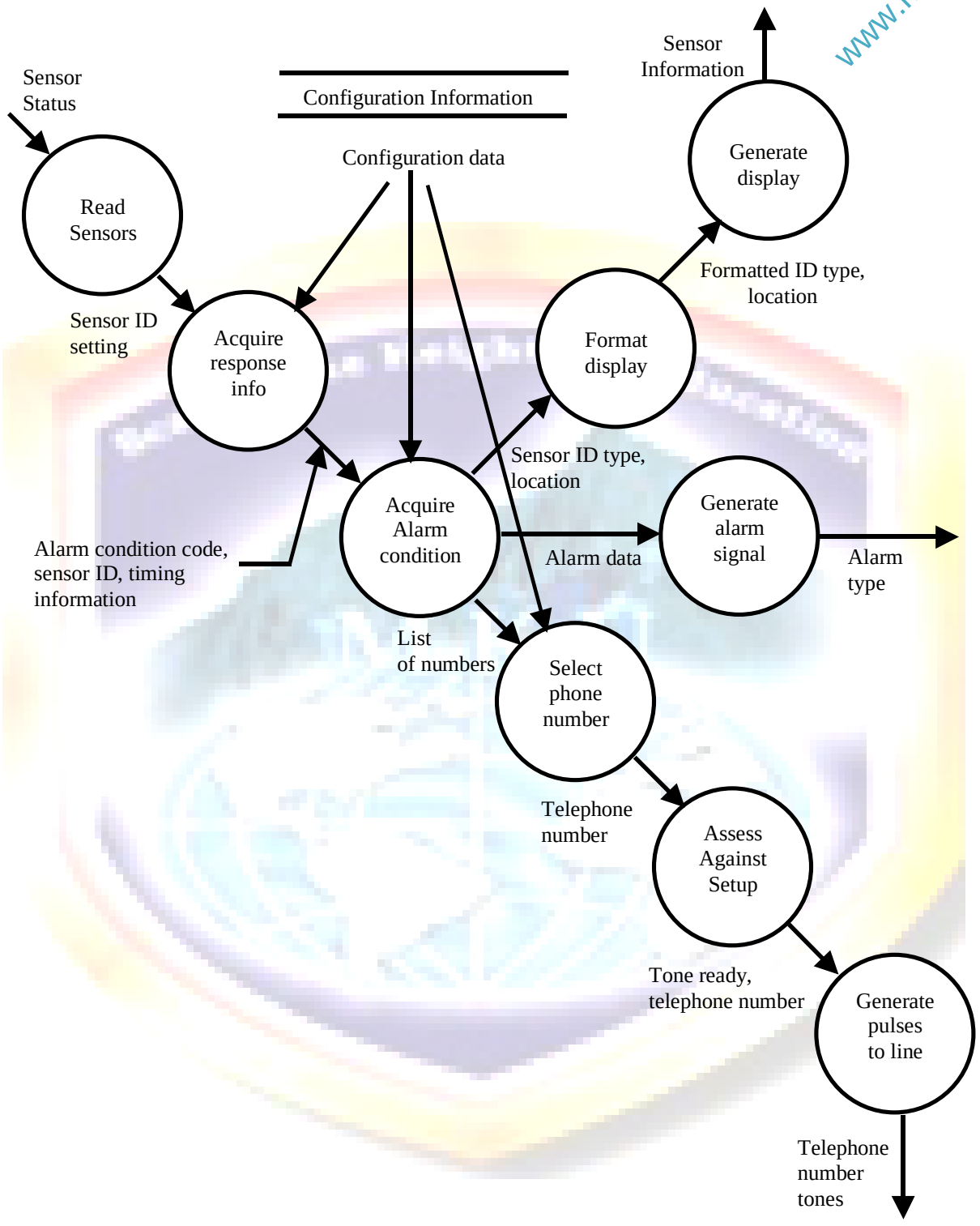


Figure 6-3 : Level 3 DFD that REFines the Monitor Sensor Process

⇒ Isolate the transform centre by specifying incoming and outgoing flow boundaries

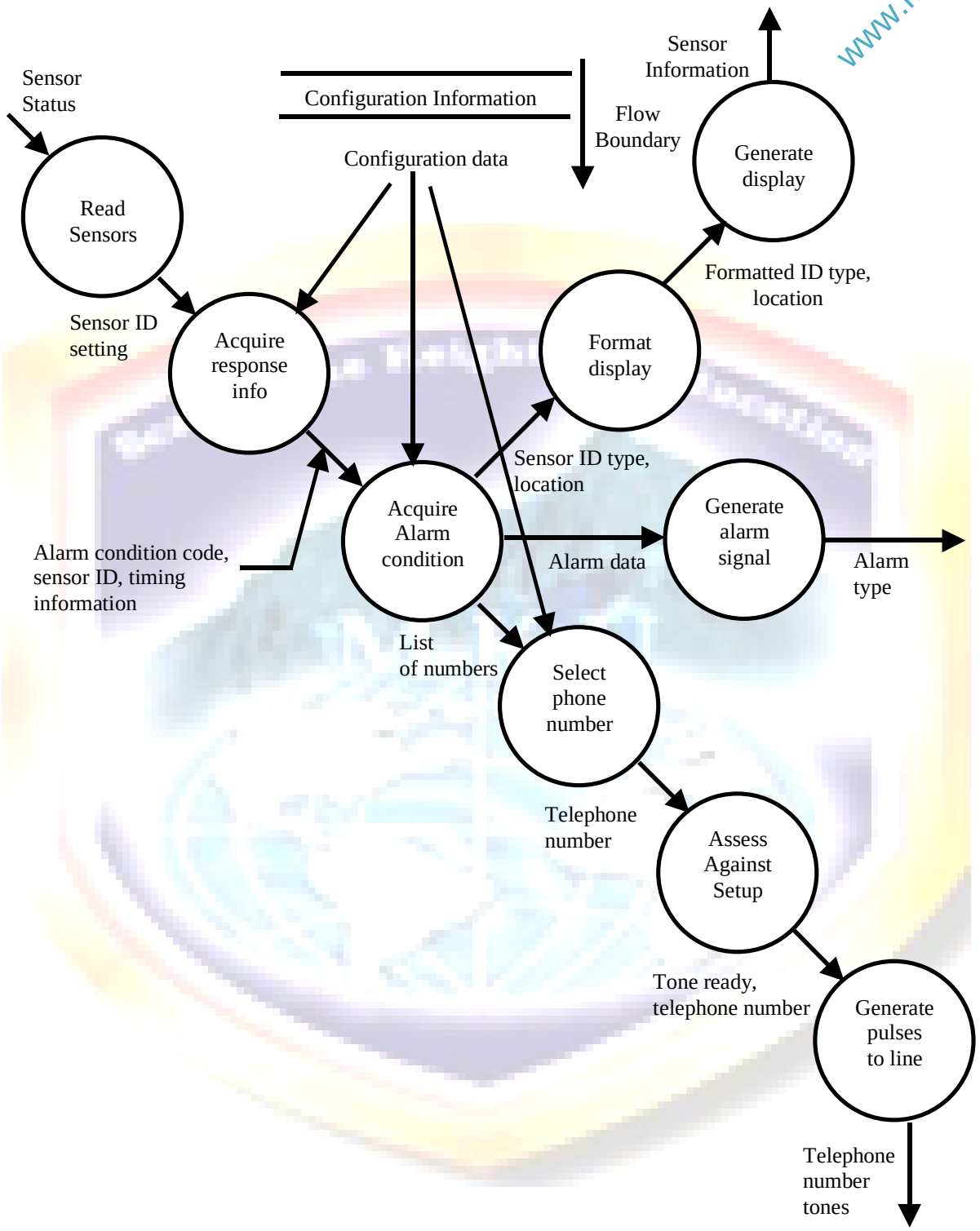


Figure 6-4 : Specifying Flow Boundaries

⇒ Perform "First-Level factoring"

- First level factoring: mapping of transform flow to a specific structure that provides control for incoming transform, and outgoing information processing

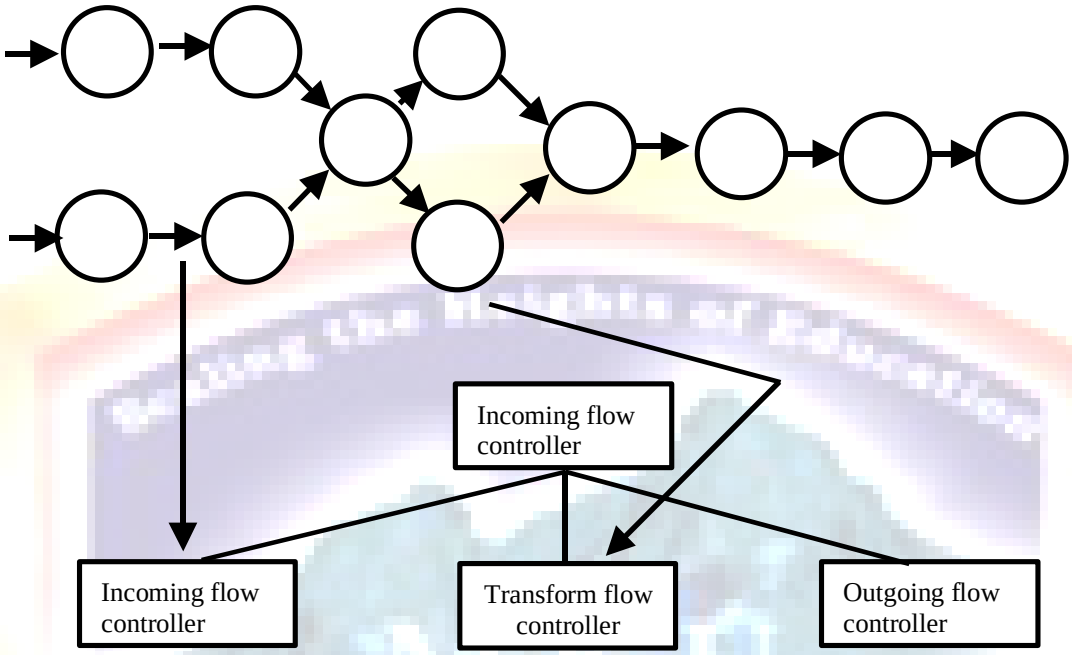


Figure 6-5 : First-level Factoring

⇒ Incoming Information Flow Controller: co-ordinates the receipt of all incoming data

⇒ Transform Flow Controller: supervises all operations on data in ternalised form

⇒ Outgoing Information Flow Controller: co-ordinates the production of output information

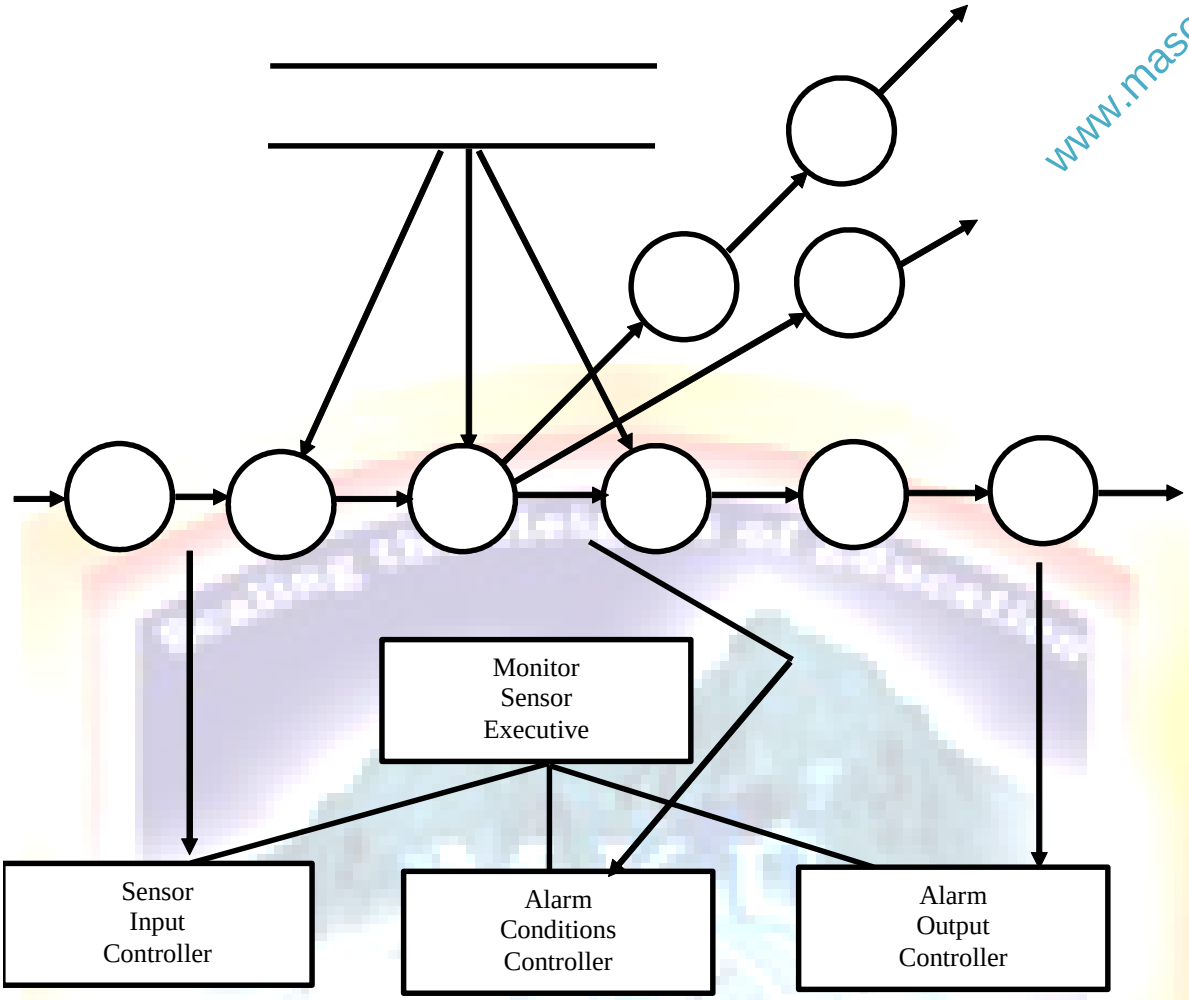


Figure 6-6 : First-level Factoring for Monitor Sensors

⇒ Perform "Second-Level factoring"

- Accomplished by mapping individual transform (bubbles) of a DFD into appropriate modules within the program structure

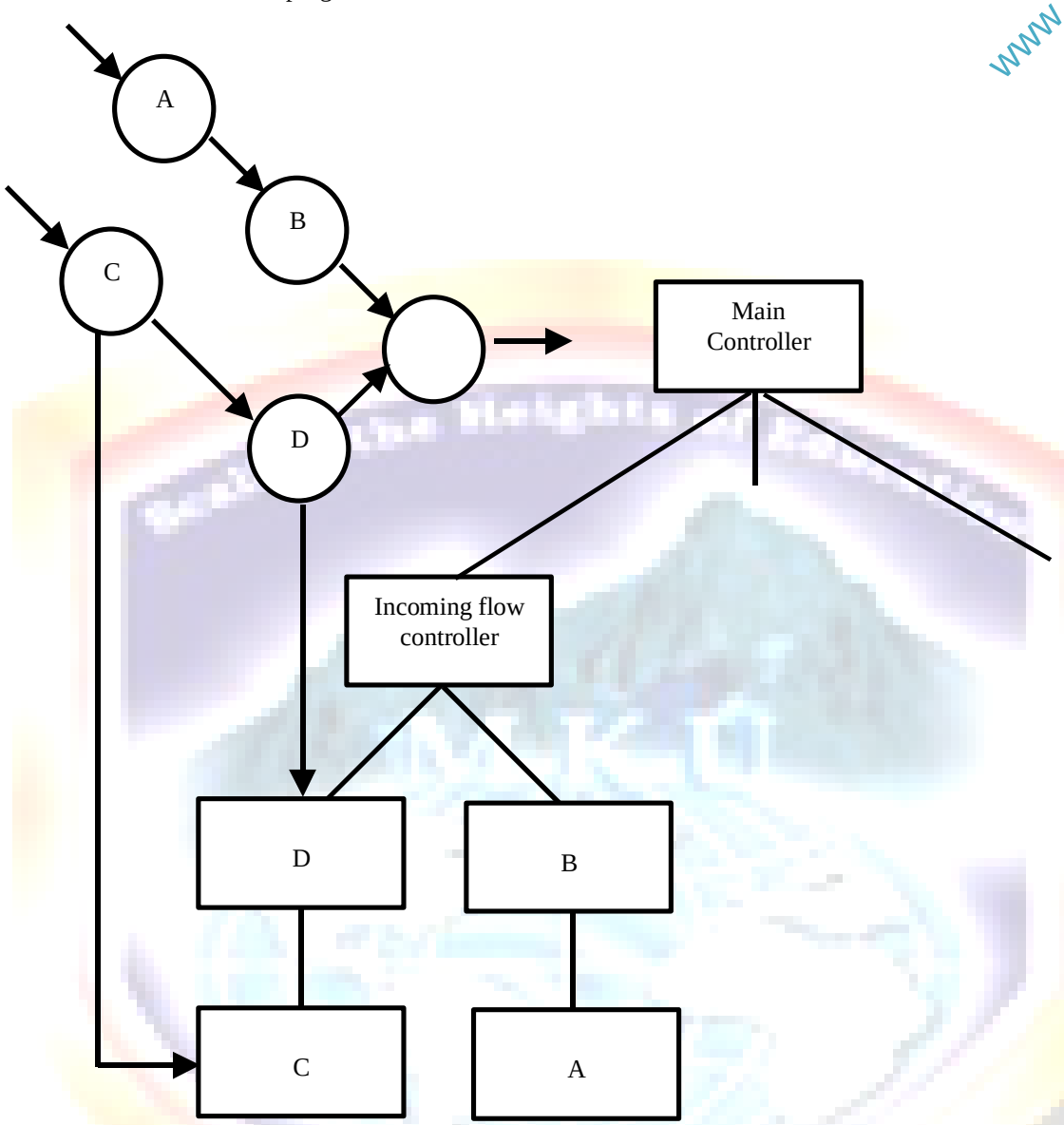


Figure 6-7 : Second-level Factoring

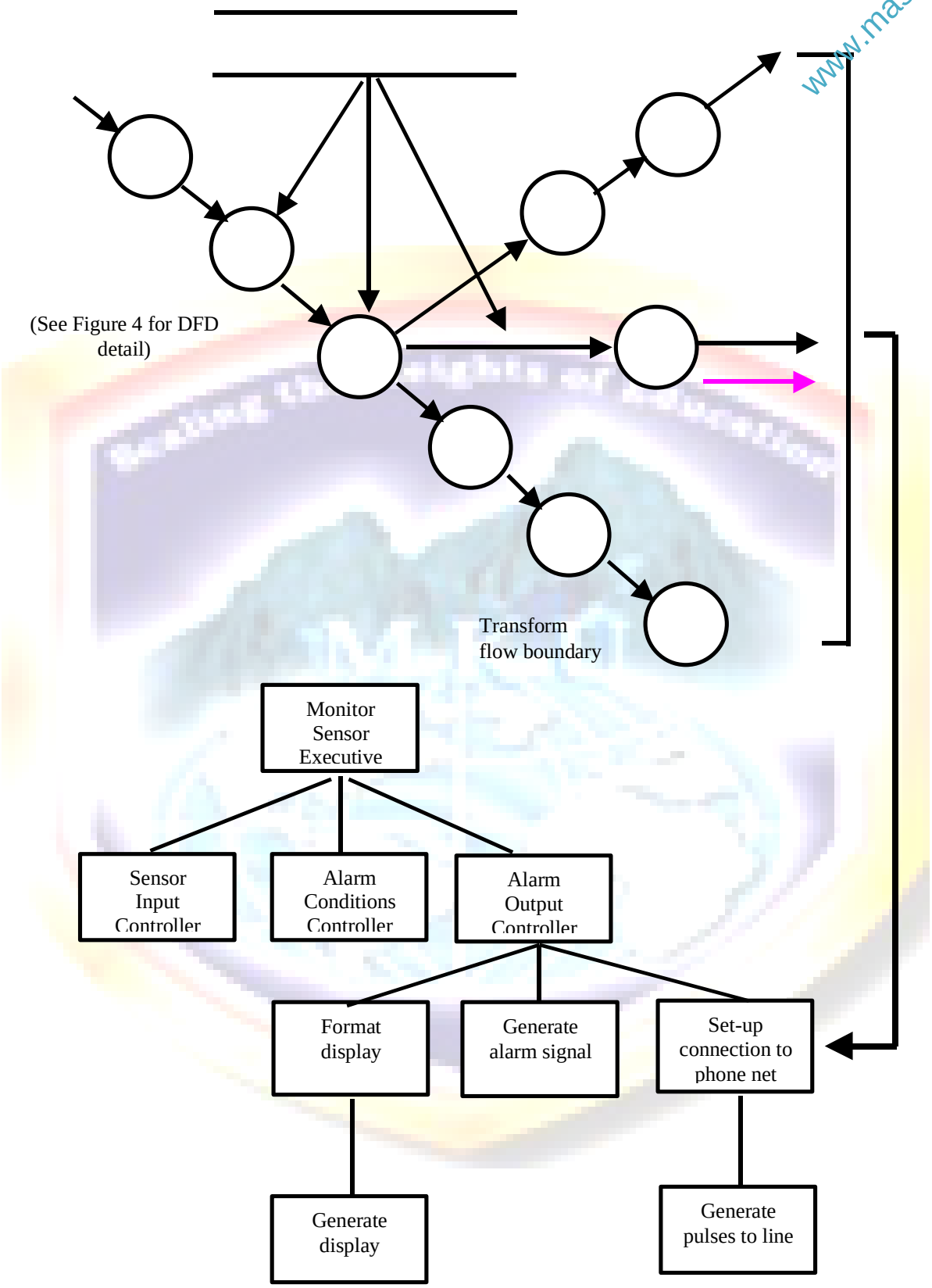


Figure 6-8 : Second-level Factoring for Monitor Sensors

⇒ Refine the "First-Cut" program structure using design heuristics for improved software quality

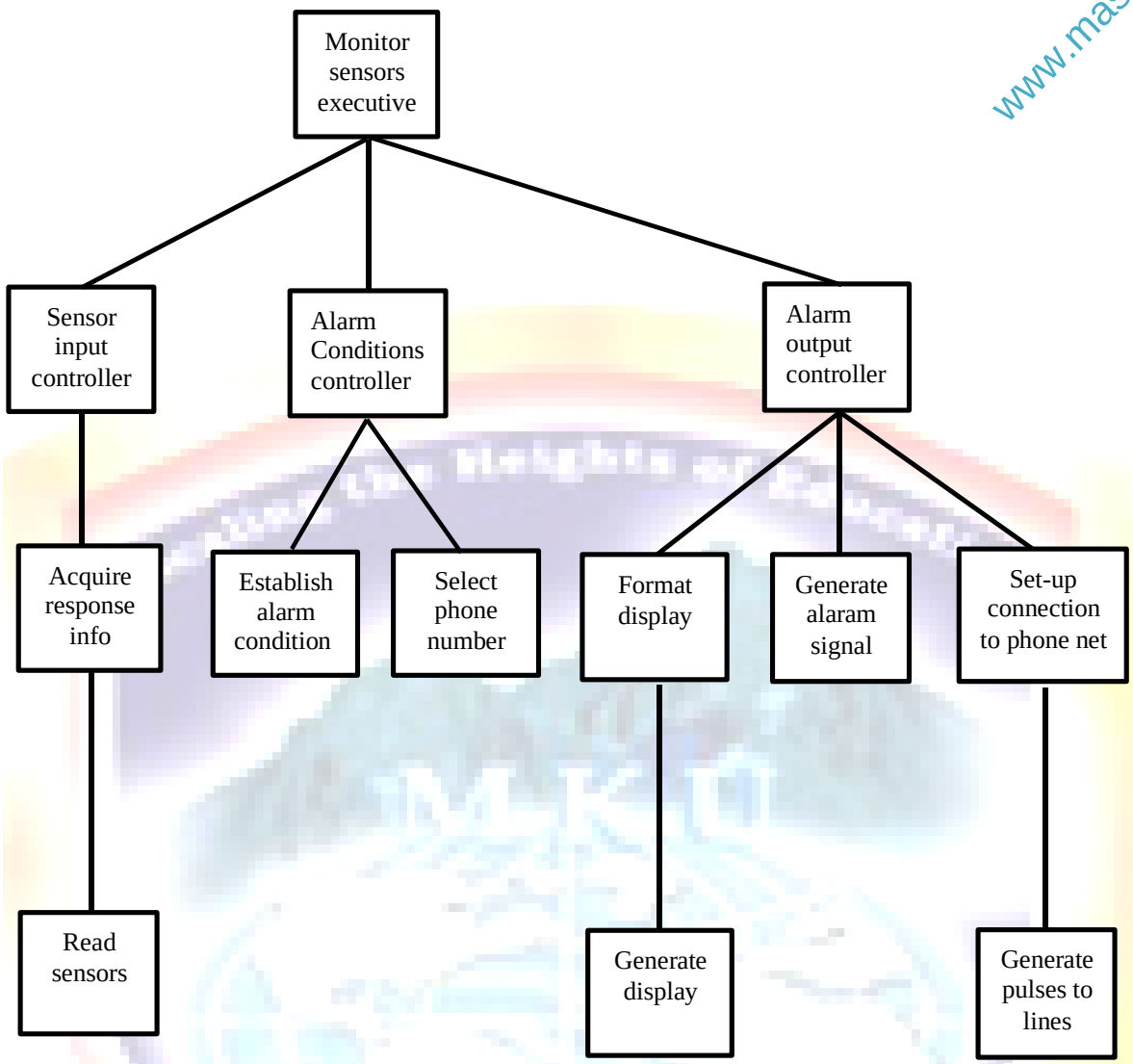


Figure 6-9 : "First cut" Program Structure (Structure Chart)

⇒ Apply the concepts of modules of modules independence by expanding or contracting modules to produce sensible factoring, good cohesion, minimal coupling,.i.e.. we want a structure that can be implemented within practical limits, tested and maintained without too much difficulty.

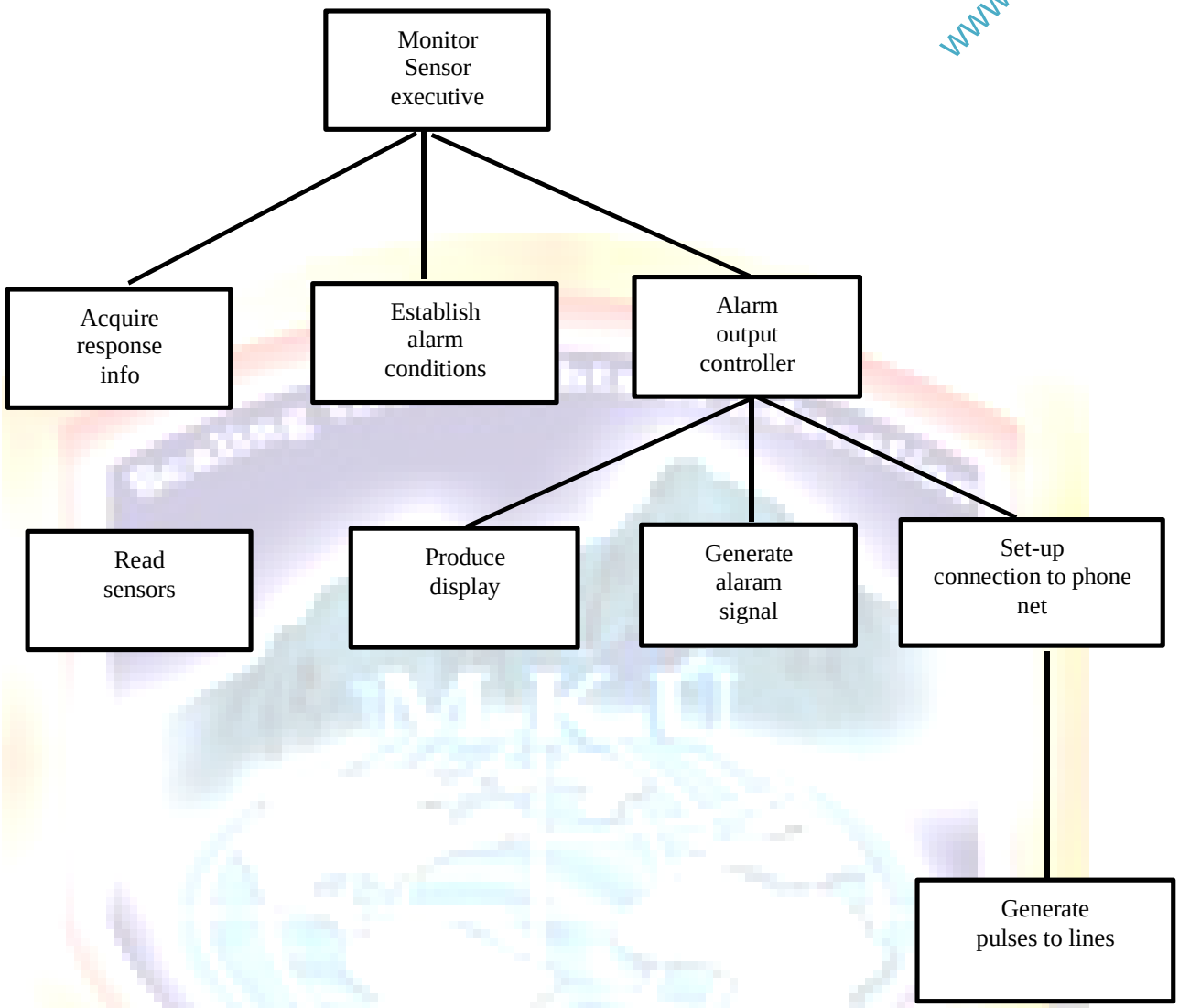


Figure 6-10 : Refined Program Structure for Monitor Sensors

6.4 Transaction Analysis

⇒ Transaction analysis refers to a set of design steps that will allow a DFD with transaction flow characteristics to be mapped into a predefined template for program structure.

Design Steps:

⇒ Step 1: Review the fundamental system model

⇒ Step 2: Review and refine DFD for the software

⇒ Step 3: Determine whether the DFD has transform or transaction flow characteristics (refer to Figure 6.11)

⇒ Step 4: Identify the Transaction center and the flow of characteristics along each of the action paths (refer to Figure 6.12)

⇒ Step 5: Map the DFD in a program structure amendable to transaction processing (refer to Figure 6.13)

⇒ Step 6: Factor and refine the transaction structure of each action path (refer to Figure 6.15)

⇒ Step 7: Refine the "First-Cut" program structure using design heuristic for improved software quality

⇒ Review the Fundamental System Model

⇒ Review and Refine Data Flow Diagrams for the software

⇒ Determine whether the DFD has transform or transaction flow characteristics.

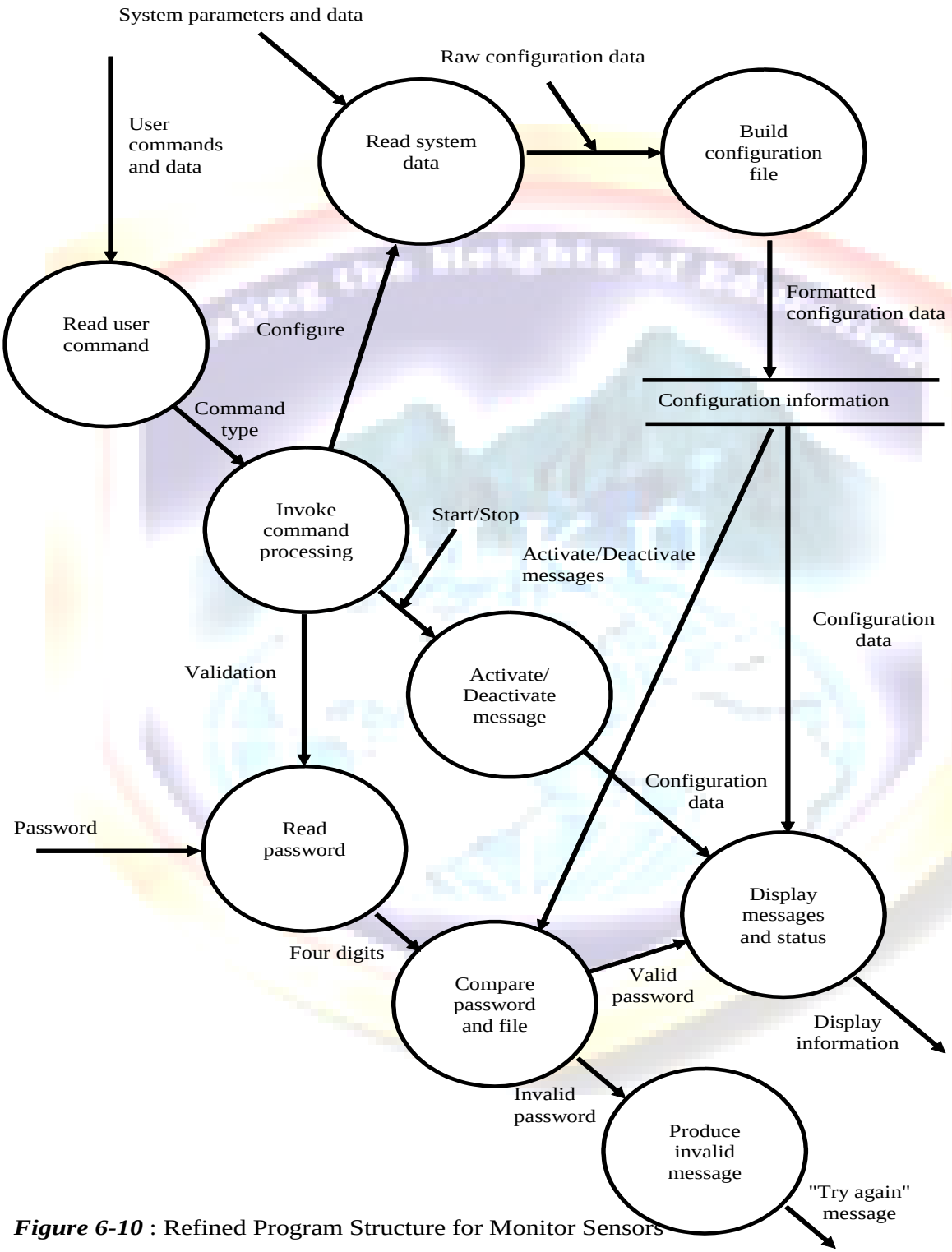


Figure 6-10 : Refined Program Structure for Monitor Sensors

⇒ Identify the transaction centre and the flow characteristics along each of the action paths.

- Transaction centre: the ©bubble[a] from where all the action paths flow from

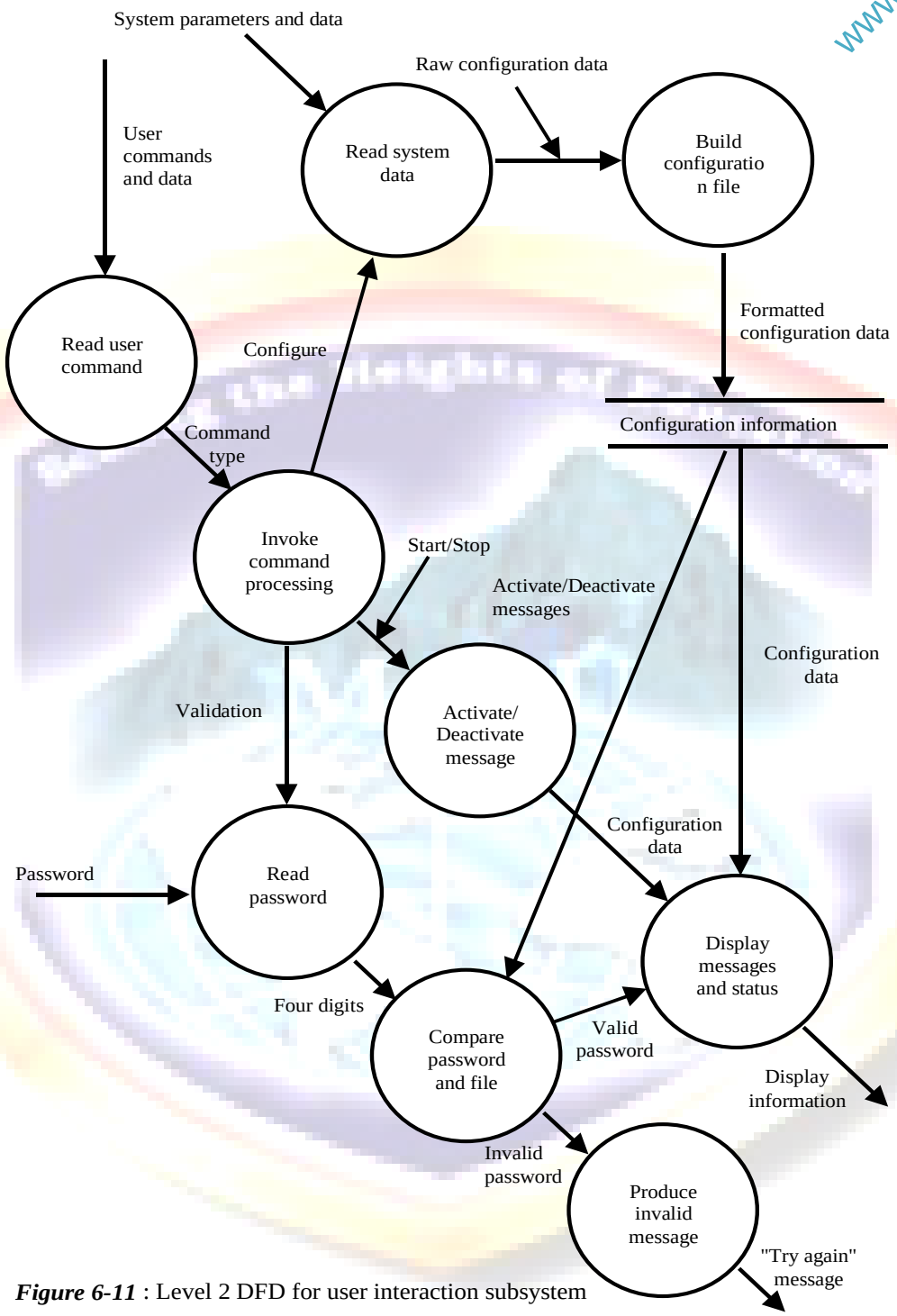


Figure 6-11 : Level 2 DFD for user interaction subsystem

⇒ The structure of the dispatch branch contains a dispatcher module that controls all subordinate action path controllers

⇒ Each action path of the DFD is mapped to a structure that corresponds to its specific flow characteristics

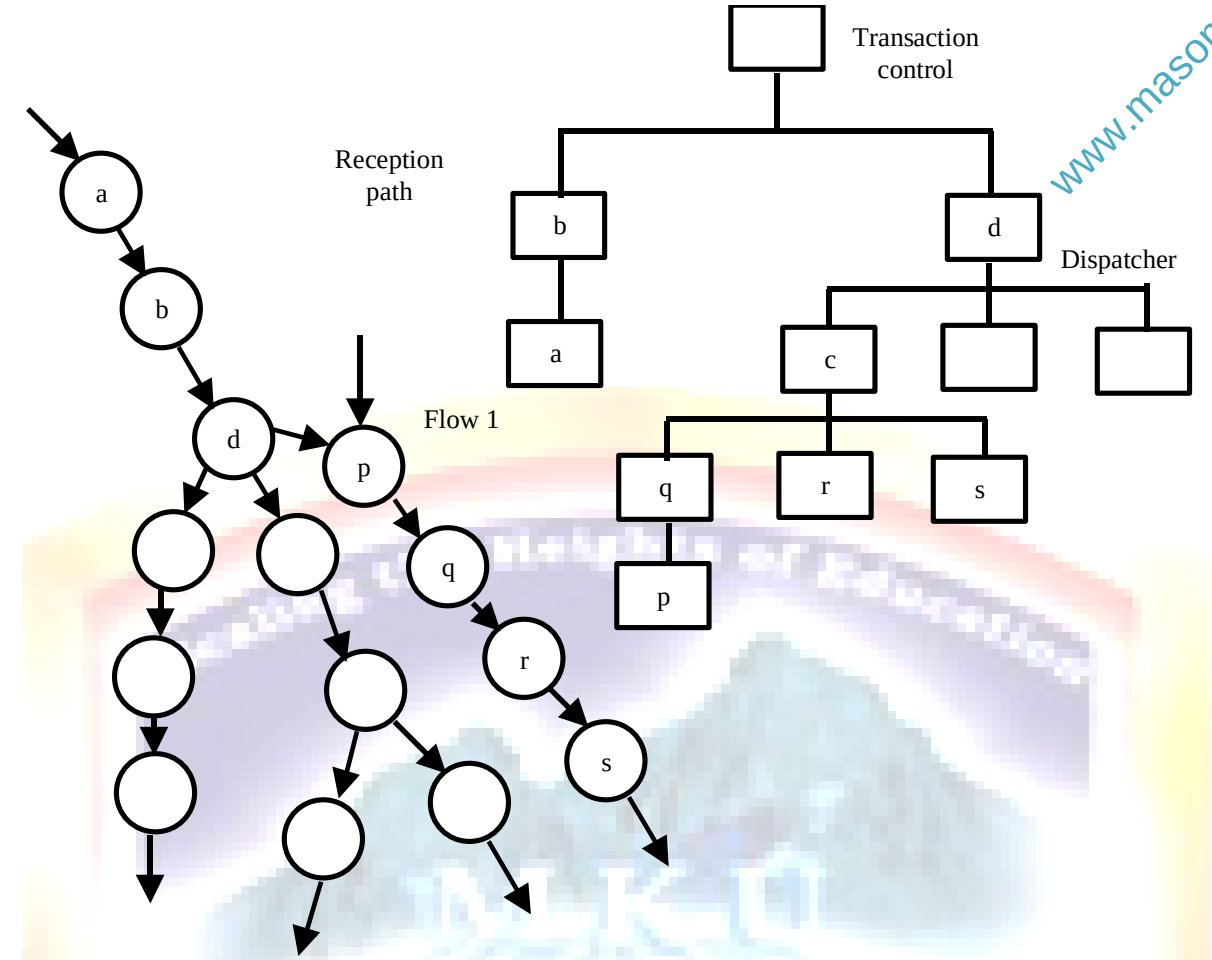


Figure 6-12 : Transaction Mapping

⇒ Factor and refine the transaction structure and the structure of each action path.

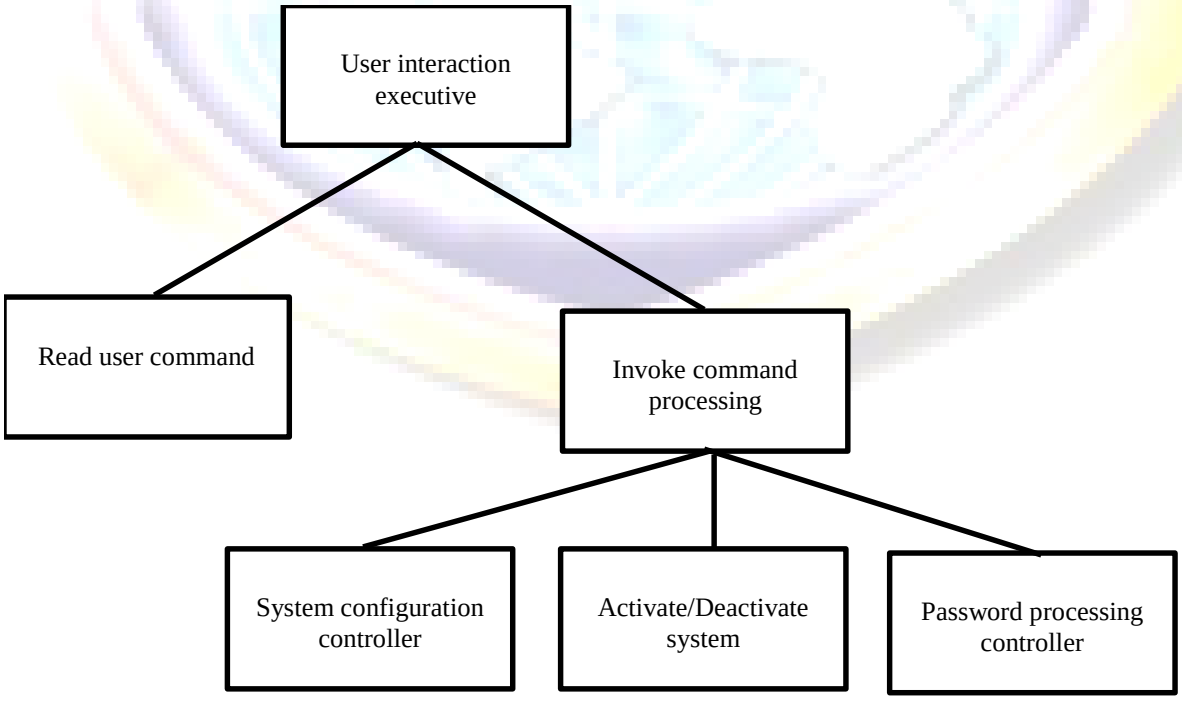


Figure 6-13 : First-level Factoring for User Interaction Subsystem

⇒ Refine the ©First-Cut[a] program structure using design heuristics for improved software quality.

- The last two steps are similar to transform analysis. In this design approach, criteria such as module independence, practicality, and maintainability must again be considered.

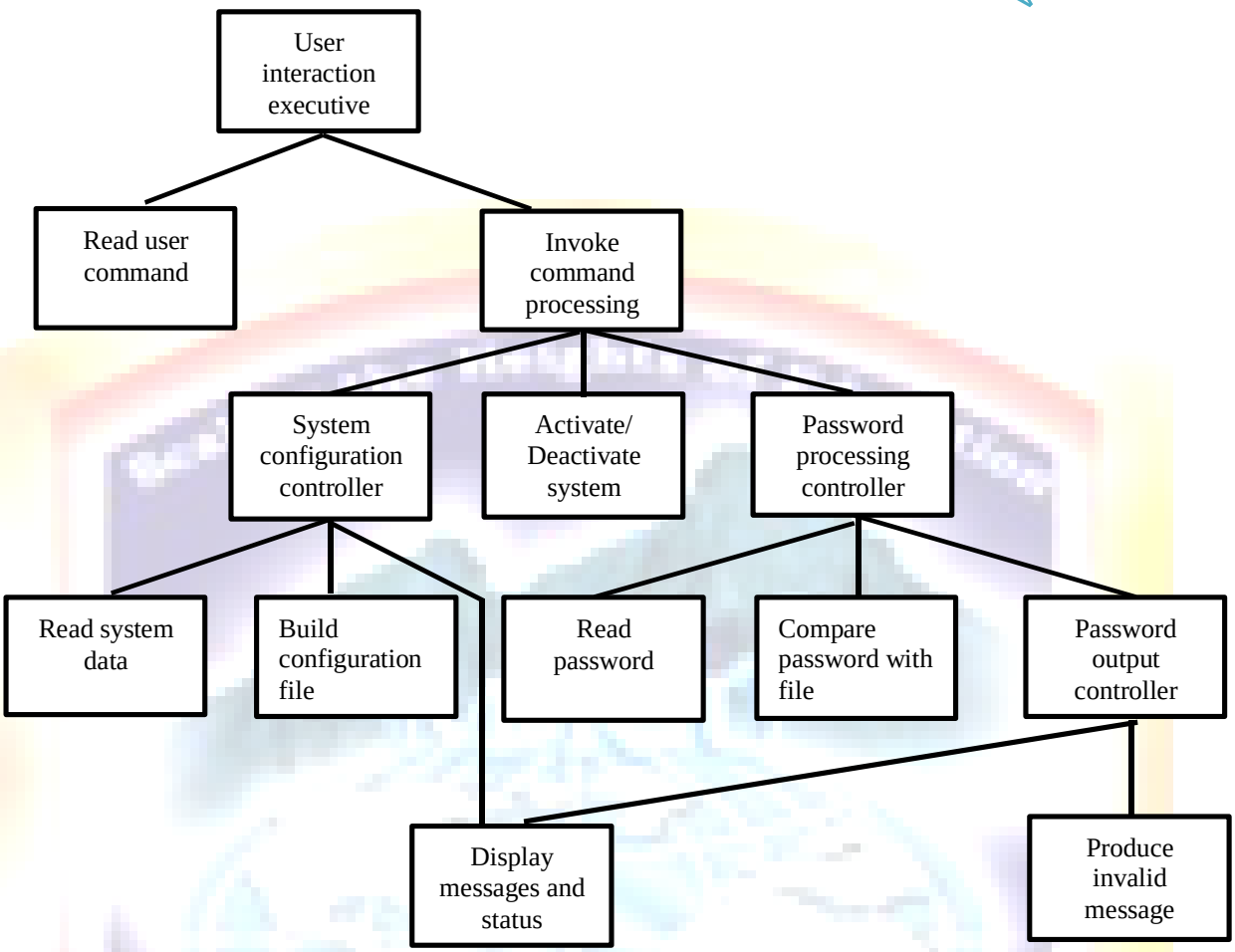


Figure 6-14 : "First-cut" Program Structure for User Interaction Subsystem

6.5 Design Heuristics

⇒ Guidelines

- Evaluate the "First-Cut" program structure to reduce coupling and improve cohesion
- Attempt to minimize structures with high fan-out; strive for fan-in as depth increase
- Keep scope of effect of a module within the scope of control of that module
- Evaluate module interfaces to reduce complexity and redundancy and improve consistency
- Define modules whose function is predictable, but avoid modules that are overly restrictive
- Strive for single-entry-single-exit modules, avoiding "pathological connections"
- Package software based on design constraints and probability requirements

6.6 Design Post processing

⇒ Approach for Time-critical Software

⇒ Develop and refine the program structure without concern for time-critical optimization

⇒ Use CASE tools that simulate run-time performance to isolate areas of inefficiency

⇒ During detail design, select modules that are suspected "time hogs" and carefully develop procedures (algorithms) for time efficiency

⇒ Code in high-order programming language

⇒ Instrument the software to isolate modules that account for heavy processor utilization

⇒ If necessary, redesign or recode in machine dependent language to improve efficiency

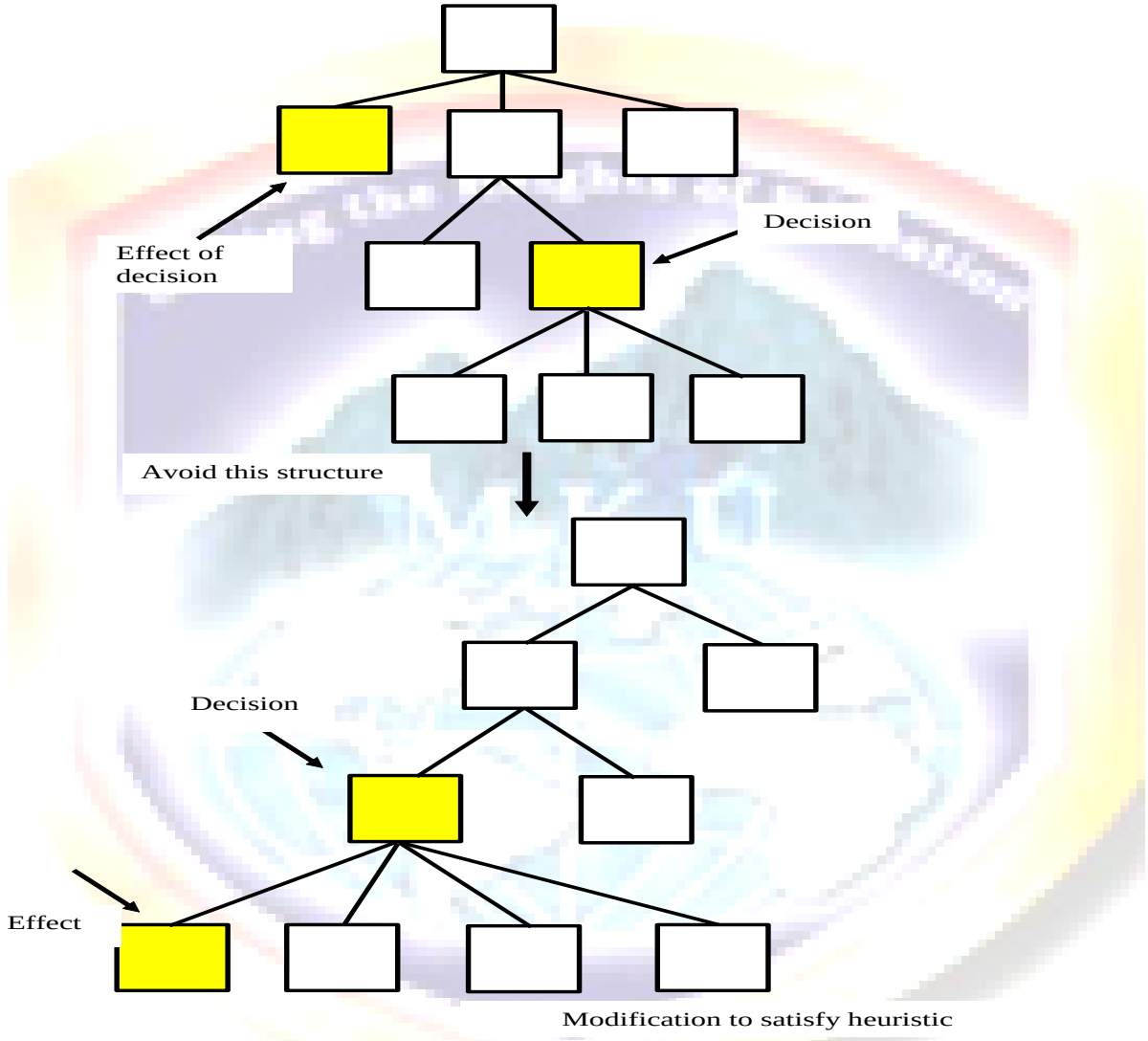


Figure 6-16 : Scope of Effect and Control

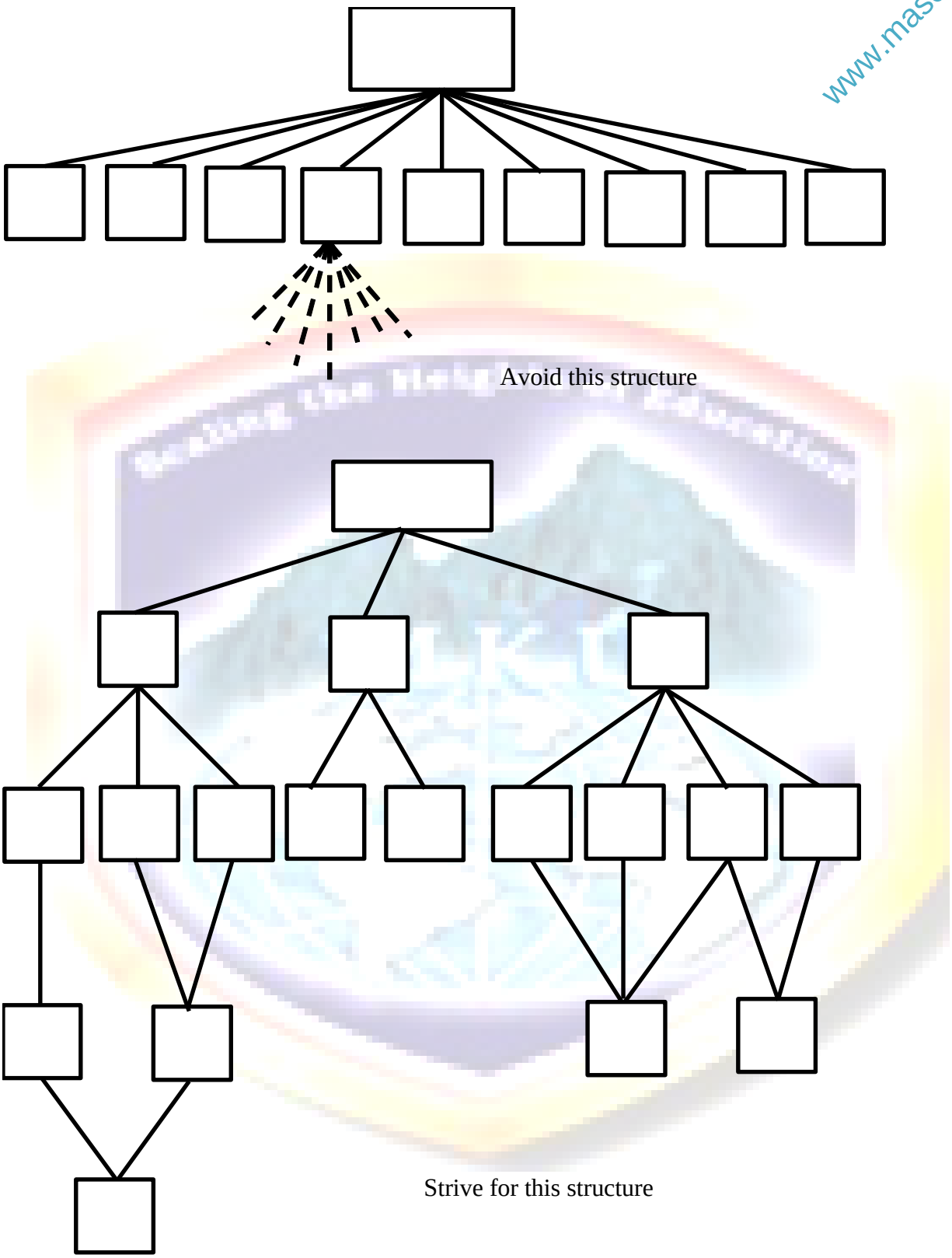


Figure 6-15 : Fan-in and Fan-out

Chapter Review Questions

1. Clearly distinguish between process oriented design and data oriented design, giving valid examples
2. Draw a context model for a patient management information system in a hospital
3. Draw a data flow diagram for a customer withdrawing cash from an ATM in a bank.

References for Further Reading

1. Pressman R.S (1997) Software engineering: a practitioner's Approach, McGraw Hill
1) Somerville Ian (2002) software engineering ,Pearson's education

CHAPTER 7: DATA-ORIENTED DESIGN METHODS

Chapter Objectives

At the end of this chapter, stdent should understand::

⇒ Data Oriented Design Methods

- Introduction
- Examples
- Areas of Application

⇒ Jackson Structured Programming (JSP)/Jackson System Development (JSD)

⇒ Correspondence between Data Structures

⇒ Listing the Elementary Program Operations

7.1 Data Oriented Design Methods

⇒ Focuses on the information domain.

⇒ Uses information structure as the driver for derivation of design.

⇒ Transforms a representation of data structure into a representation of software.

⇒ Examples of Data Oriented Design Methods: Jackson's System Development, Data Structure Systems Development.

7.2 Areas of Application

⇒ Applied in applications with well-defined, hierarchical structure of information

⇒ Examples are:

- Business information system applications which have distinct structure (e.g. input files, output records)
- Systems applications. The data structure of operating system which comprised of many tables, files and lists that have a well-defined structure
- CAD/CAE/CAM applications. Computer-aided design, engineering, and manufacturing systems require sophisticated data structures for information storage, translation and processing

7.3 Jackson Structured Programming (JSP)/Jackson System Development (JSD)

⇒ Takes the view of "paralleling the structure of input data and output (report) data will ensure a quality design"

⇒ More recent extensions to the methodology is the Jackson System Development (JSD)

⇒ JSD focuses on the identification of information entities and the actions that applied to them

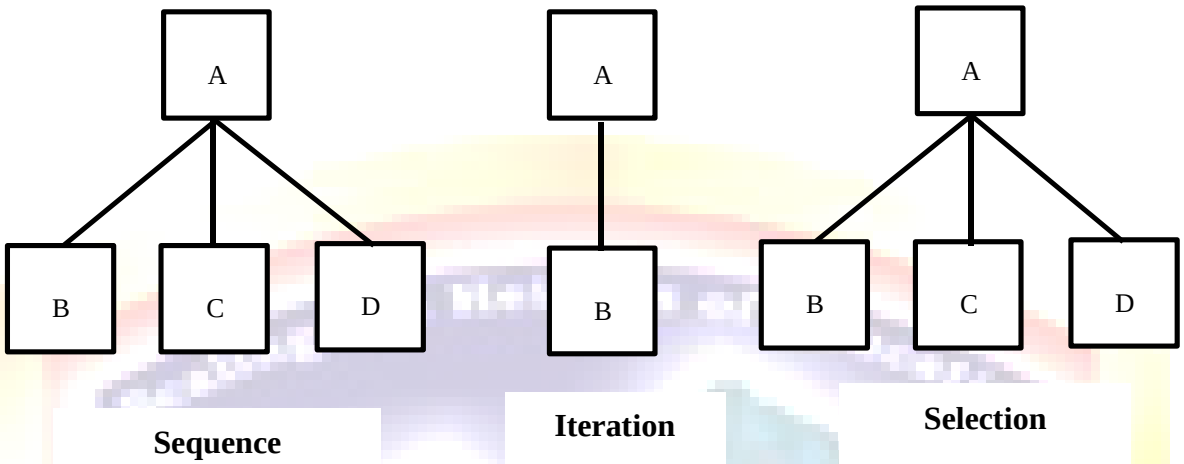
⇒ JSD emphasizes on developing techniques to transform data program structure

7.3.1 Jackson Structured Programming

⇒ Data driven program design method

⇒ Produce data structure diagrams for input and output data streams

⇒ Device independent and model as tree diagram using sequence, selection and iteration



A is a sequence of B followed by C followed by D
A consists of zero or more occurences of B
A is a selection of either B or C or D

7.4 Characteristics of JSP

⇒ It is non-inspirational. This means that it depends little or not at all, on invention or insight on the part of the engineer.

⇒ It is rational, i.e. the design procedure is based on reasoned principles, and each step can be proven in the light of these principles.

⇒ It is teachable. People can be taught to practice the method and two or more programmers using the method to solve the same problem will arrive at substantially the same solution.

⇒ It is practical. The method itself is simple and easy to understand, and the designs produced can be implemented without difficulty in any ordinary programming environment.

7.5 Advantages of JSP

⇒ Enable correct programs to be produced

⇒ Provide a method that is "workable" within the intellectual limitations of the average programmer

⇒ Techniques that can be taught and do not rely on inspiration or perspiration

⇒ Facilitate the organized control of software projects

7.6 Steps in JSP

⇒ Draw structure diagrams for each set of data such that the structure reflects the way in which the data is to be processed.

⇒ Identify points of correspondence of a one-for-one nature between components of individual data structures.

⇒ Produce a program structure diagram using the same notation as that used in data structures, and based on the data structures, combine them at the points of correspondence.

⇒ For each iteration and selection appearing in the program structure diagram, construct appropriate conditions.

⇒ Examine the specification and the conditions and from these draw up a list of basic program operations in plain language.

⇒ Allocate the conditions and operations to the appropriate components of the program structure.

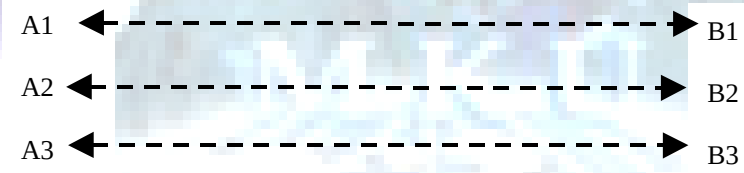
⇒ Produce ©schematic logic[a], also known as pseudo code from the program structure.

⇒ Implement the pseudo-code in a target high-level programming language.

7.7 Correspondence Between Data Structures

⇒ Program structure is derived from input and output data structures by identifying correspondences between components of individual data structures.

⇒ If component A corresponds with component B, the data which they represent can be shown diagrammatically as:



⇒ Rules of Correspondence

- Records (input and output) must be in the same order.
- Records in the same number of each;
- Output derived from input.

⇒ Structure Clash: occurrence where correspondences are not possible between the two data structures.

⇒ May be resolved by Program Inversion.

7.8 Listing the Elementary Program Operations

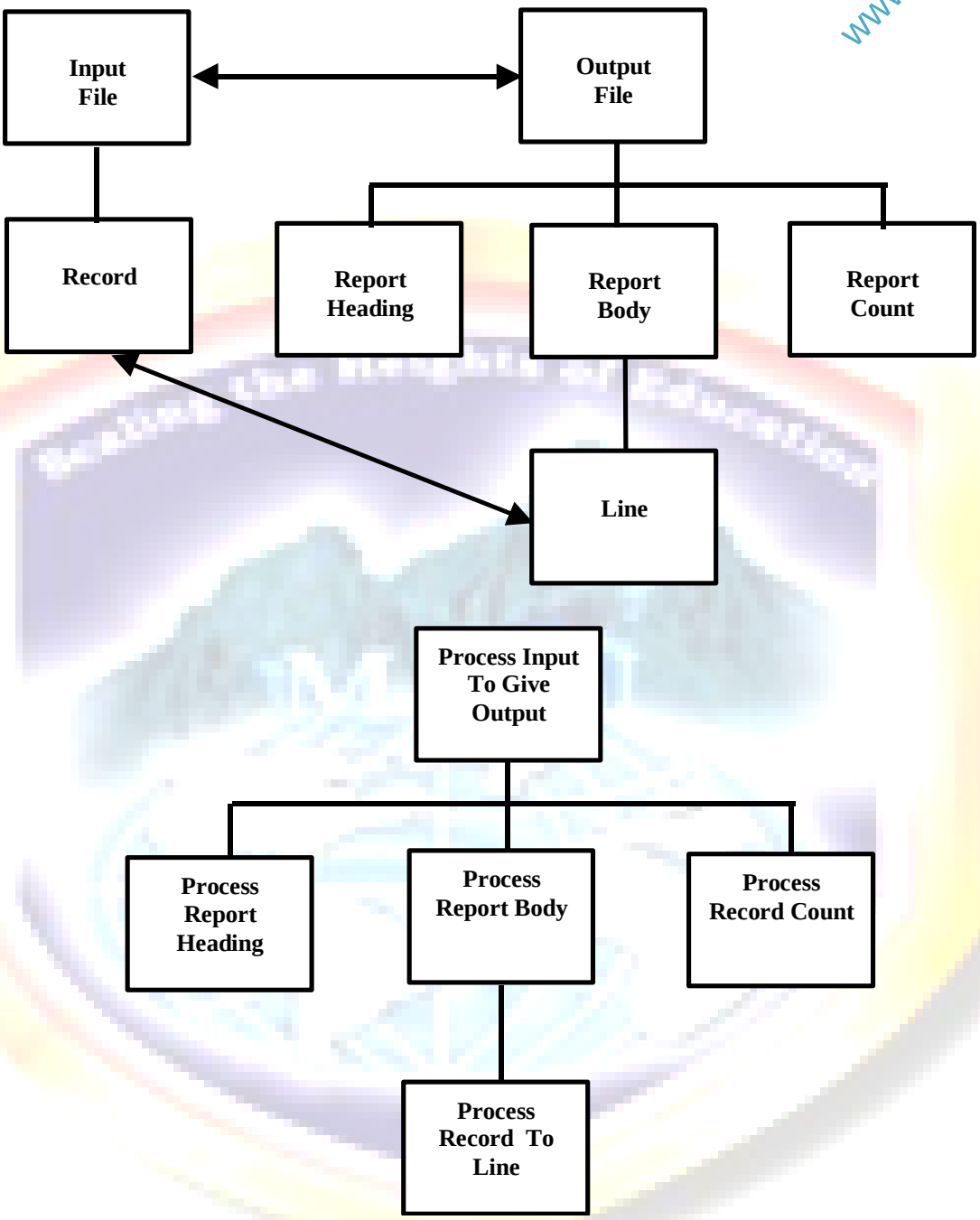
⇒ The elementary program operations can be listed by studying the program specification and taking note of the program structure and conditions.

- List the program initialization and finalization operations such as open and close files.
- Identify the input records or components and hence list the input operations.
- Identify the output records or components and hence list the output operations.
- Identify any computations or transformations from input to output necessary to produce the detailed aspects of the required results.
- List any detailed initialization operations that will be required.
- List any operations necessary to support the condition list.

Examples

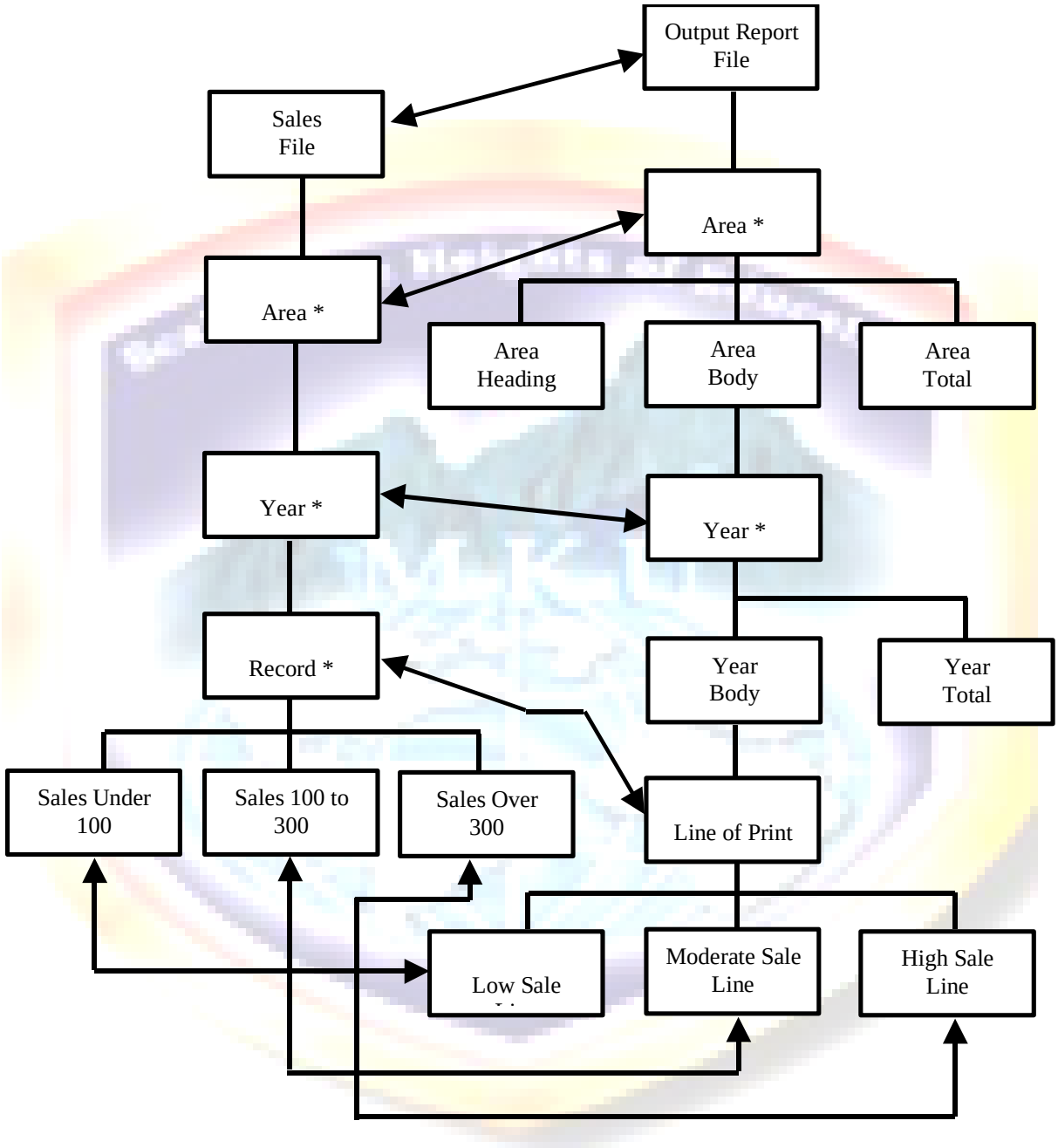
⇒ Example 1

We have a serial file of records and we wish to print them one record per line. The input file data structure is simply an iteration of records for printing, and the output file data structure an iteration of lines (each containing one record), and we want to print a report heading at the start and a line containing a record count at the end.

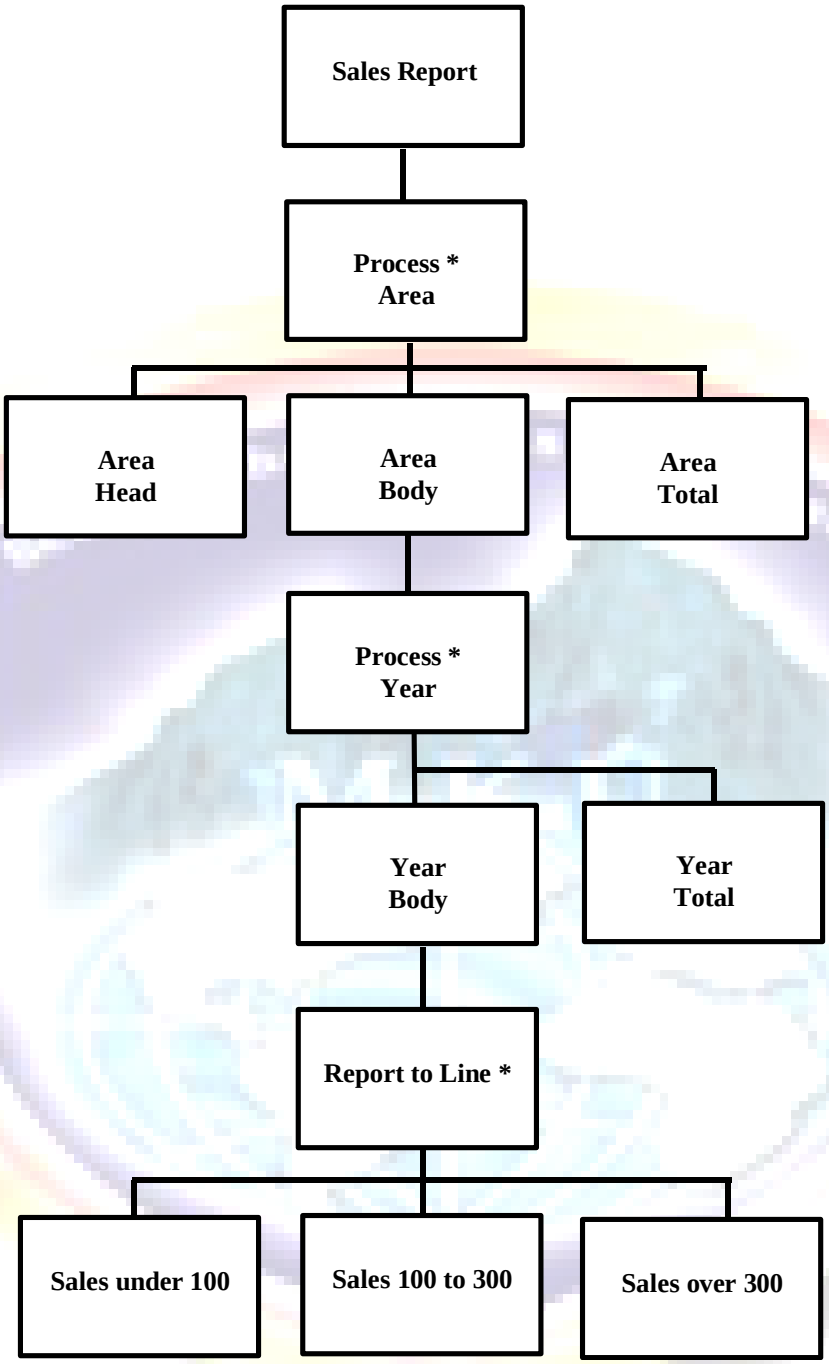


⇒ Example 2

- We have a sales file which is sorted into year-within-sales area code. It is required to produce a report to show the sales details, with appropriate high-lighting for low, moderate and high sales (that is, a single exclamation mark when the sales value is less than 100, two exclamation marks when the sales value is between 100 and 300, and three exclamation marks for values greater than 300). Headings are required for each area and totals are to be produced at relevant control breaks, that is, at change of year end area code.

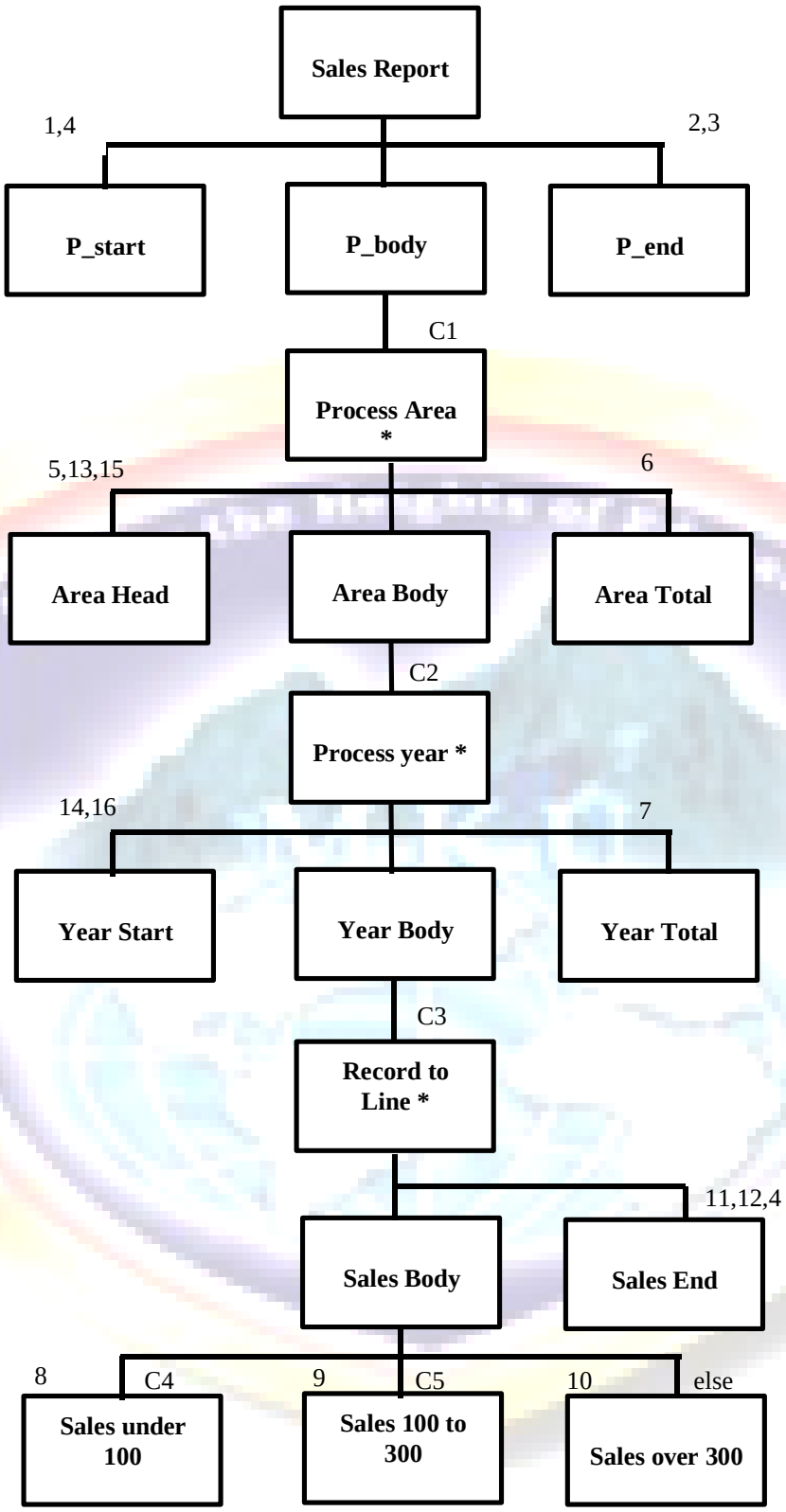


Operation List should include:



⇒ Program Initialization and Finalisation operations

- Open files



- Close files
- Stop

⇒ Input Operations

- Read a Sales file record

⇒ Output Operations

- Print area headings
- Print area total
- Print year total
- Print a sales under 100
- Print a sales from 100 to 300
- Print a sales over 300

⇒ Computation Operations

- Add to area total
- Add to year total

⇒ Initialization Operations

- Initialize area total to zero
- Initialize year total to zero

⇒ Operations to support condition list

- Store area code
- Store year

⇒ The condition list should include:

- C1 - Until end of sales file
- C2 - Until end of sales file or change of area
- C3 - Until end of sales file or change of area or change of year
- C4 - If sales value <100
- C5 - If sales value >= 100 and<= 300

Chapter Review Questions

1. A sequential-update program is required to maintain a master file of rental records on video tape rentals. The transaction file contains the following types of transaction:

   ⇒ inserting a new video title,

   ⇒ amending the rental-status of a video title,

   ⇒ deleting a video title

   These are represented by transaction type code 1,2 and 3 respectively. For transaction type 1 and 3, there would be at most one transaction for each video title. But for type 2 record, there could be multiple transactions for each video title.

   The master and transaction files sorted in ascending sequence of video title code.
   The program has to cater for possible matching errors between the master and the transaction files

   Use Jackson's Structured Programming Design Methods to:

   a) Draw the data structures for the respective files, and show the correspondences between their components

   b) List the conditions and operations

   c) Produce the final program structure

   State clearly any assumption(s) that you make

2. ABC company employs 8 salesmen. Records of sales by salesman for every month are stored in the master file. The management wants sales report listing the salesman and their sales figures from January to December in ascending order.

   Program Specification

   The program uses master file to generate a report.Master file contains

   ⇒ salesman name

   ⇒ 12 monthly sales(January to December)

   The master file is sorted sequentially using salesman name.The master file contains only valid data.

   Given the program specification above, use Jackson's Structured Programming design method to:

   a) Draw the data structures for the respective files, and show the correspondences between their components.

   b) List the conditions and operations.

   c) Produce the final program structure.

References for Further Reading

1. Pressman R.S (1997) Software engineering: a practitioner's Approach, McGraw Hill
2) Somerville Ian (2002) software engineering ,Pearson's education

CHAPTER 8: SOFTWARE QUALITY ASSURANCE

Chapter Objectives

At the end of this chapter, stdent should understand::

Software Quality Assurance

Software Quality Factors

Software Quality Assurance Major Activities

Formal Technical Review

Software Reliability

Software Quality Assurance Approach

- Examining the need for SQA
- Benefits of SQA
- Constraints of SQA

8.1 Software Quality Assurance

⇒ Software quality can be defined as:

- *Conformance to explicitly-stated functional and performance requirements, explicitly-documented development standards, and implicit characteristics that are expected of all professionally developed software.*

⇒ This definition thus emphasizes on three important points:

- Quality will be measured with Software Requirements, i.e. lack of conformance to these requirements will mean lack of quality.
- Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria is not followed, lack of quality will result.
- The set of implicit requirements, e.g. good maintainability, must still be followed. If these are not met, software quality is suspect.

⇒ Software quality is thus a mix of factors that will vary across different applications and different customers. The sections below will thus be concerned about:

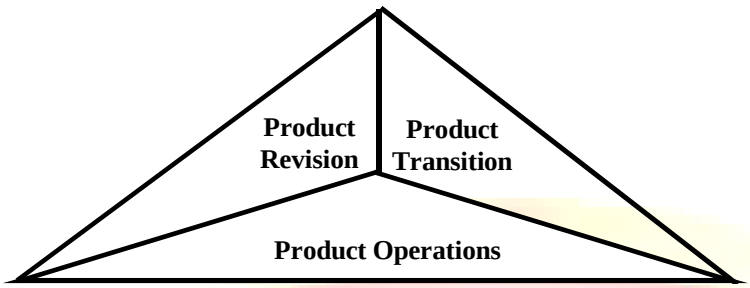
- Identification of Software Quality Factors
- Human activities required to achieve them

8.2 Software Quality Factors

⇒ In order to help us categorize software quality factors, McCall proposes a categorisation which focuses on three important aspects of a software product:

- ***Product Operations***: A software product's operational characteristics;

- ***Product Reunion:*** Its ability to undergo change,
- ***Produced Transition***: Its adaptability to new environments.



8.2.1 Product Operations

⇒ *Correctness*

- Does it do what I want?
  - The extent to which a program satisfies its specification and fulfils the mission objectives.

⇒ *Reliability*

- Does it do it accurately all the time?
  - The extent to which a program can be expected to perform its intended function with required precision.

⇒ *Efficiency*

- Will it run on my hardware as well as it can?
  - The amount of computing resources and code required by a program to perform a function.

⇒ *Integrity*

- Is it secure?
  - The extent to which access to software or data by unauthorised persons can be controlled.

⇒ *Usability*

- Is it designed for the user?
  - The effort required to learn, operate, prepare input, and interpret output of a program.

8.2.2 Product Revision

⇒ *Maintainability*

- Can I fix it?
  - The effort required to locate and fix an error in a program.

⇒ *Flexibility*

- Can I change it?
  - The effort required to modify an operational program.

⇒ *Testability*

- Can I test it?

- The effort required to test a program to ensure that it performs its intended function.

8.2.3 Product Transition

⇒ *Portability*

- Will I be able to use it on another machine?
  - The effort to transfer the program from one hardware and/or software system environment to another.

⇒ *Reusability*

- Will I be able to reuse some of the software?
  - The extent to which a program (or parts of a program) can be reused in other applications-related to the packaging and scope of the functions that the program performs.

⇒ *Interoperability*

- Will I be able to interface it with another system?
  - The effort required to couple one system to another.

8.3 Software Quality Assurance Major Activities

⇒ *Software Quality Assurance (SQA) is a ©planned and systematic pattern of actions[a] that are required to ensure quality in software.*

- The SQA group in any company serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view. Does the software adequately meet the quality factors from the customer's point of view? Does the software adequately meet the quality factors described earlier? Has software development been conducted according to pre-established standards? Have technical disciplines properly performed their roles as part of the SQA activity?

8.3.1 SQA Activities

⇒ SQA is comprised of a variety of tasks associated with seven major activities:

- *Application of Technical Methods*
  - SQA begins with a set of technical methods and tools that help the analyst to achieve a high-quality specification and the designer to develop a high-quality design.
- *Conduct of Formal Technical Review*
  - Activity that accomplishes quality assessment for the specification and the design. The FTR is a stylised meeting conducted by technical staff with the sole purpose of uncovering quality problems.
- Testing of Software
  - Combines a multistep strategy with a series of test case design methods that help ensure effective error detection.
- *Enforcement of standards*
  - Degree in which this is applied varies from company to company. In many cases, standards are dictated by customers or regulatory mandates. In other situations, standards are self-imposed.
- *Control of Change*
  - Every change to software has the potential of introducing errors or creating side effects that propagate errors. The change control process thus contributes directly

to software quality by formalising requests for change, evaluating the nature of change, and controlling the impact of change.

- *Measurement*
  - Track software quality and assess the impact of methodological and procedural changes on improved software quality.
- Record keeping and recording
  - Collection and dissemination of SQA information. The results of reviews, audits, change control, testing, and other SQA activities must become part of a historical record for a project and should be disseminated to the development staff on a need-to-know basis.

8.4 Formal Technical Reviews

⇒ Formal technical review is

- A class of reviews that include walkthroughs, inspections, round-robin reviews, and other small group technical assessments of software
- A planned and controlled meeting attended by a group of diversified people

8.4.1 Objectives of FTR

⇒ To uncover errors in function, logic, or implementation for any representation of the software

⇒ To verify that the software under review meets its requirements

⇒ To ensure that the software has been represented according to predefined standards

⇒ To achieve software that is developed in a uniform manner

⇒ To make projects more manageable

8.4.2 Effects of FTR

⇒ Early discovery of software defects so the development and maintenance phase is substantially reduced

⇒ Serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation

⇒ Serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen

8.4.3 Guidelines for the Organization and Preparation of FTR

⇒ Should involve between three and five people

⇒ Advance preparation should occur but should not require no more than 2 hours of work for each person

⇒ The duration of the review meeting should be less than 2 hours

⇒ Focus on a components of the software (eg. a portion of requirements specification, a detailed module design, a source code listing for a module)

⇒ Producer should report his/her progress

⇒ A recorder should actively record all issues as

- What was reviewed?
- Who reviewed it?
- What were the findings and conclusions?

⇒ The attenders must make a decision at the end of the session as to

- Accept the product
- Reject the product
- Accept the product provisionally, subject to further modification

8.4.4 Review Guidelines during the FTR

⇒ *Review the product, not the producer*

- Tone of the meeting should be loose and constructive, and intent should not be to embarrass or belittle.

⇒ *Set an agenda and maintain it*

- One main problem in all types of meetings is the tendency to drift. The FTR must be kept on track and on schedule.

⇒ *Limit debate and rebuttal*

- There may not be universal agreement on some issues. Rather than spending time debating the issue, the issue should be recorded for further discussion off-line.

⇒ *Enunciate problem areas, but don't attempt to solve every problem noted*

- A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone. Problem solving should be postponed until after the review meeting.

⇒ *Take written notes*

- Makes notes on a wall board so that wording and prioritisation can be assessed by the other reviewers as information is recorded.

⇒ *Limit the number of participants and insist upon advance preparation*

- Keep the number of people involved to the necessary minimum.

⇒ *Develop a checklist for each product that is likely to be reviewed*

- A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code and even test documents.

⇒ *Allocate resources and time schedule for FTRs*

- For reviews to be effective, they should be scheduled as tasks during the software engineering process.

⇒ *Conduct meaningful training for all reviewers*

- To be effective, all review participants should receive some formal training. The training should stress both process-related issues and the human psychological side of reviews.

⇒ *Review your early reviews*

- Debriefing can be beneficial in uncovering problems with the review process itself.

8.5 Software Reliability

⇒ Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g. the effects of temperature, corrosion, shock) are more likely than a design-related failure. The opposite is true for software: in fact, all software failures can be traced to design or implementation problems; wear does not enter into the picture.

⇒ Software Reliability: The probability of failure free operation of a computer program in a specified environment for a specified time.

⇒ A simple measure of reliability is mean time between failure (MTBF), where

$$MTBF = MTTF + MTTR$$

⇒ In addition to this reliability measure, a measure of availability can be calculated.

⇒ Software availability is the probability that a program is operating according to requirements at a given point in time and is defined by:

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR} * 100\%$$

8.6 Software Quality Assurance Approach

⇒ At the low end of the scale, quality is the sole responsibility of the individual who may engineer, review, and test at any comfort level. At the high end of the scale, an SQA group is charged with the responsibility standards and procedures for achieving software quality and ensuring that each is followed.

⇒ Before formal quality assurance procedures are instituted, a software development organisation should adopt software engineering procedures, methods, and tools. This methodology, when combined with an effective paradigm for software development, can do much to improve the quality of all the software produced by the organisation.

⇒ Manager/managers and practitioners are not interested in establishing formal SQA functions as:

- Managers are reluctant to incur the extra up-front cost
- Practitioners feel they are already doing everything that needs to be done
- No one knows where to put such a function organisationally
- Everyone wants to avoid the "red tape" that SQA is preceived to introduce

8.6.1 Benefits of SQA

⇒ Software will have fewer latent defects, resulting in reduced effort and time spent during testing and maintenance

⇒ Higher reliability will result in greater customer satisfaction

⇒ Maintenance costs (a substantial percentage of all software costs can be reduced)

⇒ Overall life cycle cost of software is reduced

8.6.2 Constraints of SQA

⇒ Difficult to institute in small organizations, where available resources to perform the necessary activities are not available

⇒ SQA represents cultural change - and change is never easy

⇒ SQA required the expenditure of dollars that would not otherwise be explicitly budgeted to software engineering or quality assurance

1. Explain why software design metrics only are inadequate for predicting the quality of the software
2. Elaborate how you can asses the quality of a particular application software
3. Identify various metrics that can be used to determine the reliability of software

References for Further Reading

1. Pressman R.S (1997) Software engineering: a practitioner's Approach, McGraw Hill
3) Somerville Ian (2002) software engineering ,Pearson's education

CHAPTER 9: SOFTWARE TESTING TECHNIQUES

Chapter Objectives:

At the end of this chapter, stdent should understand::

⇒ Testing Objectives

⇒ Information Flow during Testing

⇒ General guidelines in Test Case Design

⇒ Testing strategies

- White Box Testing
- Black Box Testing

⇒ Automated Testing Tools

Software Testing Techniques

⇒ Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding.

⇒ Software testing fundamentals define the overriding objectives for software testing.

9.1 Testing Objectives

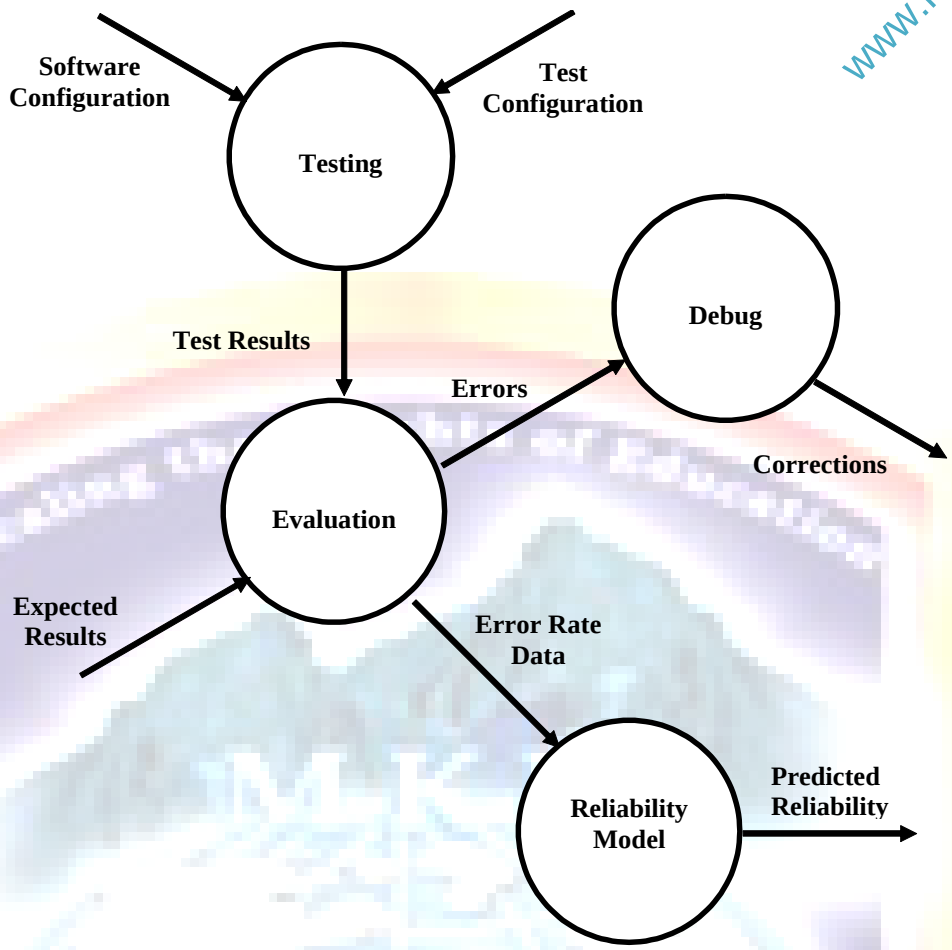
⇒ Testing is a process of executing a program with the intent of finding an error

⇒ 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.

Testing cannot show the absence of defects, it can only show that software defects are present.

9.2 Information Flow In Testing



⇒ Two classes of input are provided to the test process, namely: (1) a software configuration that includes a Software Requirements Specification, a Design Specification, and source code; (2) a test configuration that includes a Test Plan and Procedure, any testing tools that are to be used, and test cases and their expected results.

⇒ Tests are conducted and all results are evaluated, i.e. test results are compared with the expected results. When erroneous data is encountered, debugging commences. As test results are gathered and evaluated, a qualitative indication of software quality and reliability begins to surface. Two possible situations can occur here: if there are severe errors that require design modification are encountered regularly, software quality and reliability are suspect, and further tests are indicated. if, on the other hand , software functions appear to be working properly and the errors encountered are easily correctable, either (1) Software quality and reliability are acceptable, or (2) tests are inadequate to uncover severe errors.

⇒ The results accumulated during testing can be evaluated in a more formal manner: Software reliability models use error-rate data to predict future occurrences of errors, and hence, reliability.

9.3 Test Case Design

⇒ General mistakes

- Software engineers often treat testing as an afterthought
- Developing test cases that may "feel right" but have assurance of being complete

⇒ General guidelines

- Provide a mechanism
- Provide highest likelihood for uncovering errors in software

- Finding the most errors within a minimum amount of time and effort

9.4 White Box Testing

⇒ White box testing is a test case design method that uses the control structure of the procedural design to derive test cases.

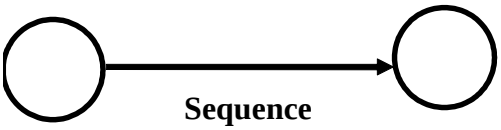
⇒ Using white box testing methods, the software engineer can derive test cases that:

- Guarantee that all independent paths within a module have been execrised at least once;
- Exercise all logical decisions on their true or false sides;
- Execute all loops at their boundaries and within their operational bounds ; and
- Exercise internal data structures to ensure their validity.

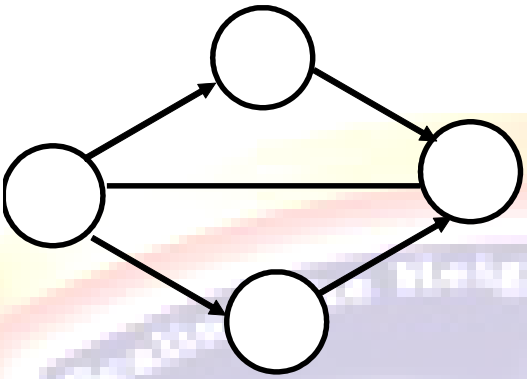
⇒ There are also several reasons why the logic-ness of a program should be tested.The answer lies in the nature of software defects:

- Logic errors and incorrect assumptions are inversely propotional to the probability that a program path will evaluated
- Misconceptions that a logical path is unlikely to be executed when, in fact, it may be executed on a regular basis.
- Typographical errors are random. When a program is translated into programming language source code; it is likely that some typing errors will occur. Many will be uncovered by syntax checking mechanisms, but others will go undetected until testing begins.

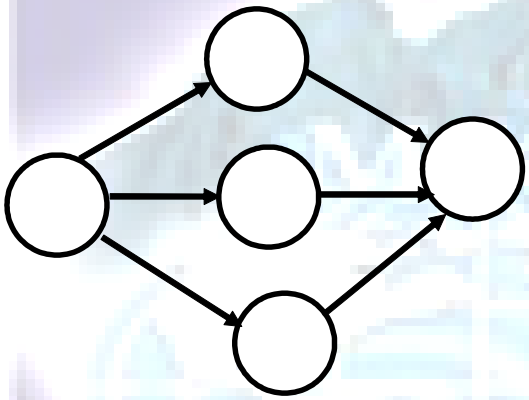
Flow Graph Notation



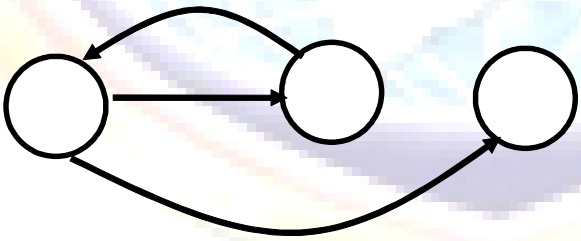
Sequence



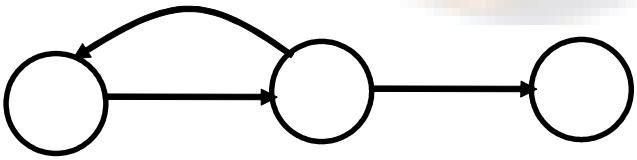
If



Case



While



Until

Basis Path Testing

⇒ This is a white box testing techniques first proposed by Tom McCabe.

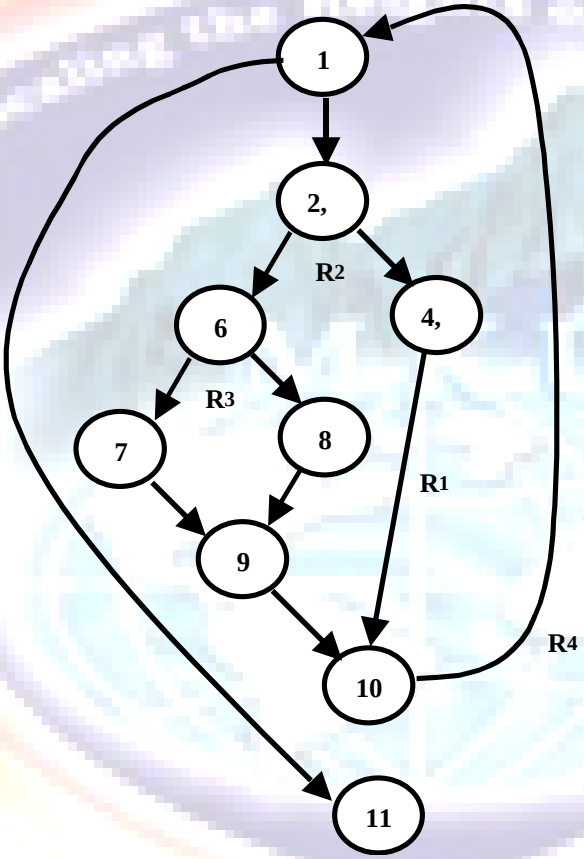
⇒ The basis path method enables that test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

⇒ Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Cyclomatic Complexity

⇒ Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.

⇒ When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.



⇒ Complexity can be computed in one of several ways:

- The number of regions of the flow graph correspond to the cyclomatic complexity
- Cyclomatic complexity, V(G), for a flow graph G is defined as V(G) = E-N+2; where E is the number of flow graph edges and N is the number of flow graph nodes.
- Cyclomatic complexity, V(G) for a flow graph G is also defined as V(G) =P+1 where P is the number of predicate nodes contained in the flow graph G.

1. The flow has 4 regions
2. V(G) = E - N + 2 = 11 edges - 9 nodes + 2 = 4
3. V(G) = P + 1 = 3 Predicate Notes + 1 = 4

- Thus, the value of V(G) provides us with an upper bound for the number of independent paths ....(etc...)

⇒ Thus, the value of V(G) provides us with an upper bound for the number of independent paths that comprise the basis set, and by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

⇒ The basis path testing can be applied as a series of steps:

- Using the design or code as foundation , draw a corresponding flow graph.
- Determine the cyclomatic complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.

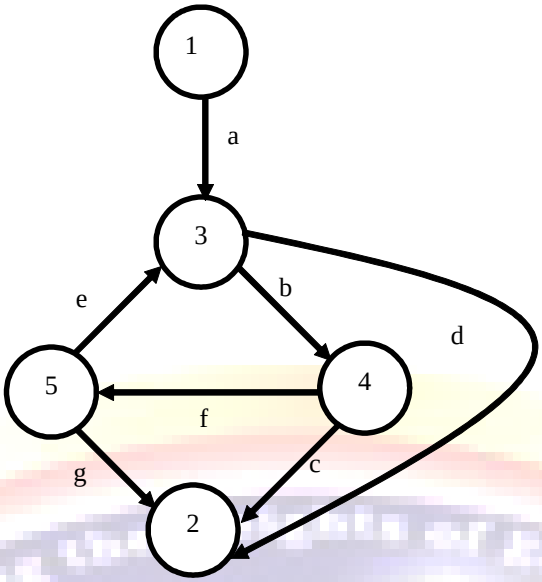
Graph Matrix

⇒ A graph matrix is a software tool that is developed that assist in basis path testing.

⇒ The graph matrix is initially nothing more than a tabular representation of a flow graph

⇒ To make it more useful , alink weight is added to each matrix entry. The link weight provides additional information about control flow . In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist) . But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be executed
- The processing time expended during traversal of a link
- The memory required during traversal of a link
- The resources required during traversal of a link



Flow Graph

Connected to node

Node	1	2	3	4	5	Connections
1			a			1-1 = 0
2						
3		d		b		2-1 = 1
4		c				2-1 = 1
5		g	e			2-1 = 1

Cyclomatic complexity : 3 + 1 = 4

Control Structure Testing

⇒ The basis path testing technique described earlier is one of a number of techniques for control structure testing.

⇒ In this section, other variations of control structure testing are discussed.

- Basis Path testing (as covered earlier in this chapter)
- Condition testing
- Data Flow testing
- Loop testing

Condition Testing

⇒ A simple condition is boolean variable or relational expression

⇒ A test case design method that exercises the logical conditions contained in a program module

⇒ Focuses on testing each condition in the program

Data flow testing

⇒ Selects test paths of a program according to the locations of definitions and uses of variables in the program

⇒ Useful for selecting test paths of a program containing nested if an loop statements

⇒ Effective for error protection

⇒ Problems of measuring test coverage and selecting test paths are more diffcult than the corresponding problems for condition testing

Loop testing

⇒ Focuses exclusively on the validity of loop constructs

⇒ Four classes of loops : simple loops, concatenated loops, nested loops, and unstructured loops

Test cases for simple loops

⇒ Where n is the maximum number of allowable passes through the loop

⇒ Skip the loop entirely

⇒ Only one pass through the loop

⇒ Two passes through the loop

⇒ m passes through the loop where m < n

⇒ n-1, n, n+1 passes through the loop

Test cases for nested loops

⇒ Start at the innermost loops. Set all other loops to minimum values

⇒ Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values. Add other tests for out-of -range or excluded values

⇒ Work outward, conducting tests for the next loop but keeping all other outer loops at minimum values and other nested loops to "typical" values

⇒ Continue until all loops have been tested

Test cases for concatenated loops

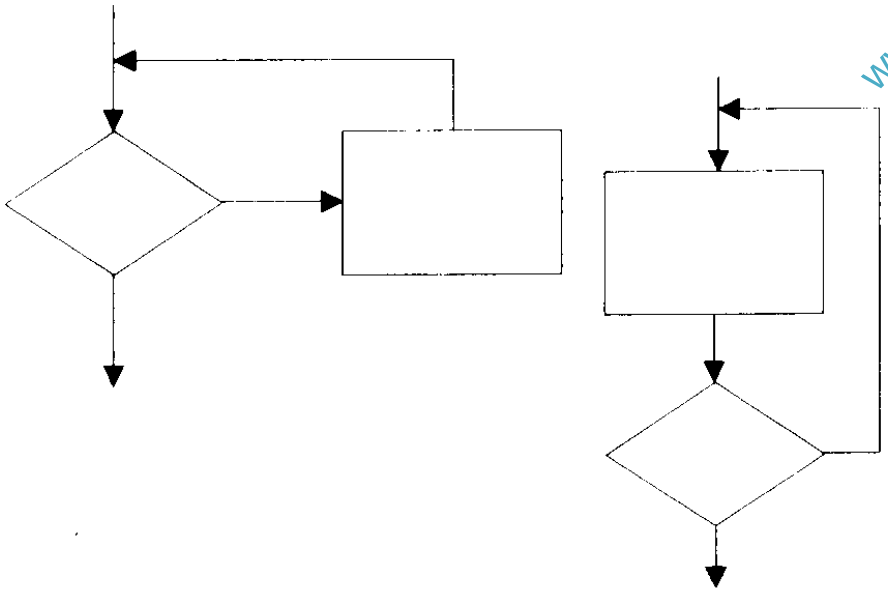
⇒ If esch of the loops is independent of the others, perform simple loop tests for each loop

⇒ If the loops are dependent, apply the nested loop tests

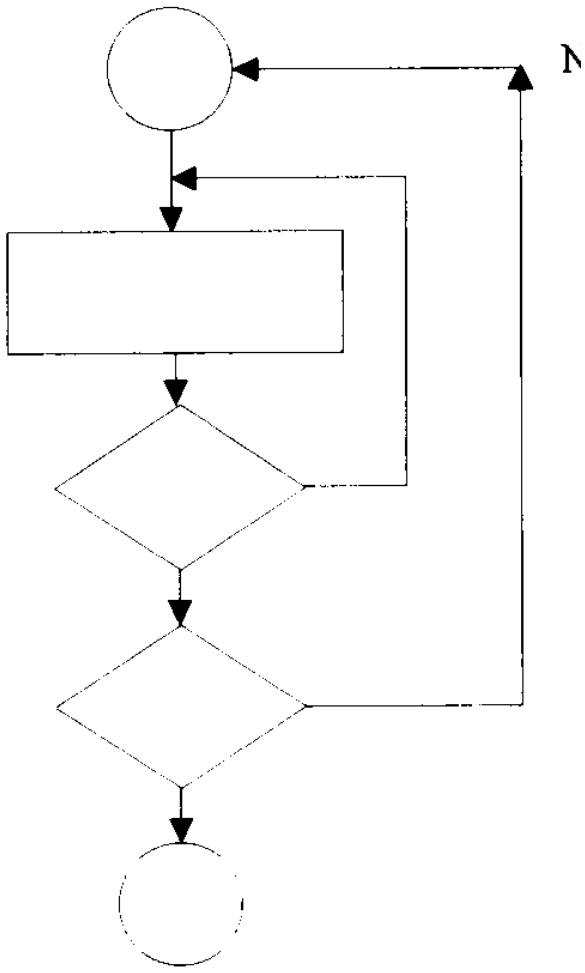
Test cases for unstructured loops

⇒ Whenever possible, redesign this class of loops to reflect the structured programming constructs

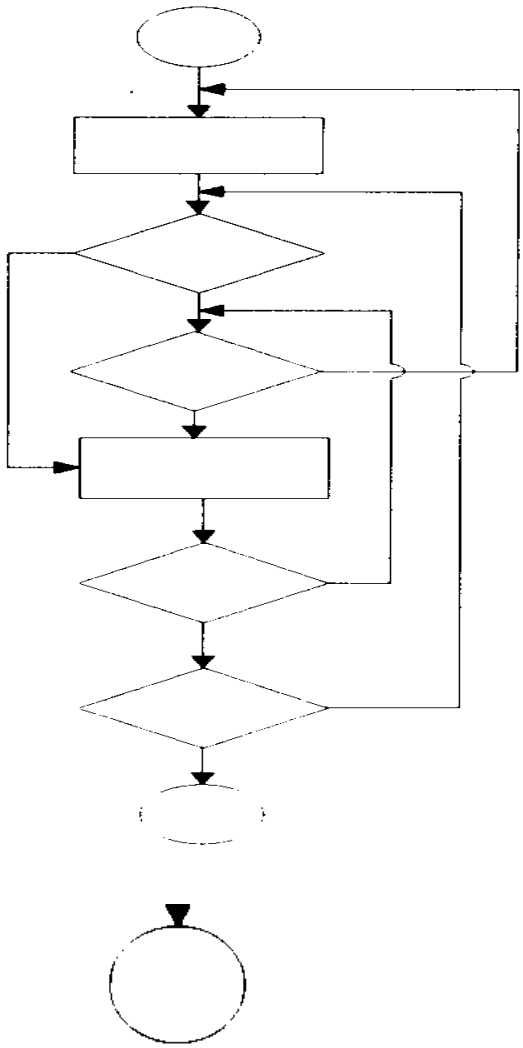
Simple Loops



Nested Loops



Concatenated Loops



Unstructured Loops

9.5 Black Box Testing

⇒ Black box testing methods focus on the functional requirements of the software, i.e. black box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

⇒ This is not an alternative technique to white box testing techniques , rather it is a complimentary approach that is likely to uncover a different class of errors than white box methods.

⇒ Black box testing attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external databses access
- Performance errors
- Initialisationand termination errors.

⇒ Unlike white box testing which is performed early in the testing process, black box testing tends to be applied during later stages of testng. This is because black box testing purposely disregards control structure, attention is focused on the information domain.

⇒ By applying black box testing techniques, a set of test cases can be derived to satisfy the folllowing criteria:

- Test cases that reduce, by a count that is greater than 1, the number of additional test cases that must be designed to achieve reasonable testing; and
- Tests cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

⇒ Black Box Testing techniques:

- Equivalent Partitioning.
- Boundary Value Analysis
- Cause-Effect Graphing Techniques
- Comparison Testing

Equivalence Partitioning

⇒ Divides the input domain of a program into classes of data

⇒ Strives to define a test case that uncovers classes of errors

⇒ Equivalence classes may be defined according to the following guideline:

- If an input condition specifies a range or a specifies values, one valid and two invalid equivalence classes are defined
- If an input condition specifies a member of a set or a boolean, one valid and one invalid class are defined

Boundary Value Analysis

⇒ Select test cases at the "edges" of the classes

⇒ Rather than focusing solely on input conditions, it also derives test cases from the output domain

⇒ Guidelines of designing test cases

- If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b, just below a and b, respectively
- If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested
- Apply guidelines 1 and 2 to output conditions
- If internal data structures have prescibed boundaries (eg. an array has a defined limit of 100 entries), be certain to design test cases to exercise the data structure to its boundary

Cause-effect Graphing Techniques

⇒ Provides a concise representation of logical condition and corresponding actions

⇒ This techniques follows four steps:

- Causes(input conditions) and effect(output conditions) are listed for a module and an identifier is assigned to each
- A cause-effect graph is developed
- The graph is converted to a decision table
- Decision table rules are converted to test case

Comparison testing

⇒ A technique used when the reliability of software is absolutely critical

⇒ In this techinique, multiple and independent versions of software is developed for critical applications, even when only a single version will be used in the delivered computer-based system

⇒ Each version is tested with the same test data to ensure that all provide identical output. Then all the versions are executed in parellel with a real-time comparison of results to ensure consistency

⇒ Also known as back-to-back testing

9.6 Automated Testing Tools

⇒ Code auditors: These special-purpose filters are used to check the quality of software to ensure that it meets minimum coding standards.

⇒ Assertion processors: These pre-processors /postprocessors systems are employed to telll whether programmer-supplied claimes , called assertions, about a program's behaviour are actually met during real program executions.

⇒ Test file generators: These processors generate, and fill with predetermined values, typical input files for programs that are undergoing testing.

⇒ Test data generators: These automated analysis systems assist auser in selecting test data that make a program behave in a particular fashion.

⇒ Test verifiers: These tools measure internal test coverage, often expressed in terms that are related to the control structure of the test object, and report the coverage value to the quality assurance expert.

⇒ Output comparators: This tool makes it possible to compare one set of outputs from a program with another (previously archived) set to determine the difference between them.

Chapter Review Questions

1. Differentiate between structural and function testing strategies
2. Distinguish between static and dynamic testing techniques
3. explain the role of Validation and verification process in ensuring software quality

References for Further Reading

1. Pressman R.S (1997) Software engineering: a practitioner’s Approach, McGraw Hill
2. Somerville Ian (2002) software engineering ,Pearson’s education

CHAPTER 10: SOFTWARE TESTING

Chapter Objectives:

At the end of this chapter, stdent should understand::

Overview of Software Testing Strategies

Verification and Validation

Software Testing

- A Software Testing Strategy
- Types of testing

Validation and verification (V&V)

System Testing

Debugging and Debugging Tools

10.1 Overview Of Software Testing Strategies

⇒ A strategy for software testing integrates software test case design techniques into a well-planned series of steps that result in the successful construction of software. A testing strategy must always incorporate test planning, test case design, text execution, and the resultant data collection and evaluation

⇒ Generic characteristics of all software testing strategies:

- Testing beings at the module level and works "outward" toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects) an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

⇒ A strategy for software testing must accomodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements.

10.2 Verification and Validation

⇒ Software testing is one element of a broader topic that is often referred to as vertification and validation.

⇒ Verification refers to the set of activities that ensure that software correctly implements a specific function, i.e.

⇒ Verification: " Are we building the project right?"

⇒ Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements, i.e

⇒ Validation: Are we building the right product?

⇒ The activities required to achieve software quality may be viewed as a set of components. Software engineering methods provide the foundation from which quality is built. Analysis, design, and implementation (coding) methods act to enhance quality by providing uniform techniques and predictable results. Formal technical reviews (walk-throughs) help to ensure the quality of the products produced as a consequence of each software engineering step. Through the process, measurement and control are applied to every element of a software configuration. Standards and procedures help to ensure uniformity and a formal SQA process enforces a "total quality philosophy".

10.3 Organization For Software Testing

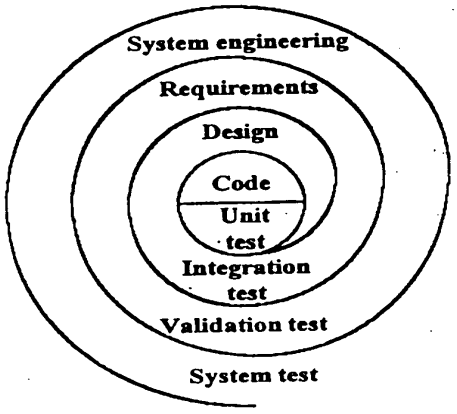
⇒ Misconceptions in Software testing:

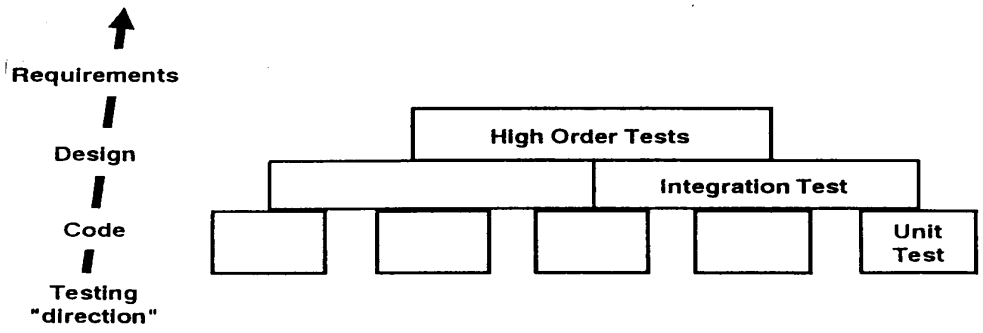
- That the developer of software should not do any testing at all;]
- that the software should be "tossed over the wall" to strangers who will test it mercilessly;
- That testers get involved with the project only when the testing steps are about to begin.

⇒ The following points can refute the above misconceptions:

- The software developer is always responsible for testing the individual units (modules) of the program, ensuring that each performs the funcction for which it was designed. In many cases , the developer also conducts integration testing - the testing step that leads to the construction (and test) of the complete program structure. Only after the software architecture is complete does an independent test group become involved.
- The role of an independent test group (ITG) is to remove the inherent problems associated with letting the building test the thing that has been built. In other words, they will remove the conflict of interest that will otherwise be present. After all, the personnel in the independent group are paid to find errors.
- However, this does not mean that the software developer does not get involved: he must still be available to correct errors that are uncovered once testing starts.

10.4 A Software Testing Strategy





⇒ A strategy for software testing may also be envisioned by moving outward along the spiral

⇒ Unit testing begins at the vortex of the spiral and concentrates on each unit of the software as implemented in the source code.

⇒ Testing progresses by moving outward along the spiral to integration testing, where the focus is on the design and the construction of the software architecture.

⇒ Taking another turn outward on the spiral , validation testing is encountered, where requirements established as part of software requirement analysis are validated against the software that has been constructed.

⇒ Finally, at system testing, where the software and other system elements are tested as a whole.

⇒ Unit tests: focuses on each module and makes heavy use of white box testing

⇒ Integration tests: focuses on the design and construction of software architecture; black box testing is most prevalent with limited white box testing.

⇒ High-order tests: conduct validation and system tests. Makes use of black box testing exclusively.

10.5 Unit Testing

⇒ Unit testing focuses vertification effort on the smallest unit of software design - the module.

⇒ Using the detail design description as a guide, important control paths are tested to uncover errors within the boundary of the module

⇒ The unit test is always white box-oriented

Unit Test Considerations

⇒ The module interface is tested to ensure that information properly flows into and out of the program unit under test

⇒ The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.

⇒ Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.

⇒ All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once.

⇒ And finally, all error-handling paths are tested.

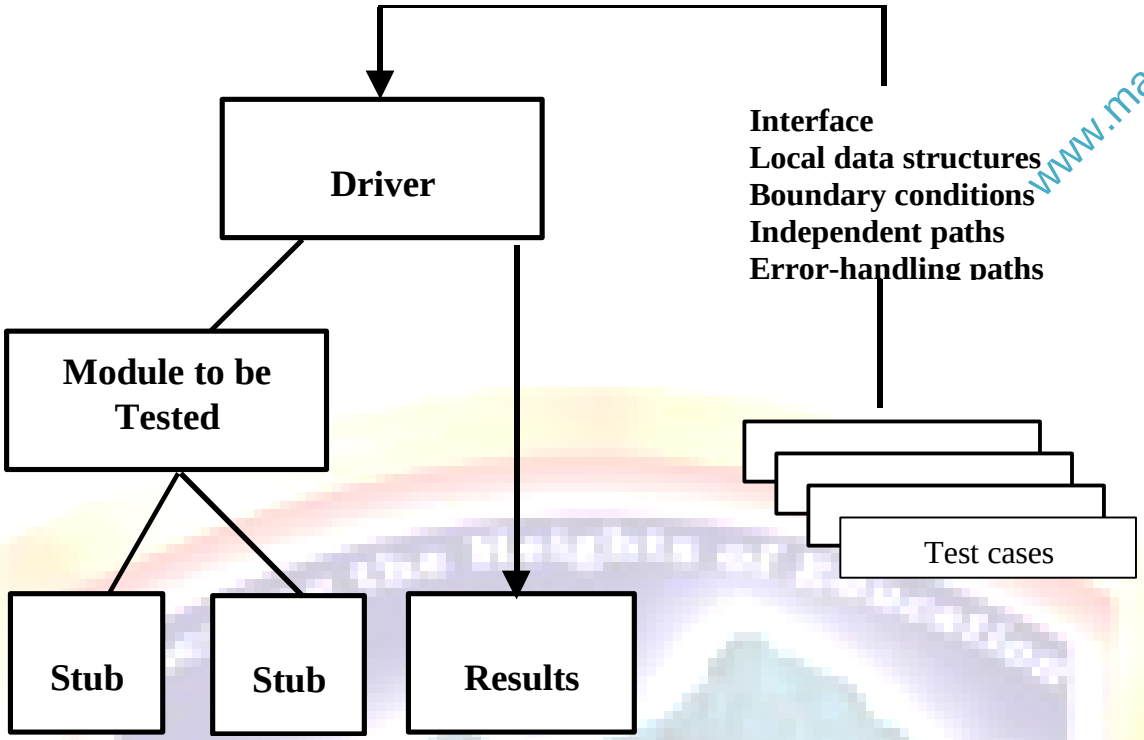
Unit Test Procedures

⇒ Because a module is not a stand-alone program, driver and/or stub software must be developed for each unit test.

⇒ In most applications, a driver is nothing more than a " main program" that accepts test case data, passes such data to the module (to be tested), and prints the relevant results.

⇒ Stubs serve to replace modules that are subordinate(called by) the module to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do nominal data manipulation, prints vertification of entry, and returns.

⇒ Drivers and stubs also represent overhead.



10.6 Integration Testing

⇒ Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover tests to uncover errors associated with interfacing

⇒ The objective is to take unit-tested modules and build a program structure that has been dictated by design.

⇒ Two-types: Top-Down integration; Bottom-up Integration

Top-down Integration

⇒ An incremental approach

⇒ Modules are integrated by moving downward through the control hierarchy, begining with main control module

⇒ Subordinate modules are incorporated into the structure in either a depth-first or breadth-first manner

⇒ Integration Process

- The main control module is used as a test driver and stubs are substitued for all modules directly subordinate to the main control module
- Subordinate stubs are replaced one at a time with actual modules
- Tests are conducted as each module is integrated
- On the completion of each set of tests, another stub is replaced with the real module
- Regression testing (ie, conducting all or some of the previous tests) may be conducted to ensure that new errors have not been introduced

⇒ Major problem

- Inadequate testing at upper levels when data flows at low levels in the hierarchy are required

⇒ Alternatives to the above problem

- Delay many test until stubs are replaced with actual modules; but this can lead to diffculties in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach
- Develop stubs that perform limited functions that simulate the actual module; but this can lead to significant overhead
- Perform bottom-up integration

Bottom-up Integration

⇒ Integration process

- Low-level modules are combined into clusters (sometimes called builds) that perform a specific software subfunction
- A driver (a control program for testing) is written to coordinate test case input and output
- The cluster is tested
- Drivers are removed and clusters are combined moving upward in the program structure

Integration Test Documentation

⇒ An overall plan for integration of the software and a description of specific tests are documentated in a Test Specification. The specification is a deliverable in the software engineering process and becomes part of the software configuration

10.7 Validation Testing

⇒ Achieve through a series of black tests that demonstrate conformity with requirements

⇒ Important element of this process is a configuration review, sometimes called an audit

⇒ A series of acceptance tests (include both alpha and beta testing)are conducted with the end users

⇒ Alpha testing

- Is conducted at the developer's site by a customer
- The developer would supervise
- Is conducted in a controlled environment

⇒ Beta testing

- Is conducted at one or more customer sites by the end user of the software
- The developer is generally not present
- Is conducted in a "live"environment

10.8 System Testing

Recovery Testing

⇒ A system test that forces software to fail in a variety of ways and verifies that recovery is properly performed

⇒ If recovery is automatic, re-initialization, check pointing mechanisms, data recovery, and restart are each evaluated for correctness

⇒ If recovery is manual, the mean time to repair is evaluated

Stress Training

⇒ Is designed to confront programs with abnormal situations where unusual quantity frequency, or volume of resources are demanded

⇒ A variation is called sensitivity testing; it attempts to uncover data combinations within valid input classes that may cause instability or improper processing

Performance Testing

⇒ To test the run-time performance of software

⇒ Extra instrumentation can monitor execution intervals, log events (eg, interrupts) as they occur, and sample machine states on a regular basis

⇒ Use of instrumentation can uncover situations that lead to degradation and possible system failure

10.9 Debugging

Characteristics of Bugs

⇒ The symptom and the cause may be geographically remote

⇒ The symptom may disappear (temporarily) when another error is corrected

⇒ The symptom may actually be caused by non-errors(eg, round-off inaccuracies)

⇒ The symptom may be caused by a human error that is not easily traced

⇒ The symptom may be caused by a result of timing problems, rather than processing problems

⇒ It may be diffcult to accurately reproduce input conditions (eg a real-time application in which input ordering is indeterminate)

⇒ The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably

⇒ The symptom may be due to causes that are distributed across a number of tasks running on different processors

Debugging Approaches

⇒ Brute force: is probably the most common and least efficient method for isolating the cause of a software error. The program is loaded with run-time traces, and WRITE statements, and hopefully some information will be produced that will indicated a clue to the cause of an error.

⇒ Backtracking: fairly common in small programs. Starting from where the sympton has been uncovered, backtrack manually until the site of the cause is found. Unfortunately, as the number of source code lines increases, the number of potential backward paths may become unmanageably large.

⇒ Cause Elimination: data related to the error occurrence are organised to isolate potential causes. A "cause hypothesis" is devised and the above data are used to prove or disapprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If the initial tests indicate that a particular cause hypothesis shows promise, the data are refined in a attempt to isolate the bug.

Debugging Tools

⇒ Debugging compliers

⇒ Dynamic debugging aides ("tracers")

⇒ Automatic test case generators

⇒ Memory dumps

⇒ Cross reference maps

Chapter Review Questions

1. Elaborate why user acceptance testing is important.
2. explain why bottom up and top down testing may be inappropriate testing strategies for object oriented systems
3. differentiate between program bugs and program defects

References for Further Reading

1. Pressman R.S (1997) Software engineering: a practitioner's Approach, McGraw Hill
2. Somerville Ian (2002) software engineering ,Pearson's education

Recommended course text Books

1. Pressman R.S (1997) Software engineering: a practitioner's Approach, McGraw Hill
2. Somerville Ian (2002) software engineering ,Pearson's education
3. Kotonya.G and Somerville, I (1998), Requirements engineering: Processes and Techniques, Wiley
4. Peters, J.F and Pedrycz (2000) Software Engineering: An Engineering Approach, John Wiley and sons.

SAMPLE EXAMINATION
SOFTWARE ENGINEERING

INSTRUCTIONS

SECTION A: THIS SECTION IS COMPULSORY

Question 1.

(a) Differentiate between software engineering and software re-engineering. (4marks)

(b) (i) With the context of software design explain what is meant by the terms cohesion and coupling.(4marks)
(ii) How are the concepts of cohesion and coupling useful in arriving at good software design? (4marks)

(c) State two factors to be considered when selecting a programming language. (4 marks)

(d) The process of software development can be complex hence challenging. Explain how the following techniques are applied in reducing the complexity and minimize the challenge.

(i) Software project management. (4marks)
(ii) Configuration management
(iii) Software quality assurance

(e) Define the following terms: (4marks)
(i) Validation
(ii) Verification

SECTION B: ANSWER ANY TWO QUESTION

Question 2.

(a) List and explain the major responsibilities of a software project manager. (4marks)
(b) Software maintenance has become an important activity of a large number of organizations. Explain the different types of maintenance that a software product management need. (8marks).
(c) Explain the terms CASE tool and CASE environment. (6 marks)

Question 3.

(a) Discuss the following terms (10marks)
(i) Risk management
(ii) Configuration management
(iii) Scheduling
(iv) Software standards.
(v) Software

(b) Explain four major shortcomings that we might face if we use the classical waterfall model for developing all types of software products. (4marks)
(c) Software design has two fundamental different approaches. State and give two advantages of each approach. (6marks).

Question 4.

(a) A software development life cycle is a structure imposed on the development of a software product. Discuss the six activities carried out in software development life cycle. (6marks)
(b) Explain how both the waterfall model of the software development and the prototyping model can be accommodated in the spiral process model. (6marks).
(c) Describe four types of non-functional requirements that may be placed on a system. Give examples of each of these types of requirements. (8marks)

Question 5.

(a) Software testing is one of major approaches in software development. Discuss the five software testing strategies. (10marks)
(b) The goal of the requirements engineering process is to create and maintain a system requirements document. The overall process includes four high level requirements engineering sub-processes. With the aid of a diagram illustrate the relationship between these activities. (10marks)