MODULE 3

INTRODUCTION TO SOFTWARE DESIGN

According to Webster, the **process of design** involves “conceiving and planning out in the mind” and “making a drawing, pattern, or sketch of”. The three types of activities in design are external design, architectural design, and detailed design. Architectural and detailed design are collectively referred to as internal design.

External design of software involves conceiving, planning out, and specifying the externally observable characteristics of a software product. These include user displays, and report formats, external data sources and data sinks, and the functional characteristics, performance requirements, and high level process structure for the product. External design begins during analysis phase and continues into design phase.

Internal design involves conceiving, planning out, and specifying internal structure and processing details of the software product. The goals of internal design are to specify internal structure and processing details, to record decisions and indicate why certain alternatives and trade-offs were chosen, to elaborate the test plan, and to provide a blueprint for implementation, testing, and maintenance activities. The work products of internal design include a specification of architectural structure, the details of algorithms and data structures, and the test plan.

Architectural design is concerned with refining the conceptual view of the system, identifying internal processing functions, decomposing high-level functions into subfunctions, defining internal data streams and data stores, and establishing relationships and interconnections among functions, data streams, and data stores.

Detailed design includes specification of algorithms, concrete data structures, the actual interconnections among functions and data structures, and packaging scheme for the system.

The test plan describes the objectives of testing, the test completion criteria, the integration plan, particular tools and techniques to be used, and the actual test cases and expected results. Functional tests and performance tests are developed during requirements analysis and are refined during the design phase.

Tests that examine the internal structure of the software product and tests that attempt to break the system (stress tests) are developed during detailed design and implementation.

External design and architectural design typically span the period from Software Requirements Review to Preliminary Design Review. Detailed design spans from Preliminary Design Review to Critical Design Review.

FUNDAMENTAL DESIGN CONCEPTS

Every intellectual design is characterized by fundamental concepts and specific techniques. Techniques are applied to particular situations. Techniques come and go with changes in technology, intellectual fads, economic conditions, and social concerns. Fundamental concepts of software design include abstraction, structure, information hiding, modularity, concurrency, verification, and design aesthetics.

Abstraction

Abstraction is the intellectual tool that allows us to deal with concepts apart from particular instances of those concepts. During requirements definition and design, abstraction permits separation of the conceptual aspects of a system from the implementation details.

For example, specify the FIFO property of a queue or the LIFO property of a stack without concern for representation scheme to be used in implementing the stack and queue. Similarly, we can specify the functional characteristics of the routines that manipulate data structures (eg., NEW, PUSH, POP, TOP, EMPTY) without concern for the algorithmic details of the routines.

Abstraction reduces the amount of complexity. Three widely used abstraction mechanisms in software design are functional abstraction, data abstraction, and control abstraction.

Functional abstraction involves the use of parameterized subprograms. The ability to parameterize a subprogram and to bind different parameter values on different invocations of the subprogram is a powerful abstraction mechanism. Functional abstraction can be generalized to collections of subprograms, called “Groups” (Package in Ada, Clusters" in CLU).

Data abstraction involves specifying a data type or a data object by specifying legal operations on objects; representation and manipulation details are suppressed. The term “**data encapsulation**” is used to denote a single instance of a data object defined in terms of the operations that can be performed on it; the term “**abstract data type**” is used to denote declaration of a data type (such as stack) from which numerous instances can be created.

Control abstraction is used to state a desired effect without stating the exact mechanism of control. IF statements and WHILE statements in modern programming languages are abstractions of machine code implementations that involve conditional jump instructions.

Information Hiding

Information hiding is a fundamental design concept for software. It was formulated by Parnas. When a software system is designed using the information hiding approach, each module in the system hides the internal details of its processing activities and modules communicate only through well-defined interfaces.

Functional, data, and control abstraction exhibit information hiding characteristics.

According to Parnas, design should begin with a list of difficult design decisions and design decisions that are likely to change. Each module is designed to hide such a decision from the other modules.

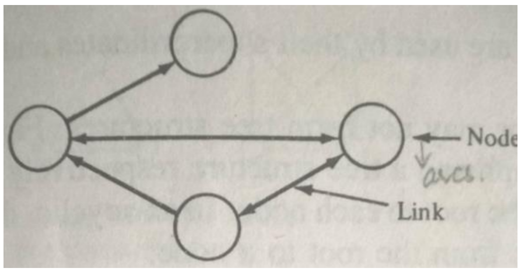
In addition to hiding of difficult and changeable design decisions, other candidates for information hiding include:

1. A data structure, its internal linkage, and the implementation details of the procedures that manipulate it (principle of data abstraction).
2. The format of control blocks such as those for queues in an operating system
3. Character codes, ordering of character sets, and other implementation details
4. Shifting, masking, and other machine dependent details

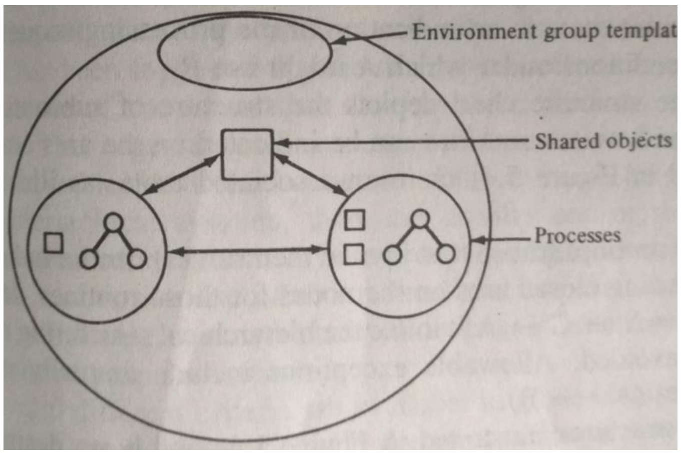
Information hiding can be used as the principal design technique for architectural design of a system, or as a modularization criterion.

Structure

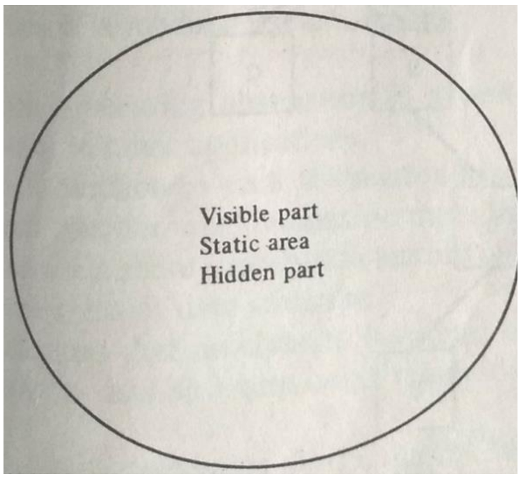
Structure is a fundamental characteristic of computer system. It permits decomposition of a large system into smaller, more manageable units with well-defined relationships to the other units in the system. The most general form of system structure is the network.



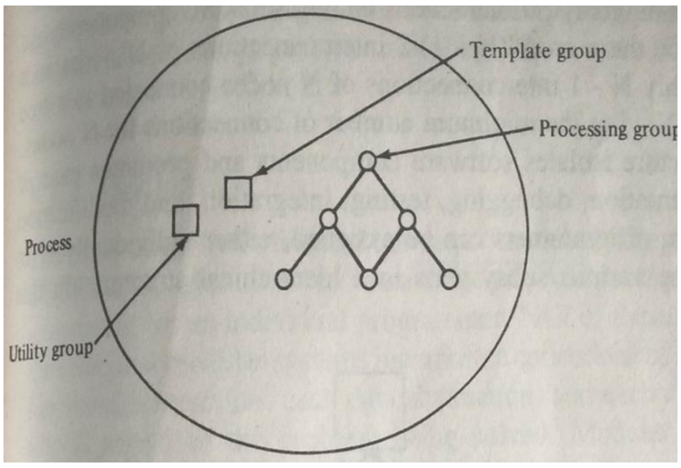
Network



Node



Group



Process

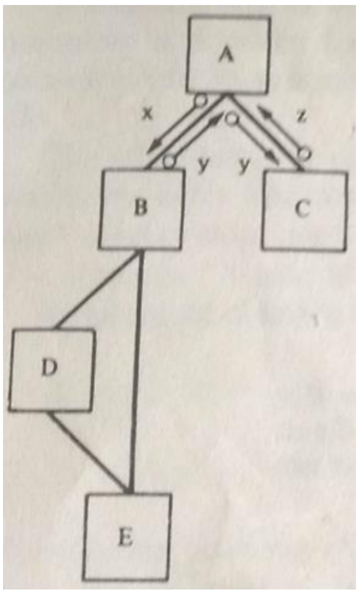
Software System Structure

A **computing network** can be represented as a directed graph, consisting of nodes and arcs. The nodes can represent processing elements that transform data and the arcs can be used to represent data links between nodes. The nodes can represent data stores and the arcs data transformation.

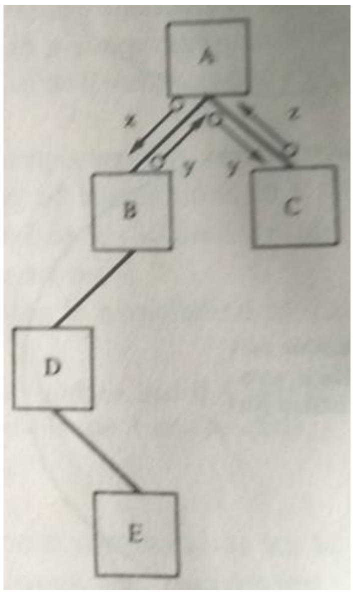
A **network** specify data flow and processing steps within a single subprogram, or the data flow among a collection of sequential subprograms. The most complex form of computing network is a distributed computing system in which each node represents a geographically distinct processor with private memory.

The structure inside a complex processing node consist of concurrent processes executing in parallel and communicating through some combination of shared variables and synchronous and asynchronous message passing.
The relationship "uses" and the complementary relationship "is used by" provide the basis for hierarchical ordering of abstractions in a software system. The "uses" relationship can be represented as a directed graph, where the notation A → B means "A uses B" or "B is used by A". Hierarchical ordering of abstractions is established by the rule:

If A and B are distinct entities, and if A uses B, then B is not permitted to use A or any entity that makes use of A.



A graph structure chart



A tree structure chart

In tree there is a unique path from the root to each node. In an acyclic, directed graph there may be more than one path from the root to a node. These figures are called structure charts.

Modularity

"A module is a FORTRAN subroutine", "a module is an Ada package", "a module is a work assignment for an individual programmer". Modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction, each control abstraction handles a local aspect of the problem being solved.

Modular systems consist of well-defined, manageable units with well-defined interfaces among the units. Desirable properties include:

- Each processing abstraction is a well-defined subsystem that is potentially useful in other applications.
- Each function in each abstraction has a single, well defined purpose.
- Each function manipulates no more than one major data structure.
- Functions share global data selectively. It is easy to identify all routines that share a major data structure.
- Functions that manipulate instances of abstract data types are encapsulated with the data structure being manipulated.

Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

Concurrency

Software systems can be categorized as sequential or concurrent. In a sequential system, only one portion of the system is active at any given time. Concurrent systems have independent processes that can be activated simultaneously if multiple processors are available.

On a single processor, concurrent processes can be interleaved in execution time. This permits implementation of time-shared, multi-programmed, and real-time systems.

Problems unique to concurrent systems include deadlock, mutual exclusion, and synchronization of processes.

- Deadlock is an undesirable situation that occurs when all processes in a computing system are waiting for other processes to complete some actions so that each can proceed.

- Mutual exclusion is necessary to ensure that multiple processes do not attempt to update the same components of the shared processing state at the same time.
- Synchronization is required so that concurrent processes operating at differing execution speeds can communicate at the appropriate points in their execution histories.

Concurrency is a fundamental principle of software design because parallelism in software introduces added complexity and additional degrees of freedom into the design process.

Verification

Verification is a fundamental concept in software design. Design is the bridge between customer requirements and an implementation that satisfies those requirements. A design is verifiable if it can be demonstrated that the design will result in an implementation that satisfies the customer's requirements.

This is typically done in two steps:

(1) Verification that the software requirements definition satisfies the customer's needs (verification of the requirements); and

(2) Verification that the design satisfies the requirements definition (verification of the design).

Aesthetics

Aesthetic considerations are fundamental to design, whether in art or technology. Simplicity, elegance, and clarity of purpose distinguish products of outstanding quality from mediocre products.

MODULES AND MODULARIZATION CRITERIA

Architectural design has the goal of producing well-structured, modular software systems.

Characteristics of software module

- Modules contain instructions, processing logic, and data structures.
- Modules can be separately compiled and stored in a library.
- Modules can be included in a program
- Module segments can be used by invoking a name and some parameters.
- Modules can use other modules.

Examples of modules include procedures, subroutines, and functions; functional groups of related procedures, subroutines, and functions; data abstraction groups; utility groups; and concurrent processes.

Modularization allows the designer to decompose a system into functional units, to impose hierarchical ordering on function usage, to implement data abstractions, and to develop independently useful subsystems.

Advantages

- Modularization can be used to isolate machine dependencies.
- Improve the performance of a software product.
- Ease debugging, testing, integration, tuning, and modification of the system.

Coupling and Cohesion

A fundamental goal of software design is to structure the software product so that the number and complexity of interconnections between modules is minimized. An appealing set of heuristics for achieving this goal involves the concepts of coupling and cohesion.

Coupling and Cohesion were first described by Stevens, Constantine, and Myers.

The strength of coupling between two modules is influenced by the complexity of the interface, the type of connection, and the type of communication.

For example, interfaces established by common control blocks, common data blocks, common overlay regions of memory, common I/O devices, and/or global variable names are more complex (more tightly coupled) than interfaces established by parameter lists passed between modules.

Modification of a common data block or control block may require modification of all routines that are coupled to that block. If modules communicate only by parameters, and if the interfaces between modules remain fixed, the internal details of modules can be modified without having to modify the routines that use the modified modules.

Connections established by referring to other module names are more loosely coupled than connections established by referring to the internal elements of other modules.

Coupling between modules can be **ranked** on a scale of strongest (least desirable) to weakest (most desirable) as follows:

1. Content coupling
2. Common coupling
3. Control coupling
4. Stamp coupling
5. Data coupling

Content coupling occurs when one module modifies local data values or instructions in another module. It can occur in assembly language programs.

In **common coupling**, modules are bound together by global data structures.
Control coupling involves passing control flags (as parameters or globals) between modules so that one module controls the sequence of processing steps in another module.

Stamp coupling is similar to common coupling, except that global data items are shared selectively among routines that require the data.

Data coupling involves the use of parameter lists to pass data items between routines. The most desirable form of coupling between modules is a combination of stamp and data coupling.

The internal cohesion of a module is measured in terms of strength of binding of elements within the module. Cohesion of elements occurs on the scale of weakest (least desirable) to strongest (most desirable) in the following order:

1. Coincidental cohesion
2. Logical cohesion
3. Temporal cohesion
4. Communication cohesion
5. Sequential cohesion
6. Functional cohesion
7. Informational cohesion

Coincidental cohesion occurs when the elements within a module have no apparent relationship to one another. This results when a large, monolithic program is “modularized” b arbitrarily segmenting the program into several small modules, or when a module is created from a group of unrelated instructions that appear several times in other modules.

Logical cohesion implies some relationship among the elements of the module; as for example, in a module that performs all input and output operations, or in a module that edits all data.

Math library routines often exhibit logical cohesion. Logically cohesive modules usually require further decomposition.

For example, a logically cohesive module to process records might be decomposed into four modules to process master records, process update records, process addition records, and process deletion records.

Modules with **temporal cohesion** exhibit many of the same disadvantages as logically bound modules. However, they are higher on the scale of binding because all elements are executed at one time, and no parameters or logic are required tp determine which elements to execute.

A typical example of temporal cohesion is a module that performs program initialization.

The elements of a module possessing **communicational cohesion** refer to the same set of input and/or output data. For example, “Print and Punch the Output File” is communicationally bound.

Communicational binding is higher on the binding scale than temporal binding because the elements are executed at one time and also refer to the same data.

Sequential cohesion of elements occurs when the output of one element is the input for the next element. For example, “Read Next Transaction and Update Master File” is sequentially bound. Sequential cohesion is high on the binding scale because the module structure usually bears a close resemblance to the problem structure.

Functional cohesion is a strong, and hence desirable, type of binding of elements in a module because all elements are related to the performance of a single function. Examples of functionally bound modules are “Compute Square Root”, “Obtain Random Number”, and “Write Record to Output File”.

Informational cohesion of elements in a module occurs when the module contains a complex data structure and several routines to manipulate the data structure. Each routine in the module exhibits functional binding. Informational cohesion is the concrete realization of data abstraction.

Other Modularization Criteria

Additional criteria for deciding which functions to place in which modules of a software system include: hiding difficult and changeable design decisions; limiting the physical size of modules; structuring the system to improve observability and testability; isolating machine dependencies to a few routines; easing likely changes; providing general-purpose utility functions; developing an acceptable overlay structure in a machine with limited memory capacity; minimizing page faults in a virtual memory machine; and reducing the call-return overhead of excessive subroutine calls. For each software product, the designer must weigh these factors and develop a consistent set of modularization criteria to guide the design process.

DESIGN NOTATIONS

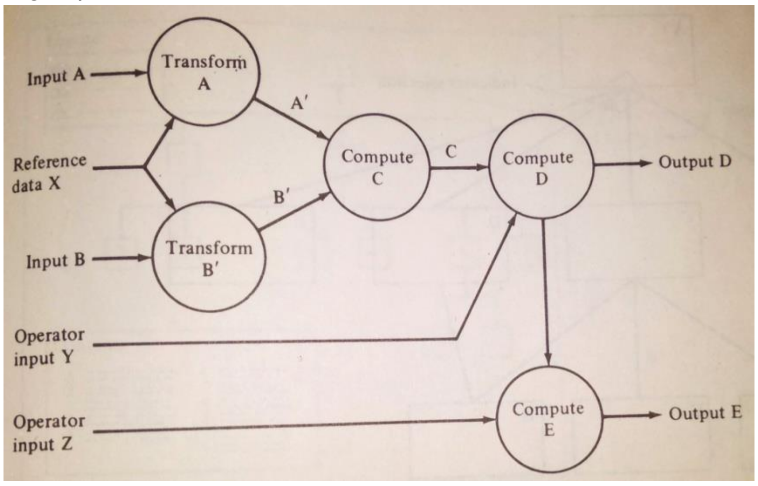
In software design, the representation schemes used are of fundamental importance. Good notations clarify interrelationships & interactions, while poor notations complicate and interfere with good design practice. At least three levels of design specifications exists: **External Design Specifications**, which describe external characteristics of software system; **Architectural Design Specifications**, which describe structure of system; and **Detailed Design Specifications**, which describe control flow, data representations, & other algorithmic details within modules.

Notations used to specify external characteristics, architectural structure & processing details of a software system include Data Flow Diagram, Structure Charts, HIPO Diagrams, Procedure Templates, Pseudocode, Structured Flowcharts, Structured English, and Decision Tables.

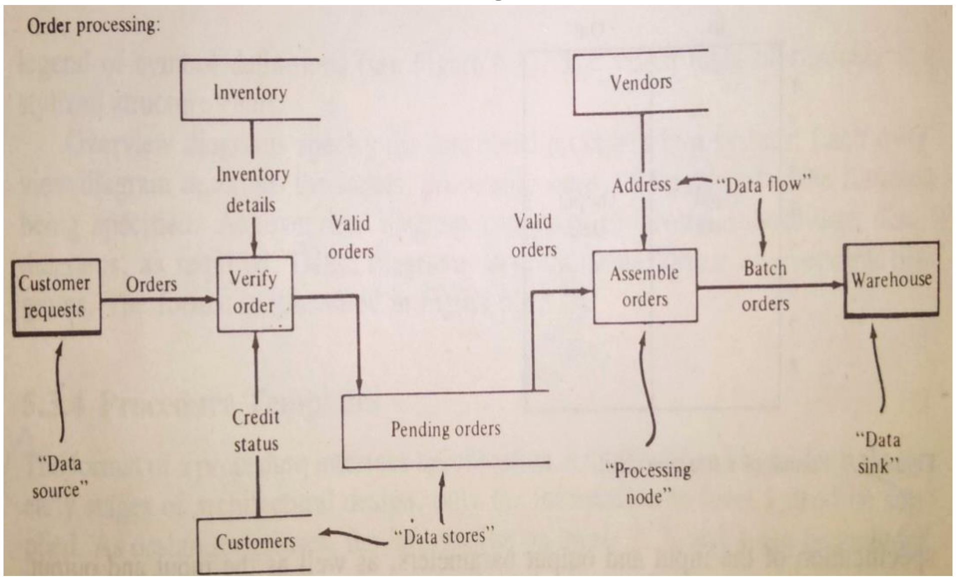
Data Flow Diagrams (DFD)

Data Flow Diagrams (**Bubble Charts**) are directed graphs in which nodes represent processing activities and the arcs specify data items transmitted between processing nodes. Like flowcharts, DFDs can be used at any level of abstraction. A DFD represent data flow between individual statements or blocks of statements, sequential routines, concurrent processes, or distributed computing system where each node represents geographically remote processing unit.

Unlike flowcharts, DFDs do not indicate decision logic or conditions. DFDs can be expressed using informal notation, or special symbols. DFDs are excellent mechanisms for communicating with customers during requirements analysis; they are also widely used for representation of external & top-level internal design specifications.



An Informal Data Flow Diagram or "Bubble Chart"

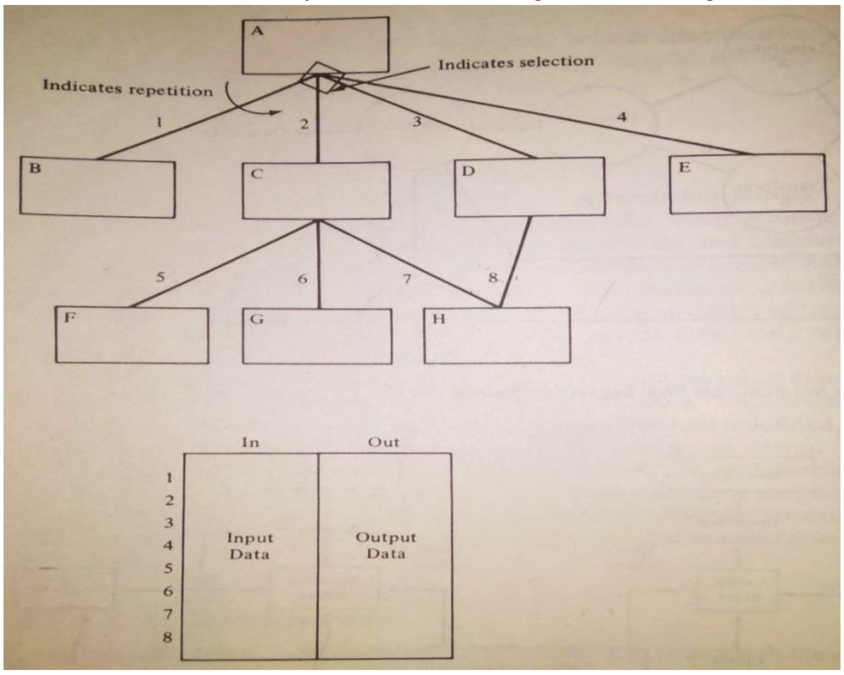


A Formal Data Flow Diagram

Structure Charts

Structure charts are used during architectural design to document hierarchical structure, parameters, & interconnections in a system. A structure chart differs from a flowchart in two ways: a structure chart has no decision boxes, and the sequencing order of tasks inherent in a flowchart can be suppressed in a structure chart.

The structure of a hierarchical system can be specified using a structure chart. The chart can be augmented with module-by-module specification of the input and output parameters, as well as the input and output parameterattributes. During architectural design the parameter attributes are abstract; they are refined into concrete representations during detailed design.



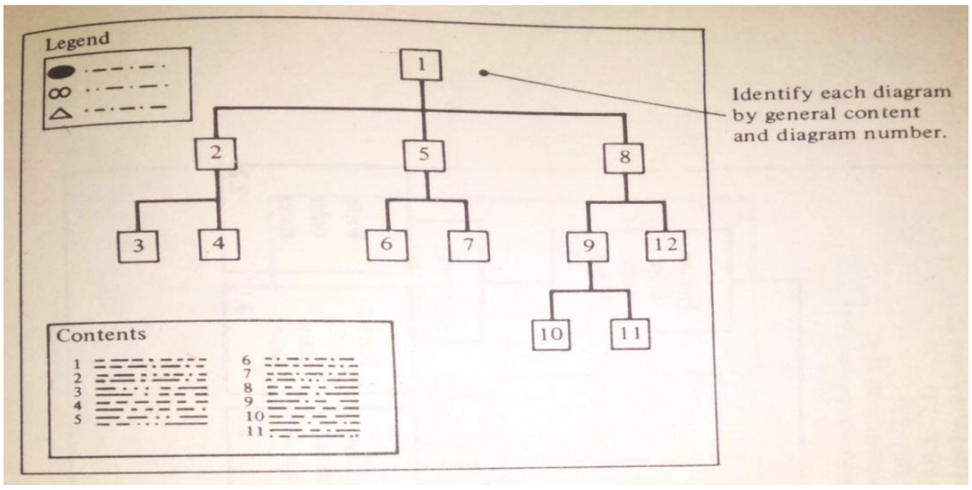
Format of a Structure Chart

HIPO Diagrams

HIPO diagrams (Hierarchy-Process-Input-Output) were developed at IBM as design representation schemes for top-down software development; and as external documentation aids for released products.

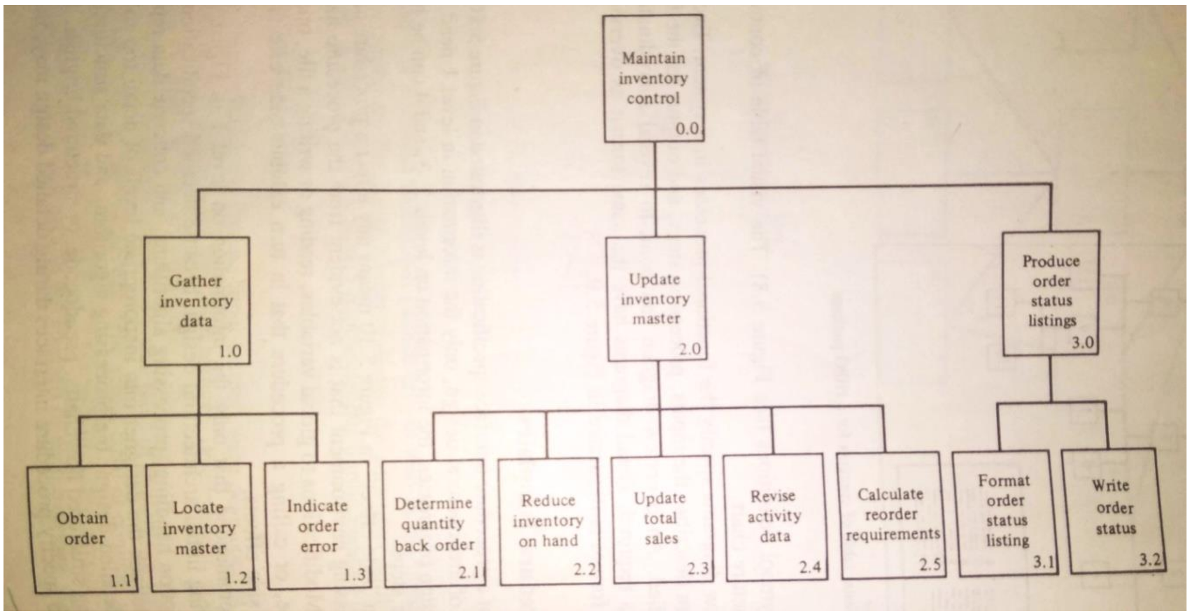
A set of HIPO diagrams contains a visual table of contents, a set of overview diagrams, and a set of detail diagrams. The visual table of contents is a directory to the set of diagrams in the package; it consists of tree-structured (or graph structured) directory, a summary of the contents of each overview diagram, and a legend of symbol definitions.

The visual table of contents is a stylized structure chart. Overview diagrams specify functional processes in a system. Each overview diagram describes the inputs, processing steps, & outputs for the function being specified



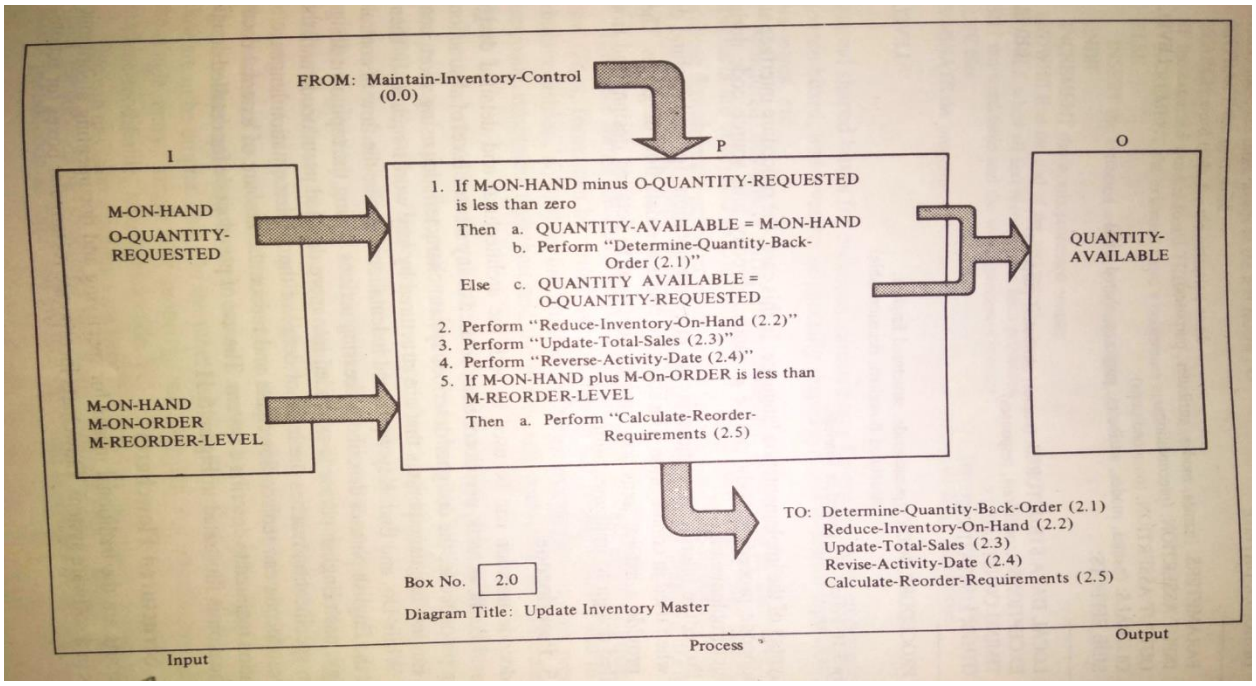
Legend
Contents
1
2
3
4
5
6
7
8
9
10
11
12
Identify each diagram by general content and diagram number.

Visual Table of Contents for a HIPO Package



Maintain inventory control 0.0
Gather inventory data 1.0
Update inventory master 2.0
Produce order status listings 3.0
Obtain order 1.1
Locate inventory master 1.2
Indicate order error 1.3
Determine quantity back order 2.1
Reduce inventory on hand 2.2
Update total sales 2.3
Revise activity data 2.4
Calculate reorder requirements 2.5
Format order status listing 3.1
Write order status 3.2

HIPO Table of Contents


FROM: Maintain-Inventory-Control (0.0)
I
P
O
M-ON-HAND
O-QUANTITY-REQUESTED
M-ON-HAND
M-ON-ORDER
M-REORDER-LEVEL
1. If M-ON-HAND minus O-QUANTITY-REQUESTED is less than zero
Then a. QUANTITY-AVAILABLE = M-ON-HAND
b. Perform "Determine-Quantity-Back-Order (2.1)"
Else c. QUANTITY AVAILABLE = O-QUANTITY-REQUESTED
2. Perform "Reduce-Inventory-On-Hand (2.2)"
3. Perform "Update-Total-Sales (2.3)"
4. Perform "Reverse-Activity-Date (2.4)"
5. If M-ON-HAND plus M-On-ORDER is less than M-REORDER-LEVEL
Then a. Perform "Calculate-Reorder-Requirements (2.5)
QUANTITY-AVAILABLE
TO: Determine-Quantity-Back-Order (2.1)
Reduce-Inventory-On-Hand (2.2)
Update-Total-Sales (2.3)
Revise-Activity-Date (2.4)
Calculate-Reorder-Requirements (2.5)
Box No. 2.0
Diagram Title: Update Inventory Master
Input
Process
Output

A HIPO Detail Diagram

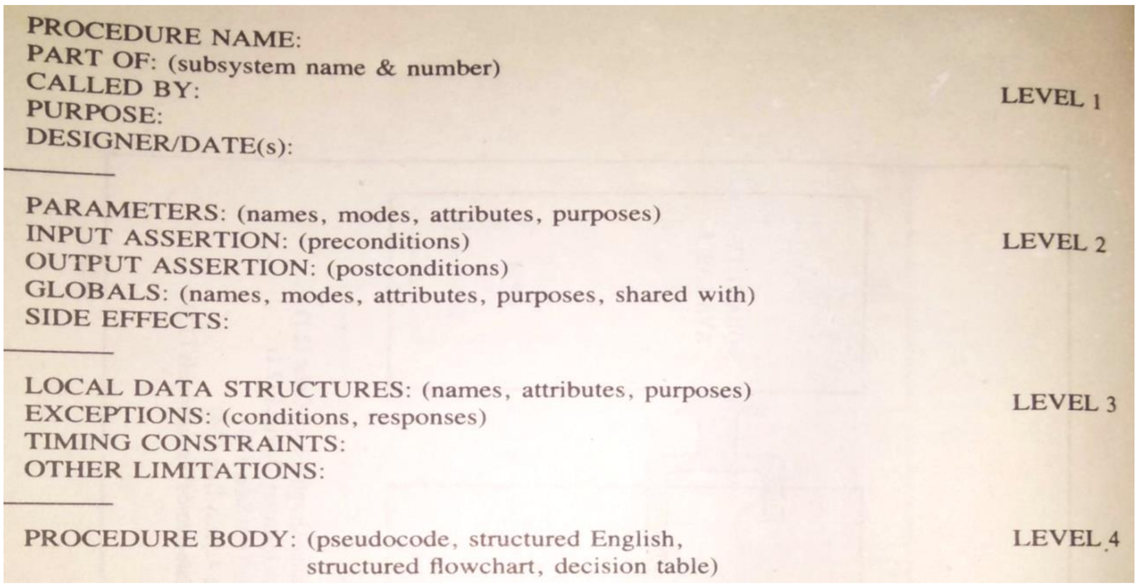
Procedure Templates
In the early stages of architectural design, only the information in level 1 need be suppressed. As design progresses, the information on levels 2, 3, and 4 can be included in successive steps.

The term "**Side Effect**" means any effect a procedure can exert on the processing environment that is not evident from the procedure name and parameters. Modifications to global variables, reading or writing a file, opening or closing a file, or calling a procedure that in turn exhibits side effects are all examples of side effects.

It is recommended that only the information on level 1 be provided during initial architectural design, because detailed specification of side effects, exception handling, processing algorithms, and concrete data representations will sidetrack the designer into inappropriate levels of detail too soon.

During detailed design, the processing algorithms and data structures can be specified using structured flowcharts, pseudocode, or structured English.

Procedure interface specifications are effective notations for architectural design when used in combination with structure charts and DFDs. They also provide a natural transition from architectural to detailed design, and from detailed design to implementation.



PROCEDURE NAME:
PART OF: (subsystem name & number)
CALLED BY: LEVEL 1
PURPOSE:
DESIGNER/DATE(s):

PARAMETERS: (names, modes, attributes, purposes)
INPUT ASSERTION: (preconditions) LEVEL 2
OUTPUT ASSERTION: (postconditions)
GLOBALS: (names, modes, attributes, purposes, shared with)
SIDE EFFECTS:

LOCAL DATA STRUCTURES: (names, attributes, purposes)
EXCEPTIONS: (conditions, responses) LEVEL 3
TIMING CONSTRAINTS:
OTHER LIMITATIONS:

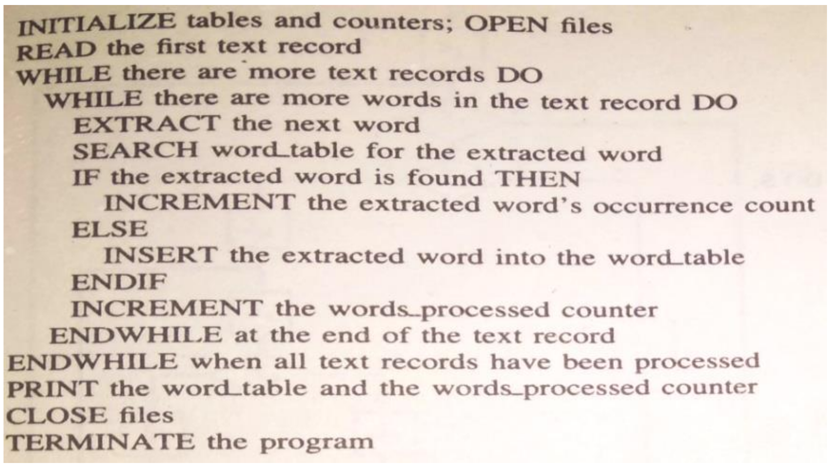
PROCEDURE BODY: (pseudocode, structured English, structured flowchart, decision table) LEVEL 4

Format of a Procedure Template

Pseudocode

Pseudocode notation can be used in both architectural & detailed design phases. Like flowcharts, pseudocode can be used at any desired level of abstraction. Unlike pseudocode, the designer describes system characteristics using short, concise, English language phrases (keywords such as If-Then-Else, While-Do, and End). Keywords & Indentation describe flow of control.

Pseudocode can replace flowcharts & reduce amount of external documentation to describe a system.



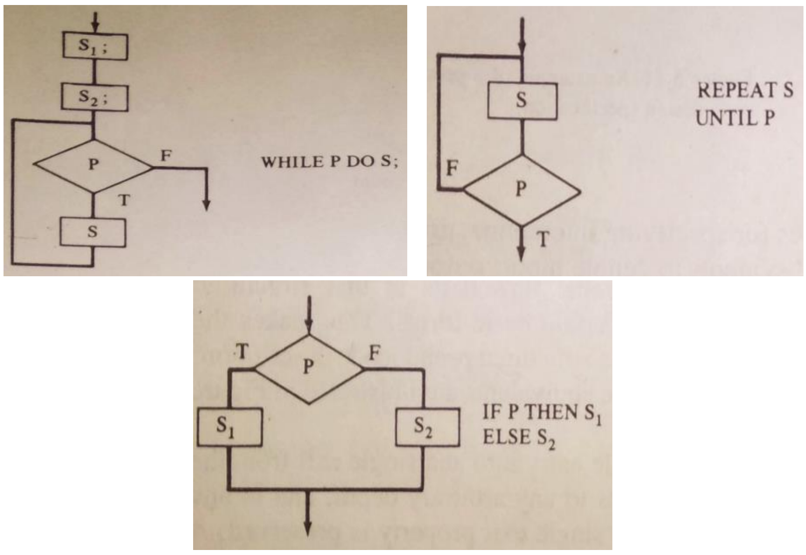
```
INITIALIZE tables and counters; OPEN files
READ the first text record
WHILE there are more text records DO
  WHILE there are more words in the text record DO
    EXTRACT the next word
    SEARCH word_table for the extracted word
    IF the extracted word is found THEN
      INCREMENT the extracted word's occurrence count
    ELSE
      INSERT the extracted word into the word_table
    ENDIF
    INCREMENT the words_processed counter
  ENDWHILE at the end of the text record
ENDWHILE when all text records have been processed
PRINT the word_table and the words_processed counter
CLOSE files
TERMINATE the program
```

An Example of a Pseudocode Design Specifications

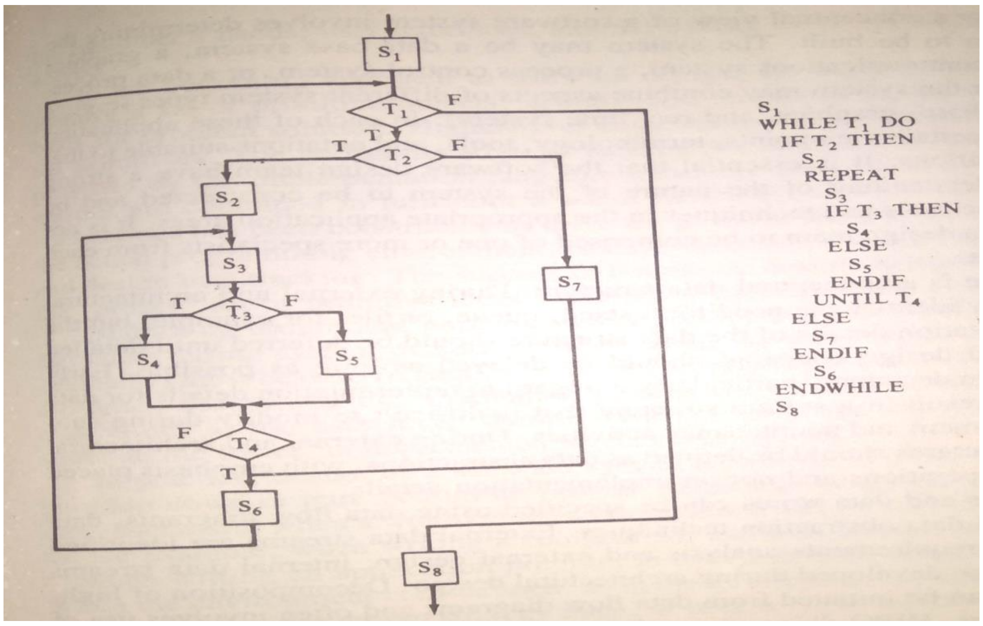
Structured Flowcharts

Flowcharts are the traditional means for specifying and documenting algorithmic details in a software system. Flowcharts incorporate rectangular boxes for actions, diamond shaped boxes for decisions directed arcs for specifying interconnections between boxes, and a variety of specially shaped symbols to denote input, output, data stores, etc.

The basic forms are characterized by single entry into and single exit from the form. Thus, forms can be nested within forms to any arbitrary depth. Structured flowcharts are logically equivalent to pseudocode.



Basic Forms for Structured Flowchart



```
$S_1$
WHILE $T_1$ DO
  IF $T_2$ THEN
    $S_2$
    REPEAT
      $S_3$
      IF $T_3$ THEN
        $S_4$
      ELSE
        $S_5$
      ENDIF
    UNTIL $T_4$
  ELSE
    $S_7$
  ENDIF
  $S_6$
ENDWHILE
$S_8$
```

A Structured Flowchart and Pseudocode Equivalent

Structured English

Structured English can be used to provide a step-by-step specification for an algorithm. Like pseudocode, structured English can be used at any desired level of detail. For example, Specify cookbook recipe:

1. Preheat oven to 350 degrees
2. Mix egg, milk & vanilla
3. Add flour & baking soda
4. Pour into a greased baking dish
5. Cook until done

Decision Tables

Decision tables can be used to specify complex decision logic in a high-level software specification. They are also useful for specifying algorithmic logic during detailed design. Decision tables can be specified and translated into source code logic.

DESIGN TECHNIQUES
The design process involves developing a conceptual view of the system, establishing system structure, identifying data streams and data stores, decomposing high-level functions into subfunctions, establishing relationships and relationships and interconnections among components, developing concrete data representations, and specifying algorithmic details.

Developing a conceptual view of a software system involves determining the type of system to be built. The system may be a database system, a graphics system, a telecommunications system, a process control system, or a data processing system; or the system may combine aspects of different system types (eg. a combined database, graphics, and real time system).

In each of these application areas there are certain viewpoints, terminology, tools, and notations suitable to that class of applications. It is essential that the software design team have a strong conceptual understanding of the nature of the system to be constructed and be familiar with the tools and techniques in the appropriate application areas.

Design techniques are typically based on the “top-down” and/or “bottom-up” design strategies. Using the top-down approach, attention is first focused on global aspects of the overall system. As the design progresses, the system is decomposed into subsystems and more consideration is given to specific issues. Backtracking is fundamental to top-down design.

The primary **advantage** of the top-down strategy is that attention is first directed to the customer’s needs, interfaces, and the overall nature of the problem being solved.

In the bottom-up approach to software design, the designer first attempts to identify a set of primitive objects, actions, and relationships that will provide a the basis for problem solution.

Bottom-up design may also require redesign and design backtracking. The success of bottom-up design may depends on identifying the “proper” set of primitive ideas sufficient to implement the system.

Stepwise Refinement
Stepwise refinement is a top-down technique for decomposing a system from high-level specifications into more elementary levels. Stepwise refinement is also known as “**Stepwise Program Development**” and “**Successive Refinement**”.

As originally described by Wirth, Stepwise Refinement involves the following activities:

1. Decomposing design decisions to elementary levels.
2. Isolating design aspects that are not truly interdependent.
3. Postponing decisions concerning representation details as long as possible.
4. Carefully demonstrating that each successive step in the refinement process is a faithful expansion of previous steps.

Incremental addition of detail at each step in the refinement process postpones design decisions. Stepwise refinement begins with the specifications derived during requirements analysis and external design. The problem is first decomposed into a few major processing steps that will decomposed solve the problem.

An explicit representation technique is not prescribed in stepwise refinement. However, use of structure charts, procedure specifications, and pseudocode is consistent with successive refinement. Successive refinement can be used to perform detailed design of the individual modules in a software product.

The major **benefits** of stepwise refinement as a design technique are:

1. Top–down decomposition
2. Incremental addition of detail
3. Postponement of design decisions
4. Continual verification of consistency (formally or informally)

Using stepwise refinement, a problem is segmented into small, manageable pieces, and the amount of detail that must be dealt with at any particular time is minimized.

Levels of Abstraction

Levels of abstraction was originally described by Dijkstra as a bottom-up design technique in which an operating system was designed as a layering of hierarchical levels starting at level 0 (processor allocation, real-time clock interrupts) and building up to the level of processing independent user programs.

In Dijkstra's system (T.H.E. system, that is Technische Hogeschool Eindhoven in Dutch language, an Eindhoven Univeristy of Technology, Netherland), each level of abstraction is composed of a group of related functions, some of which are externally visible (can be invoked by functions on higher levels of abstraction) and some of which are internal to the level.

Internal functions are hidden from other levels; they can only be invoked by functions on the same level. The internal functions are used to perform tasks common to the work being performed on that level of abstraction.

Each level of abstraction performs a set of services for the functions on the next higher level of abstraction. Each level of abstraction has exclusive use of certain resources (I/O devices, data structures) that other levels are not permitted to access.

The strict hierarchical ordering of routines facilitates "intellectual manageability" of a complex software System.

The levels of abstraction utilized in the T.H.E. operating system

Level 0: Processor allocation and clock interrupt handling
Level 1: Memory segment controller
Level 2: Console message interpreter
Level 3: I/O buffering

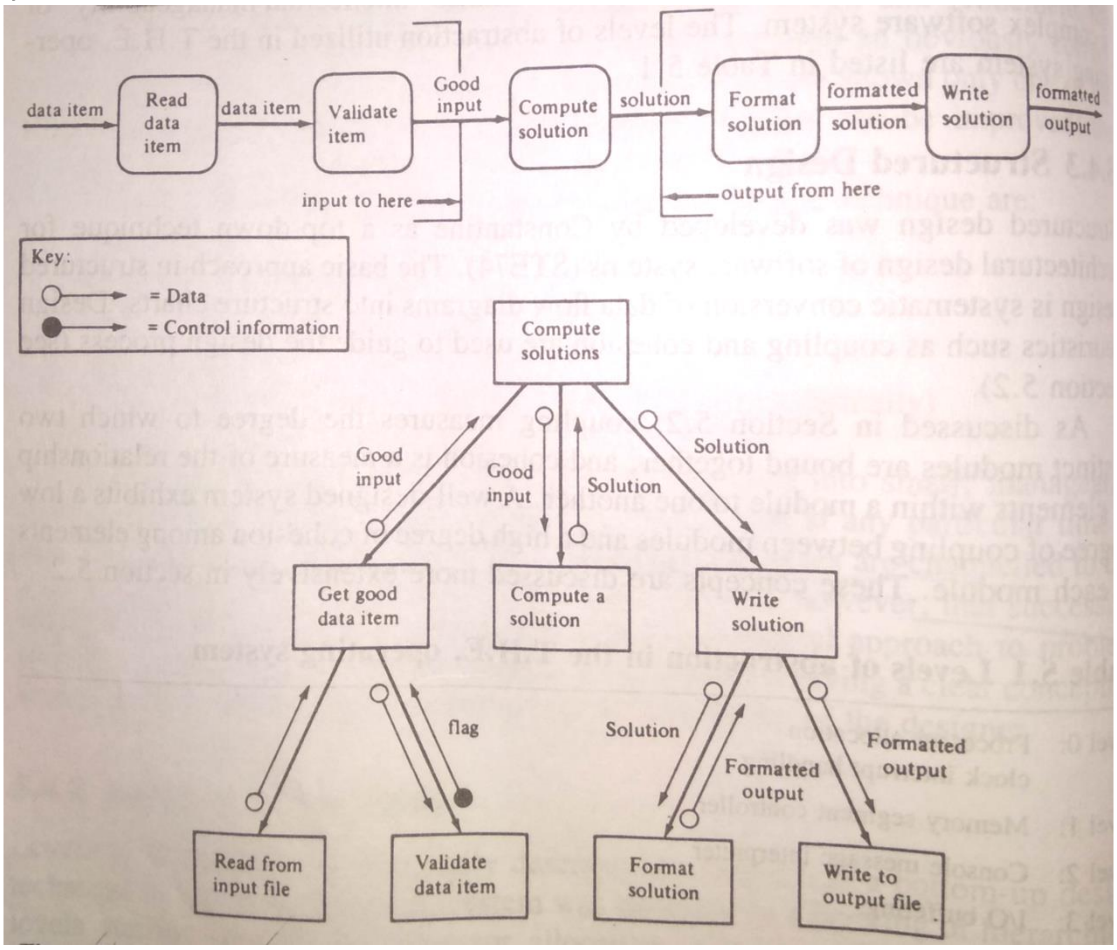
Level 4: User programs
Level 5: Operator

Structured Design

Structured design was developed by Constantine as a top down as a top-down technique for architectural design of software systems. The basic approach in structured design is systematic conversion of data flow diagrams into basic approach in structured design is systematic conversion of data flow diagrams into structure charts. Design heuristics such as coupling and cohesion are used to guide the design process.

Coupling measures the degree to which two distinct modules are bound together, and cohesion is a measure of the relationship of elements within a module to one another. A well–designed system exhibits a low degree of coupling between modules and a high degree of cohesion among elements in each module.

The first step in structured design is review and refinement of the data flow diagram(s) developed during requirements definition and external design. The second step is to determine whether the system is transform-centered or transaction driven, and to derive a high-level structure chart based on this determination, and transform-centered that are converted into Input, processing, and Output subsystems in the structure chart.



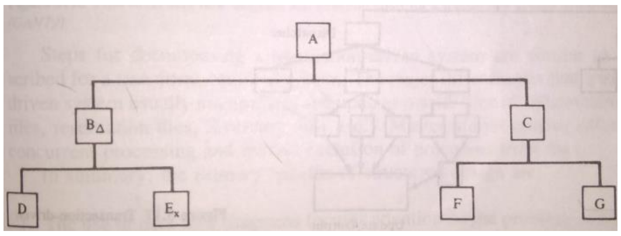
Conversion of a transform-centered data flow diagram into an input, process, output structure chart

The point of most abstract input data is the point in the data flow diagram where the input of stream can no longer be identified.

The third step is decomposition of each subsystem using guidelines such as coupling, cohesion, information hiding, levels of abstraction, data abstraction, and other decomposition criteria.

In addition to coupling, cohesion, data abstraction, information hiding, and other decomposition criteria, the concepts of “scope of effect” and “scope of control” can be used to determine the relative positions of modules in a hierarchical framework.

The “**scope of control**” of a module is that module plus all modules that are subordinate to it in the structure chart.



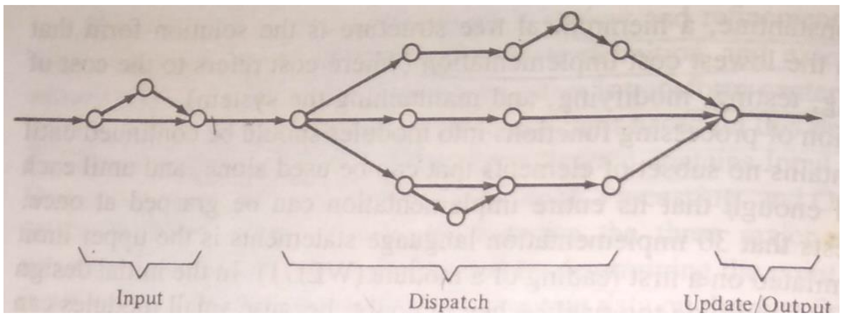
A hierarchical structure chart

The scope of control of module B is B, D, and E.

The "**Scope of Effect**" of a decision is the set of all modules that contain code that is executed based on the outcome of that decision. Suppose execution of some code in mode B depends on the outcome of a decision, X, in module E. Either E will return a control flag to B or the decision process will have to be repeated in B.

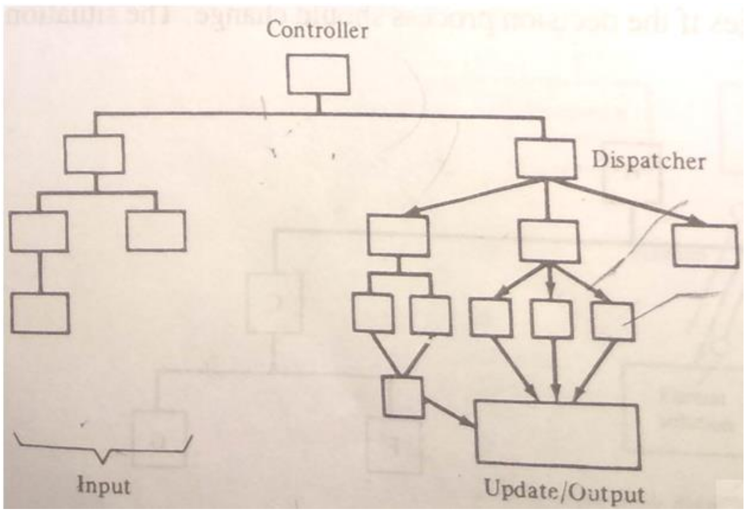
A transform-centered system is characterized by similar processing steps for each data item processed by the Input, Process, and Output subsystems. In a transaction-driven system, one of several possible paths through the data flow diagram is traversed by each transaction.

The path traversed is typically determined by user input commands. Transaction-drive systems have data flow diagrams of the form as



Transaction-driven data flow diagram

Which is converted into a structure chart having Input, Controller, Dispatcher, and Update/Output subsystems as



Transaction-driven structure chart

The primary **benefits** are:

1. The use of DFD focuses attention on the problem structure. This follows naturally from requirements analysis and external design.

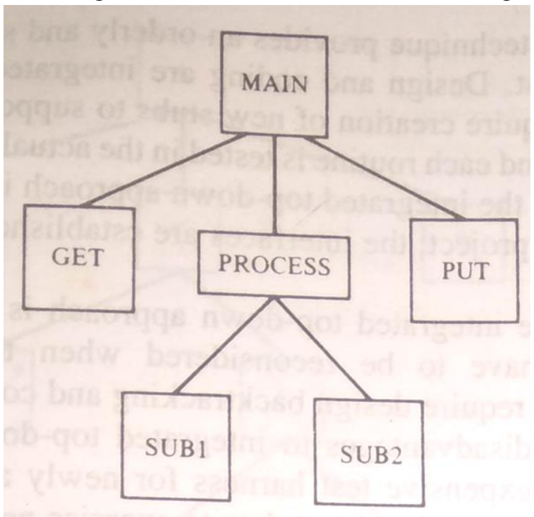
2. The method of translating DFDs into structure charts provides a method for initiating architectural design in a systematic manner.
3. Data dictionaries can be used in conjunction with structure charts to specify dta attributes and data relationships.
4. Design heuristics such as coupling and cohesion, and scope of effect and scope of control provide criteria for systematic development of architectural structure and for comparison of alternative design structures.
5. Detailed design techniques and notations such as successive refinement, HIPO diagrams, procedure specification forms, and pseudocode can be used to perform detailed design of the individual modules.

The primary **disadvantage** of structured design is that the technique produces systems that are structured as sequences of processing steps.

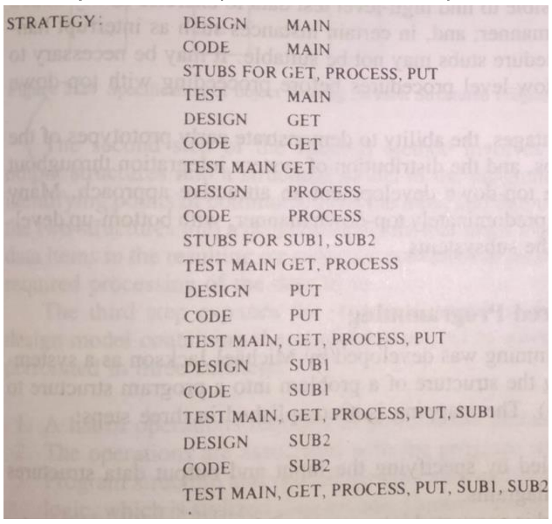
Integrated Top-Down Development

Integrated top-down development integrates design, implementation, and testing. Using integrated top-down development, design proceeds top-down from the highest-level routines; they have the primary function of coordinating and sequencing the lower-level routines.

Lower-level routines may be implementation of elementary functions (those that call no other routines), or they may in turn involve more primitive routines. There is thus a hierarchical structure to a top-down system in which routines can invoke lower-level routines but cannot invoke routines on a higher level.The integration of design, implementation, and testing is illustrated as follows: The design



of a system has proceeded to the point illustrated as



```
STRATEGY:  DESIGN     MAIN
           CODE       MAIN
           STUBS FOR GET, PROCESS, PUT
           TEST       MAIN
           DESIGN     GET
           CODE       GET
           TEST MAIN GET
           DESIGN     PROCESS
           CODE       PROCESS
           STUBS FOR SUB1, SUB2
           TEST MAIN GET, PROCESS
           DESIGN     PUT
           CODE       PUT
           TEST MAIN, GET, PROCESS, PUT
           DESIGN     SUB1
           CODE       SUB1
           TEST MAIN, GET, PROCESS, PUT, SUB1
           DESIGN     SUB2
           CODE       SUB2
           TEST MAIN, GET, PROCESS, PUT, SUB1, SUB2
```

Integrated top-down development strategy

The purpose of procedure MAIN is to coordinate and sequence the GET, PROCESS, and PUT routines. These three routines can communicate only through MAIN; similarly, SUB1 and SUB2 (which support PROCESS), can communicate only through PROCESS.

The stubs referred to dummy routines written to simulate subfunctions that are invoked higher-level functions. As coding and testing progresses, the stubs are expanded into full functional units that may in turn require lower-level stubs to support them.

The integrated top-down design technique provides an orderly and systematic framework for software development. Design and coding are integrated because expansion of a stub will typically require creation of new stubs to support it. Test cases are developed systematically, and each routine is tested in the actual operating environment.

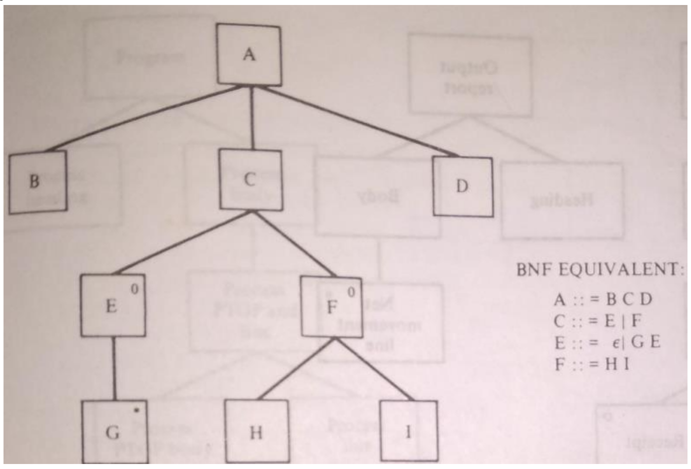
A further **advantage** is distribution of system integration across the project; the interfaces are established, coded, and tested as the design progresses. The primary **disadvantage** of the integrated top-down approach is that early high-level design decisions may have to be reconsidered when the design progresses to lower levels. This may require design backtracking and considerable rewriting of code.

These are other **disadvantages**: The system may be a very expensive test harness for newly added procedures; it may not be possible to find high-level test data to exercise newly added procedures in the desired manner; and, in certain instances such as interrupt handlers and I/O drivers, procedure stubs may not be suitable. It may be necessary to first write and test some low-level procedures before proceeding with top-down development.

Jackson Structured Programming

1. Jackson Structured Programming was developed by Michael Jackson as a systematic technique for mapping the structure of a problem into a program structure to solve the problem. The mapping is accomplished in three steps: The problem is modeled by specifying the input and output data structures using tree structured diagrams.
2. The input-output model is converted into a structural model for the program by identifying points of correspondence between nodes in the input and output trees.
3. The structural model of the program is expanded into a detailed design model that contains the operations needed to solve the problem.

Input and output structures are specified using a graphical notation to specify data hierarchy, sequences of data, repetition of data items, and alternate data items.



Specification of object A using Jackson Structured Programming notation

According to this notation, item A consists of a B followed by a C followed by a D (reading left to right on the same level). B and D have no substructures. C consists of either an E or an F (denoted by “o”). E consists of zero or more occurrences of G (denoted by “*”), and F consists of an H followed by an I.

This notation is the graphical equivalent of regular expressions. The formats of input and output data structures are thus specified using graphical representations of regular grammars.

The second step of the Jackson method involves converting the input and output structures into a structural model of the program. This is accomplished by identifying points of commonality in the input and output structures and combining the two structures into a program structure that maps inputs into outputs.

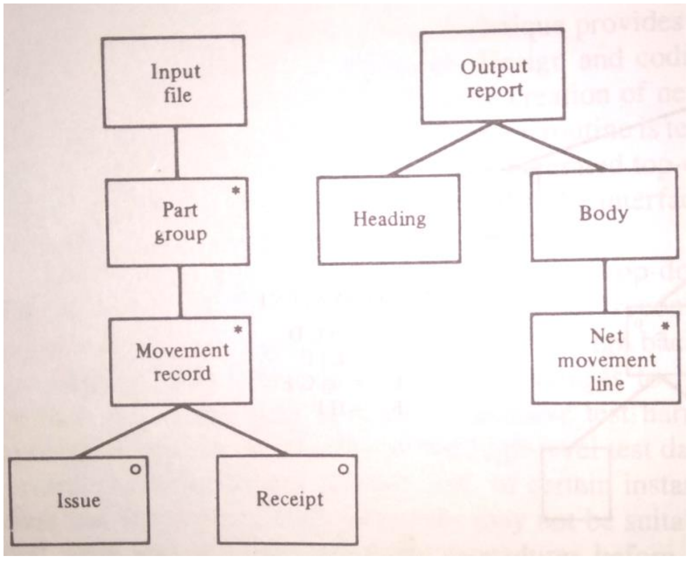
Labels on data items in the resulting structure are converted to process names that perform the required processing of the data items.

The third step expands the structural model of the program into a detailed design model containing the operations needed to solve the problem. This step is performed in three substeps:

1. A list of operations required to perform the processing steps is developed.
2. The operations are associated with the program structure.
3. Program structure and operations are expressed in a notation called **Schematic Logic**, which is stylized pseudocode. Control flow for selection and iteration are specified in this step.

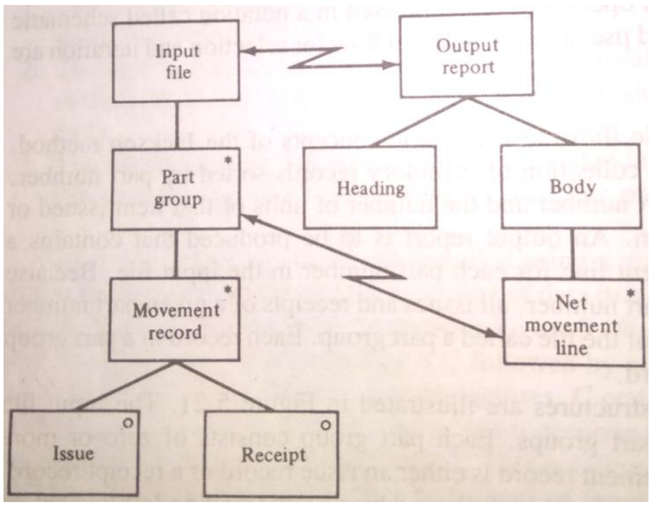
The following example illustrates the basic concept of the Jackson method. An input file consists of a collection of inventory records sorted by part number. Each record contains a part number and the number of units of that item issued or received in one transaction.

An output report is to be produced that contains a heading and a net movement line for each part number in the input file. Because the input file is sorted by part number, all issues and receipts of a given part number are in a contiguous portion of the file called a **part group**. Each record in a part group is called a **movement record**.



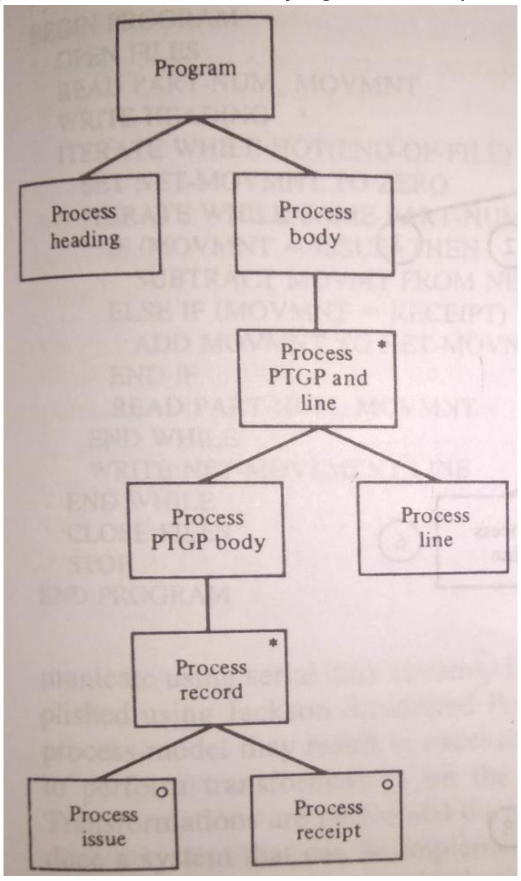
Input and output structures for an inventory problem

The input file consists of zero or more part groups. Each part group consists of zero or more movement records. A movement record is either an issue record or a receipt record. The output report consists of a heading followed by a body (reading left to right on the same level). The body consists of zero or more net movement lines.



Correspondence between input and output structure for an inventory problem

The input file corresponds to the output report, and each part group in the input file corresponds to a net movement line in the output report. The program structure is derived by superimposing the input file structure on the output report structure and overlaying the corresponding nodes in the two graphs.



Program structure for an inventory problem

The program consists of a number of processing steps. There is a processing step to write the report heading, followed by a step to write the report body. The body consists of repetitive invocation of PTGP & line (part group and line), one invocation per part group in the input file. PTGP & line contains a processing step for a part group followed by a step to print the net movement line for that part group.

The part group body consists of a processing step that is invoked once for each part record in the part group. Each invocation of process record processes an issue or a receipt.

More recently, Michael Jackson has developed a method for software design called the **Jackson Design Method**. This approach involves modeling the real world phenomenon of interest as a network of sequential processes that communicate using serial data streams.
Jackson structured programming is widely **used** and is quite effective in situations where input and output data structures can be defined in a precise manner. It appears to be most effective in data processing applications. The utility of the Jackson Design Method is yet to be determined; more experience with the method is required.

DETAILED DESIGN CONSIDERATIONS

Detailed design is concerned with specifying algorithmic details, concrete data representations, interconnections among functions and data structures, and packaging of the software product. Detailed design is strongly influenced by the implementation language. Detailed design is more concerned with semantic issues and less concerned with syntactic details than is implementation.

The starting point for detailed design is an architectural structure. Implementation addresses issues of programming language syntax, coding style, internal documentation, and insertion of testing and debugging probes into the code. Detailed design permits design of algorithms and data representations at a higher level of abstraction and notation. Detailed design separates the activity of low-level design from implementation.

Product packaging is an important aspect of detailed design. Packaging is concerned with the manner in which global data items are selectively shared among program units, specification of static data areas, packaging of program units as functions and subroutines, specification of parameter passing mechanisms, file structures and file access techniques, and the structure of compilation units and load modules.

Detailed design should be carried to a level where each statement in the design notation will result in a few (less than 10) statements in the implementation language. Given the architectural and detailed design specifications, any programmer familiar with the implementation language should be able to implement the software product.

REAL-TIME AND DISTRIBUTED SYSTEM DESIGN

According to Franta, a **distributed system** consists of a collection of nearly autonomous processors that communicate to achieve a coherent computing system. Each processor possesses a private memory, and processors communicate through an interconnection network.

Major issues to be addressed in designing a distributed system include specifying the topology of the communication network, establishing rules for accessing the shared communication channel, allocating application processing functions to processing nodes in the network, and establishing rules for process communication and synchronization.

The design of distributed systems is further complicated by the need to allocate network functionality between hardware and software components of the network. For example, trade-offs of costs and complexity between hardware and software components of network interconnection devices are not obvious.

Message traffic between nodes must be analyzed to establish the necessary communication rates. Reliability issues such as coping with loss of a communication link or loss of a processing node must be considered. Mechanisms for message flow control, error control in response to failures of redundancy checks in arriving messages, systems status monitoring, and network diagnostic techniques must be considered.

Mechanisms for addressing processes in remote nodes, queue management in the network interconnection devices, message flow control between nodes, allocation of the communication network to various nodes, messages parity error checks, and system status monitoring must all be specified.

Real-Time Systems must provide specified amounts of computation within fixed time intervals. Real-time systems typically sense and control external devices, respond to external events, and share processing time between multiple tasks. Processing demands are both Cyclic and Event-driven.

Event-driven activities may occur in bursts, thus requiring a high ratio of peak to average processing. Real-time systems often form distributed networks; local processors may be associated with sensing devices and actuators.

A real-time network for **process control** may consist of several minicomputers and microcomputers connected to one or more large processors. Each small processor may be connected to a cluster of real-time devices.

Decomposition criteria for distributed real-time systems include the need to maintain process simplicity and to minimize inter-process communication bandwidths by communicating simple processed messages rather than raw data.

Process control systems often utilize communication networks having fixed, static topology and known capacity requirements. Many process control systems are designed with only two levels of abstraction, which comprise basic system functions and application programs.

Petri nets are a fundamental state-oriented notation that can be used to specify requirements and high level design of real-time and distributed systems. Petri nets were developed because traditional finit state mechanisms are not adequate for specifying parallel and concurrent system properties.

In summary, the traditional considerations of hierarchy, information hiding, and modularity are important concepts for the design of real-time systems which are typically applied to the individual components of a real-time system. Higher-level issues of networking, performance, and reliability must be analyzed and designed before the component nodes or processes are developed.

TEST PLANS

The test plan is an important product of the software design. A test plan describes various kinds of activities that will be performed to demonstrate that the software product meets it requirements.

The test plan specifies the objectives of testing (eg., to achieve error-free operation under stated conditions for a stated period of time), the test completion criteria (to achieve a specified rate of error exposure, to achieve a specified percent of logical path coverage), the system integration plan (strategy, schedule, responsible individuals), methods to be used on particular modules (walkthroughs, inspections, static analysis, dynamic tests, formal verification), and the particular test cases to be used.

There are four types of tests that a software product must satisfy:

1. Functional test
2. Performance tests
3. Stress tests
4. Structural tests

Functional tests and performance tests are based on the requirements specifications; they are designed to demonstrate that the system satisfies its requirements.

Functional test cases specify typical operating conditions, typical input values, and typical expected results. Functional tests should also be designed to test boundary conditions just inside and just beyond the boundaries (eg., square root of negative numbers, inversion of one-by-one matrices, etc.).

Performance tests should be designed to verify response time (under various loads), execution time, throughput, primary and secondary memory utilization, and traffic rates on data channels and communication links.

Stress tests are designed to overload a system in various ways, such as attempting to sign on more than the maximum allowed number of terminals, processing more than the allowed number of identifiers or static levels, or disconnecting a communication link. The purposes of stress testing are to determine the limitations of the system and, when the system fails, to determine the manner in which the failure is manifest. Stress tests can provide valuable insight concerning the strengths and weaknesses of a system.

Structural tests are concerned with examining the internal processing logic of a software system. The particular routines called and the logical paths traversed through the routines are the objects of interest. The goal of structural testing is to traverse a specified number of paths through each routine in the system to establish thoroughness of testing.

MILESTONES, WALKTHROUGHS, AND INSPECTIONS

One of the most important aspects of a systematic approach to software development is the resulting visibility of the evolving product. The system becomes explicit, tangible, and accessible. Products of analysis and design to be examined during system development include specifications for the externally observable characteristics of the system, the evolving *User's Manual,* architectural design specifications, detailed design specifications, and the test plan.

Development of these intermediate work products provides the opportunity to establish milestones and to conduct inspections and reviews. These activities expose errors, provide increased project communication, keep the project on schedule, and permit verification that the design satisfies the requirements.

The two major milestones during software design are the Preliminary Design Review (PDR) and the Critical Design Review (CDR). The **PDR** is typically held near the end of architectural design and prior to detailed design. **CDR** occurs at the end of detailed design and prior to implementation.

Depending on the size and complexity of the product being developed, the PDR and CDR may be large. The major goal of a PDR is to demonstrate that the externally observable characteristics and architectural structure of the product will satisfy the customer's requirements.
Functional characteristics, performance attributes, external interfaces, user dialogues, report formats, exception conditions and exception handling, product subsets, and future enhancements to the product should all be reviewed during the PDR.

The CDR is held at the end of detailed design and prior to implementation. Among other things, CDR provides a final management decision point to build or cancel the system. The CDR is in essence a repeat of the PDR, but with the benefit of additional design effort.

A sign-off is required to indicate that the milestone has been achieved. The product designers may benefit from increased customer involvement.

Walkthroughs and Inspections

A structured **walkthrough** is an in-depth, technical review of some aspect of a software system. Walkthroughs can be used at any time, during any phase of a software project. Thus, all or any part of the software requirements, the architectural design specifications, the detailed design specifications, the test plan, the code, supporting documents, or a proposed maintenance modification can be reviewed at any stage of evolution.

A walkthrough team consists of four to six people. The person whose material is being reviewed is responsible for providing copies of the review material to members of the walkthrough team in advance. During the walkthrough the reviewee "walks through" the material while the reviewers look for errors, request clarifications, and explore problem areas in the material under review.

The focus of a walkthrough is on detection of errors and not on corrective actions. A designated secretary for the session records action items to be pursued by the reviewee following the review session. The reviewee is responsible for follow-up and for informing the reviewers of corrective actions taken.

Design specifications are conducted by teams of trained **inspectors** who work from checklists of items to examine. Special forms are used to record problems encountered. A typical design inspection team consists of a moderator/secretary, a designer, an implementor, and a tester.

The designer, implementor, and tester may or may not be the people responsible for actual design, implementation, and testing of the product being inspected. Team members are trained for their specific roles and typically conduct two 2-hour sessions per day. Formal design and code inspections are thus an effective mechanism for error detection and removal.