Discussed is a procedure of hierarchical functional design by which programming projects can be analyzed into system, program, and module levels. It is shown that program design is made more efficient by applying Hierarchy plus Input-Process-Output (HIPO) techniques at each level to form an integrated view of all levels.

HIPO and integrated program design

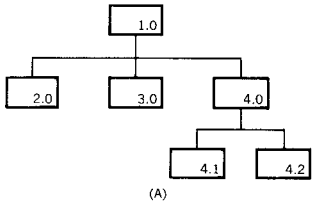
by J. F. Stay

By the mid-1970s, programming appears to be reaching a stage of refinement and cost-effectiveness such that regular business management and control methods can be applied to it. Top-down development, structured programming, chief programmer teams, structured walk throughs, Hierarchy plus Input-Process-Output (HIPO), and structured design have taken us a long way toward transforming "a private art into a public practice."[1] As a result, a body of programming knowledge and methods that are teachable and practicable has been building. This paper discusses the integration of several programming methods by means of an example.

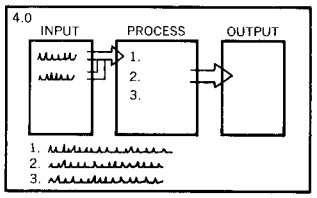
hierarchical functional design

Most of the change in system development has been directed toward the programming effort. Although programming errors are the direct cause of many rework costs, perhaps one third of the rework ultimately can be traced back to errors in the analysis and design phases of a project.[2] Since maintenance can account for as much as seventy precent of all programming costs, more emphasis must be placed on the quality of analysis and design. Structured design and HIPO are useful techniques for organizing the application design process. This article describes how these two methods can be integrated to create a hierarchical functional design. This integrated method allows an application system to be specified from the highest functional level of a conceptual design to the lowest detailed level in a coded routine, using a single method and format. Both the concepts and the techniques of hierarchical functional design can be employed effectively throughout a development cycle in the following phases:

Figure 1 HIPO: Hierarchy plus Input Process Output (A) Schematic diagram of a hierarchy chart (B) Schematic diagram of an input-process-output chart



(A)



(B)

- Requirement definition.
- System analysis.
- System design.
- Program design.
- Detailed module design.
- System and program documentation.

This paper presents a basis for the thought processes involved in designing a system through the use of these techniques. Although this paper does not explain the design techniques in detail, the references provide practical help.

Two techniques for achieving functional design are the following:

- Hierarchy plus Input-Process-Output (HIPO)
- Structured design

HIPO, a technique for use in the top-down design of systems, was developed originally as a documentation tool. HIPO charts continue to serve as the final programming documentation. HIPO consists of two basic components: a hierarchy chart, which shows how each function is divided into subfunctions; and input-process-output charts, which express each function in the hierarchy in terms of its input and output. These two types of charts are illustrated in Figure 1.

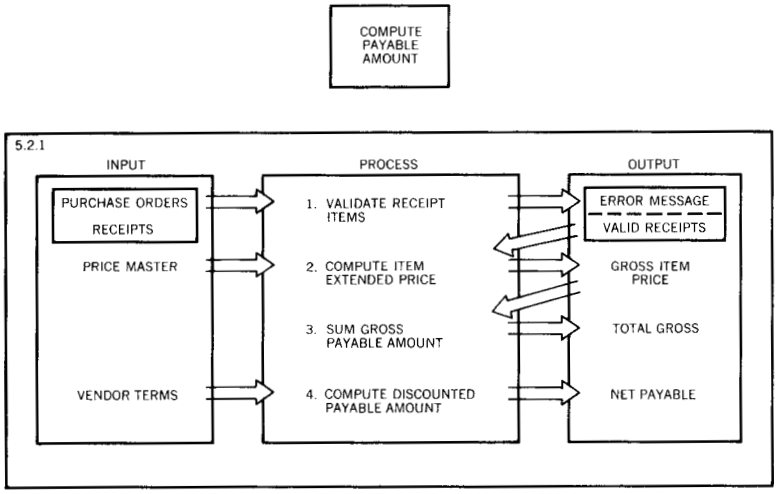
example

The HIPO design process is an iterative top-down activity in which it is essential that the hierarchy chart and the input-process-output charts be developed concurrently, so as to create a functional breakdown. The example of COMPUTE PAYABLE AMOUNT is followed through its development process as part of an accounts payable system.

The first step is to describe a given function as a series of steps, in terms of their inputs and outputs. The input-process-output chart for the example of COMPUTE PAYABLE AMOUNT is shown in Figure 2.

Having completed the input-process-chart, it is possible to move to the next level of the hierarchy. The COMPUTE PAYABLE AMOUNT hierarchy now appears as shown in Figure 3. It is now possible to develop an input-process-output chart for each of the boxes at the level shown in Figure 3. If additional definitions are required, the recommended approach is to make each line on the input-process-output chart a box on the next level of the hierarchy. This process causes the developer to focus on the level of function that is being defined.

Figure 2 HIPO chart for COMPUTE PAYABLE AMOUNT function



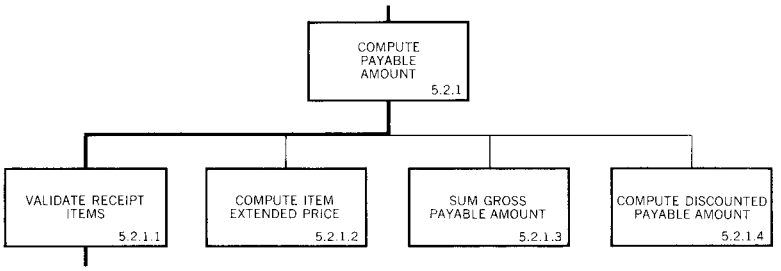
Structured design

The hierarchy plus input-process-output charting is that part of the hierarchical functional design process by which a problem description is made. The other component is structured design. Structured design[3] is a set of techniques for converting from a problem description to a functional, modular program structure. Myers[4] uses the term composite design in reference to the "structure attribute of a program, in terms of module, data, and task structure and module interfaces." This goes beyond the concept of modular design and addresses how a program module is designed, the proper scope of function of a module, and the appropriate communication between modules.

Two concepts of structured design are *module strength* (relationships within a module) and *module coupling* (relationship between modules). The way in which functions are grouped within modules determines the strength of the modules. A module may consist of a group of related functions, such as all editing functions, functions grouped according to the procedure of the problem, or all functions related to a data set.

Functional strength is the grouping of all steps to perform a single function. Although any application may contain modules that have some or all of these strengths, the objective is to produce modules that have functional strength. Functional strength does not apply only to the lowest modular level. A module may call other modules to perform subordinate functions, but if the upper-level module performs a single function and performs it completely, the module probably has functional strength. For example, the module COMPUTE PAYABLE AMOUNT might consist of the following statements (in pseudo-code):

Figure 3 COMPUTE PAYABLE AMOUNT function hierarchy




```
COMPUTE-PAY-AMT (RECEIPT, PAYABLE, RETURN-CODE);
      DO WHILE MORE-ITEMS; GET PRICE-MASTER;
         CALL VALIDATE-ITEM (RECEIPT-ITEM);
         CALL EXTEND-PRICE (RECEIPT-ITEM, PRICE);
         CALL SUM-AMOUNT (PRICE, TOTAL-PRICE);
         CALL COMPUTE-DISCOUNTED (TOTAL-PRICE,
         DISCOUNT-RATE, PAYABLE);
      END DO;
END;
```

This module performs very few functions by itself. However, it transforms one input (RECEIPT) into one output (PAYABLE) completely; at the same time it does no unrelated processing. Therefore, this module is said to have functional strength.

Interactions between modules, termed *module coupling*, may be as varied as interactions within a module. The extreme of module coupling exists when one module directly modifies an instruction in another module.

The preferred relationship between modules is *data coupling*. In data coupling, each module simply passes application data, usually as parameters to the next lower-level module. Use of artificial switches and indicators is avoided. When a calling module passes switches, indicators, or other control information to another module, these items must only communicate the status of the calling program. The calling program should not assume that it knows what the called program will do, based on this control information. A serious problem in program maintenance results when a simple change to a module changes the meaning of an item of control that has an unsuspected effect on the logic flow of one or more other modules.

This paper does not treat structured design in depth but only with sufficient detail to carry the concepts of functional strength and data coupling into earlier stages of the design process. The reader may find References 3 and 4 to be of valuable assistance in module design.

Hierarchical functional design

Hierarchical functional design addresses mental processes that may be applied to the application analysis and design tasks. Hierarchical functional design applies the design concepts of functional strength and data coupling to the functional decomposition and graphic techniques of HIPO to provide a single methodology that allows an application design to develop in an orderly manner from a clear statement of the requirement to an intelligible, well constructed set of application functions. A system that is designed through the use of hierarchical functional design is implemented in a top-down manner. Modules should have a single entry point and a single exit point; they should be small; and they should use the SEQUENCE, IF-THEN-ELSE, and DO-WHILE concepts of structured programming. Hierarchical functional design can be used in all phases of the development cycle, and thereby provide a visable system that is suitable for a design walk through. The chief programmer team concept is also supported, since functional breakdown with clearly defined interfaces allows modules to be delegated to developers or to other teams, with the common understanding that is required for programs to integrate properly.

Hierarchical functional design employs the following three design concepts:

- A functional design in which the computer solution is structured in terms of the user's function.
- An iterative process in which each level of design is validated against the level above it.
- Conceptual levels of design, in which each level emphasizes a particular aspect of the problem solution.

functional design

A computer system can be viewed as a single function that can be divided (or decomposed) into a hierarchy of sets of successively lower-level functions until the elemental functions are described. An understanding of the meaning of the term "function" is necessary for further discussion. *Function* can be defined as an action upon an object, or, for our purposes, the transformation of some input data to some return data.[3] A statement of function describes what is done rather than how it is done. Since a function is also singular, it is defined with a simple declarative statement that consists of only one verb and one object. Both the verb and the object may be conditional.

A function should also have the characteristics that are defined for structured design. A function should be completely defined in one place, and relationships among functions should be primarily data relationships. Thus the concepts of structured design are valid for designing systems as well as modules. Functional

design, then, consists of stating what is to be done in terms of data in and out. A high-level functional statement is reduced to a set of more detailed low-level statements, in a verb-object format. The set of lower-level statements must equal the function of the higher-level statement. In the accounts payable example that is used in this paper, the high-level function is COMPUTE PAYABLE AMOUNT. This function is stated in terms of its input data (purchase receipt) and its output (net payable amount). In this case, the output may simply be passed to another function. The function COMPUTE PAYABLE AMOUNT is then reduced to the following four functional statements:

VALIDATE RECEIPT ITEMS
COMPUTE ITEM EXTENDED PRICE
SUM GROSS PAYABLE AMOUNT
COMPUTE DISCOUNTED PAYABLE AMOUNT

Each of these statements can then be expressed in terms of its input and output data. This set is an explicit statement of the steps required to perform the function COMPUTE PAYABLE AMOUNT.

This is only one example of the way in which a function may be subdivided. The structuring of subfunctions requires analytical skill and imagination, and each analyst may define the subcomponents of a function slightly differently. It is important, however, that the definition of a function determine the functions that are subordinate to it. The function COMPUTE PAYABLE AMOUNT could not legitimately have a subordinate function that, for example, updates the inventory balance.

iterative process

The design of an application should be an orderly growth process from inception to implementation. During development, frequent reviews should be conducted so that a given design always meets its objective. Design has usually been done at least twice before a system is complete and running. First, a functional design has been made to provide an understanding between the user and the programming department. Then a logic design has been made, from which programming could proceed. With hierarchical functional design, the function is the logic, and redundant effort may thus be avoided.

Hierarchical functional design is an evolving, top-down process. The first step is a translation of a statement of need into a functional statement of system objectives. As information about a required system is gathered, that information is organized according to the functional structure. The first statement should contain display screen formats, report layouts, perhaps a two-level hierarchy chart, and a single level of HIPO charts. The analyst should walk through these charts with the customer to veri-

fy that this level of design conforms with the requirement. As the process of design moves to areas such as file access methods, record layout, and message traffic, definitions in successively lower levels the hierarchy may cause upper levels to change. This is typical of the program development process and is the reason for continuing discussion with the customer. This is the iterative process—the refining of the higher levels of design as the more detailed levels are developed. As the iteration of detail progresses downward, the impact on the top-level design should become minimal. Because of the successive iterations of assessing the upward impact of design decisions, the result is a stable, intelligible, and maintainable design.

Levels of the design hierarchy

As an application is divided into functions and each function in turn is subdivided, the hierarchy proceeds toward greater detail. All levels of the system are described as functions, and can be grouped into three categories, proceeding from the broadest to the finest level of detail, as follows:

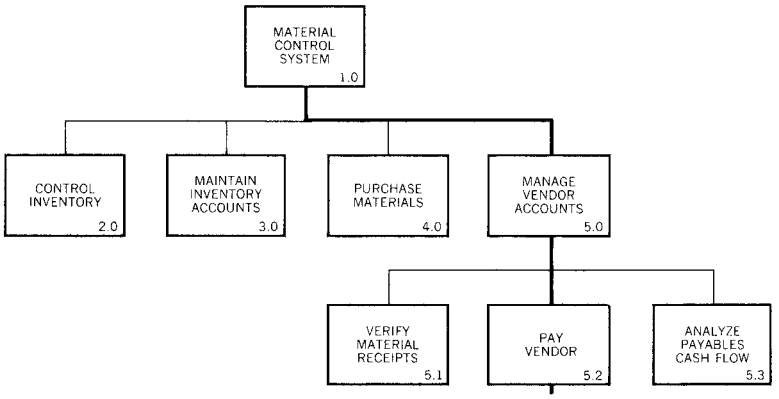
- System
- Program
- Module

These three levels are conceptual, and are not a physical part of the design. Each conceptual level may represent multiple levels on the hierarchy chart, and any given box on a chart may be both the bottom of one conceptual level and the top of the next level.

system level

The system level of the hierarchy contains the major component parts of the application, and is the view that a department manager might have of the application. A system might contain multiple system levels: Accounts payable is a component of a material control system, and in turn, the subsystems of accounts payable itself would be contained within the system level. This level of hierarchy is started by structuring the original statement of requirement for an application. The analysis phase of a project may result in a system-level hierarchy such as that in the example in Figure 4. Each box in Figure 4 can be stated in functional terms, and can be represented by a HIPO chart. Although the terms of the user may differ somewhat from those on the chart, the functional statement should be used because it is more explicit than the common term. For example, the term ACCOUNTS PAYABLE may be simply a subset of the general ledger, or it may be a complete system for managing the payment of vendor accounts. The functional term MANAGE VENDOR ACCOUNTS makes the objective of this application more clear.

Figure 4 System level of the accounts payable application



The system level normally does not include the representation of any executable computer instructions, but rather it provides a conceptual view of the application. Input and output are defined in terms of forms, files, and reports, which are a user's view of the data. If it appears that the user manager's view of the structure of an application does not provide the basis for program design, it should be mentioned that one of the primary objectives of functional design is to have a single view of the application that represents both the user's requirement and the program design. Although functions defined at the top level may be grouped differently for program design, they should all still exist with the same basic relationships in the computer implementation. This is the key to building systems that can be readily maintained and enhanced.

program level

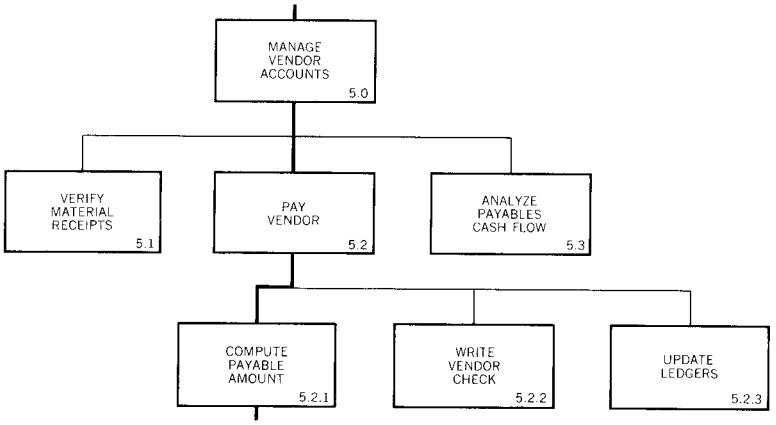
The program level of the hierarchy shows the highest level of segmenting of the computer system. The program level may also be characterized as the end-user level, and represents the level of tasks initiated by a terminal operator in an interactive application, or batch programs in a batch application.

In the accounts payable application, consider the boxes below MANAGE VENDOR ACCOUNTS as tasks or programs. An example program level (one of the boxes in the accounts payable application) is shown in Figure 5. The program or task level is a result of the general design phase of a development project. Input and output for this level are usually defined as records or groups of records, messages, and report lines. Since this level normally represents executable computer instructions, it is recommended that program and module names be assigned to the boxes.

module level

Detailed program design occurs at the module level. Since design is an iterative process, detailed module design may expose

Figure 5 Program level of the accounts payable application



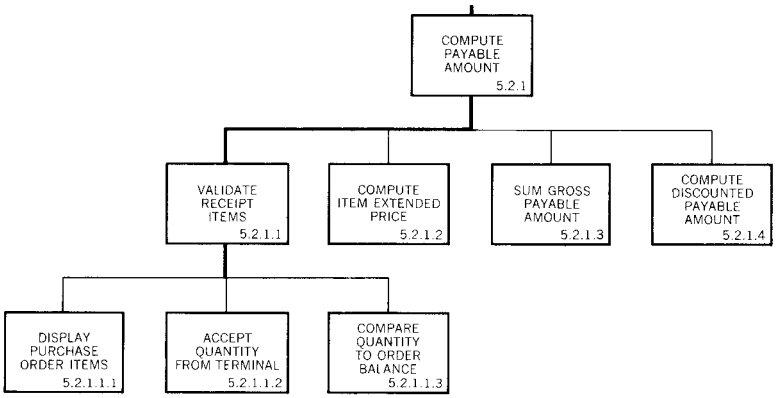
flaws or required restructuring of the more general design. It is essential that the upper-level HIPO charts be revised and revalidated before continuing with the design process. Design modification may be required when a low-level change causes the higher-level design to do something different from that which the user requires.

The module level represents an executable segment of program code that is usually compiled as a unit. This unit of code is typically called an "object module" to distinguish it from a "load module," which may be created by linking several functional modules together. At the module level of the hierarchy, the design is sufficiently detailed that program code can be written directly from the design. In the accounts payable application, two levels of modules are shown in Figure 6 below COMPUTE PAYABLE AMOUNT, which can be related to the program level in Figure 5.

The module level is the product of the specification phase of development. Input and output at this level are fields of data or parameters from or to other modules. The module level should represent a functional statement that can be completely grasped within a normal attention span. When translated into executable code, a module should usually contain fewer than fifty lines of structured high-level language statements. The HIPO chart at the module level may contain structured English (pseudo-code) statements to explain complex logic. In addition, where necessary, the extended description section of the HIPO chart may provide implementation notes as discussed in Reference 5.

The purpose of these conceptual levels is to reflect the objectives of users of the documents. The system level must be stated in terms that are relevant to user management. The module level is organized in such a way as to allow the programmer to write

Figure 6 Module level of the accounts payable application



code. The program level provides the vehicle of communication between the system level and the module level by giving detail to the user and a higher-level view to the programmer.

Hierarchical functional design in a virtual system

The discussion thus far has considered the structure of an application as an aid in the design of an intelligible, maintainable system. A current major concern in system design is that of providing efficient performance of interactive applications in a virtual system environment. Performance tuning in a virtual system is a complex science that involves relationships among hardware, system software, and application design.[6, 7] Optimizing the performance of the application programs alone does not result in an efficient system. A conscious effort of tuning all the components that affect performance is required because programmers, for example, often attempt to write efficient—sometimes complex—code without regard for the way in which the modules may ultimately affect performance.

When viewing the structured design of an application, one can see readily that functional decomposition does not reflect the performance requirements of a transaction-driven application. Hunter,[5] for example, makes clear the issue of the compounded effect of excessive paging. He defines the working set as the twenty percent of the code that does eighty percent of the work, and advises that the "working set should become the focus of application tuning." The design process, if correctly executed, can produce modules each of which requires less than a single 4K byte page of storage. On that basis, the task of application tuning becomes an effort of identifying the most active modules, rewriting those modules for efficiency (if necessary), combining modules into logically related pages, and fixing active pages in main storage.

Performance tuning may, in extreme circumstances, require the radical modification of a few critical modules. Such modifications may include changing a CALL (transfer of control) to a COPY (compile-time inclusion) or even the integration and restructuring of modules. It must be clearly understood that only a very few modules ever seriously affect the performance of most systems. Thus, if an application is structured, the necessary tuning can be done consciously with proper control. With this perspective, even extreme coding techniques may be justified for those few modules that must be efficient to avoid performance degradation.

Concluding remarks

The improvement of the system development process requires innovation in two areas: the development of a discipline that involves a set of structured techniques, and an understanding of the theories on which that discipline is based.

Significant progress has been made in developing a discipline that, in time, should make program design and implementation an engineering skill. Structured programming provides basic building blocks for code development, much as electronic circuit development can be based on a set of predesigned, basic electronic components. HIPO and structured design are first steps in bringing that type of discipline to the design stage of application development.

It has been the intention of this article to address the following mental processes:

- Identification of function.
- Functional decomposition.
- Iterative design.
- Module relationship.
- Delayed performance optimization.

By applying the concepts of hierarchical functional design to the disciplines of HIPO and structured design throughout the analysis and design process, several of the following possible benefits are typically realized:

- User understanding and agreement on functional content are made easier.
- Missing or inconsistent information is identified early.
- Functions are discrete and are therefore more easily documented and, if necessary, modified.
- Documentation is accomplished with a single effort rather than multiple efforts at different stages of development.

- Module interfaces are simple and therefore reduce the probability of logic errors.
- The resultant design supports structured, top-down coding.
- Maintenance and enhancement are more transferable because the system can be easily understood at all levels.

Since these processes are ways of thinking about the design activity, it is often difficult to measure objectively the effect of using this knowledge. By applying these principles to a development project, however, one becomes aware of their value.

CITED REFERENCES

1. H. D. Mills and F. T. Baker, "Chief programmer teams," *Datamation* **19,** No. 12 (December 1973).
2. P. Moody and R. Perry, "Application development cycle problems," *Proceedings of Guide 40*, Miami Beach, May 18–23 (1975).
3. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal* **13,** No. 2, 115–139 (1974).
4. G. J. Myers, *Reliable Software Through Composite Design*, Mason/Charter Publishers, New York, New York (1975).
5. John J. Hunter, "Rethinking application programs, key to VS success," *Computerworld*, March 25, 1975.
6. J. G. Rogers, "Structured programming for virtual storage systems," *IBM Systems Journal* **14,** No. 4, 385–406 (1975).
7. H. A. Anderson, Jr., M. Reiser, and G. L. Galati, "Tuning a virtual storage system," *IBM Systems Journal* **14,** No. 3, 246–263 (1975).

GENERAL REFERENCES

1. *HIPO—A Design Aid and Documentation Technique*, Order No. GC20-1851, IBM Corporation, Data Processing Division, White Plains, New York 10504.

2. *Structured Programming Independent Study Program*, Order No. SR20-7149, IBM Corporation, Data Processing Division, White Plains, New York 10504.

3. J. D. Aron, *The Program Development Process, Part 1, The Individual Programmer*, Addison-Wesley Publishing Company, Reading, Massachusetts, 31–36 (1974).

Reprint Order No. G321-5031.