A simple architecture for consistent application program design

by G. R. Rogers

This paper addresses the architectural design aspects of general business computer application programs written in high-level procedural programming languages. It puts forth design concepts for easily built, maintainable programs and describes a unique approach to program decomposition.

The trend in computer application development is toward the use of application program generators, nonprocedural languages, and other advanced technologies. However, a great amount of current development effort still involves procedural language application programming. This paper mainly concerns those programmers who must continue their work in procedural programming languages without the use of application generators.

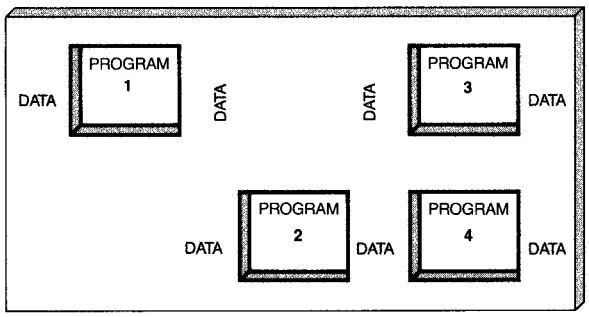
A major expense in computer applications is the cost of ongoing program maintenance, a significant portion of which occurs because different programmers have different solutions to the same programming problems. In many environments, if the same program specifications are given to a number of different programmers, there will be little likelihood of getting any similar solutions and great difficulty in having any programmer maintain any other programmer's solution. When responsibility for a program changes, additional expense is incurred because the new programmer must take the time to understand design techniques that are different from the ones he knows. If the design is not fully understood, changes can damage the program, causing unnecessary cost and inconvenience to the business organization.

A contributing factor to this problem is the programmer approach to functional decomposition. Some published techniques tend to be too general, and programmers apply individual interpretations to them. Other, more specific, published approaches are unsuitable in active, dynamic application environments.

Described herein is a new design approach[1] that can be used for general business application programs written in high-level procedural programming languages. It has been used for batch report programs, batch edit programs, batch sequential update programs, batch selection programs, copy programs, the Application Development Facility (ADF)[2] special processing programs, interactive data entry programs, interactive inquiry programs, and interactive update programs. The approach utilizes a single architecture that views computer application programs as four-level hierarchies of logical modules. Benefits of the technique include improved maintenance productivity through design consistency, faster learning curves for novice programmers, and some simplification of the system design process.

The techniques are used with temporary and new employees. These employees learn the concepts

Figure 1 Detailed system design



quickly, and subsequent people are able to maintain the resulting programs easily.

The following sections of this paper review the system development cycle to show where the concepts fit, discuss other published decomposition approaches and describe the problems with each, explain the four-level concept, show how reusable code applies, demonstrate the concepts with a sample problem, and conclude with a description of other benefits of the approach.

Program decomposition in the system development cycle

The first, and key, step in the system development cycle is to analyze thoroughly the requirements of each particular business enterprise. The system design can be represented by a data flow network such as that shown in Figure 1. The network consists of data streams and nodes (or functions) commonly called programs. A program exists when an identifiable operation, such as report printing, takes place.

When the programs have been identified, they can be created in one of three ways: by using a supplied system utility function such as a sort, by using a program generator such as RPG III,[3] or by writing a procedural program in a language such as PL/I.[4] The last approach is used when the particular programming organization chooses not to use a system utility function or program generator.

If the third approach is adopted, the application programmers are faced with a key problem that is the subject of the remainder of this paper: How is a particular program to be subdivided into its functional components after the overall program functions have been described and documented?

Available decomposition techniques

Four program decomposition techniques currently available to programmers are source/transform/sink decomposition, transactional decomposition, functional decomposition, and data structure decomposition. These techniques have been described by Myers.[5]

Source/transform/sink decomposition is the principal technique used in composite design. It is based on the premise that every problem has an inherent structure and that the program structure should closely resemble the problem structure. Decomposition using this technique involves discovering the inherent structure of the problem and understanding how the data are flowing through the problem structure as well as how the data are transformed while flowing. This information is used to identify the immediate-subordinate modules of the program/module being analyzed. The major steps in this decomposition approach are

1. Identify and outline the structure of the problem.
2. Identify, in this problem structure, the major stream of input data and the major stream of output data.
3. Identify the point in the problem structure where the input data stream last exists as a logical entity and the point where the output data stream first exists as a logical entity.
4. Describe each division of the problem as a single function, using these points as dividing points in the problem structure. These divisions indicate the functions of the immediate-subordinate modules.

Not all programs or program modules can be decomposed using the source/transform/sink method. If the problem does not seem to fit this technique, transactional decomposition is a second method that programmers may use.

When a problem cannot be depicted as a fixed sequence of subproblems (source/transform/sink), it can often be viewed as a set of actions relating to the specific input transactions that must be processed. The program modules may be organized according to the specific types of transactions that must be processed.

There are still other situations where source/transform/sink decomposition and transactional decomposition do not suit the problem. If this is the case, a third approach that programmers may use is functional decomposition. Here programmers look for common functions around which to build a program. Myers[5] dismisses this approach as ad hoc. However, many programs appear to be organized on this basis as programmers recognize and take advantage of common functions appearing repeatedly in business application programs.

To improve productivity, individual programmers must achieve consistency in program decomposition.

When programmers design programs, they often use a combination of these three techniques, with refinement steps, to arrive at a final design. However, a great variance in programmers' solutions to similar programming problems results, although each programmer is, himself, consistent. After gaining some experience, he begins to notice similarities in the types of programs that he develops.

The variance appears to be caused by the general nature of the above-described decomposition techniques. Each programmer interprets the approaches according to his own thought processes. Difficulties arise in the program maintenance phase of the system development cycle when one programmer has to adapt to another's thinking pattern.

To improve maintenance productivity, individual programmers must achieve consistency in program decomposition.

To date, the most consistent decomposition approach is a data structure technique developed by Jackson.[6] This technique is based on the assumption that the structure of a program is related to the input and output data structures of the data it processes. The steps to achieve data structure decomposition are

1. Define the input and output data to clarify the understanding of the problem environment.
2. Find one-to-one relationships between the input and output data structures.
3. Define intermediate data structures to provide a transformation, if there are no one-to-one relationships.
4. Develop a program logic structure based on the data relationships.

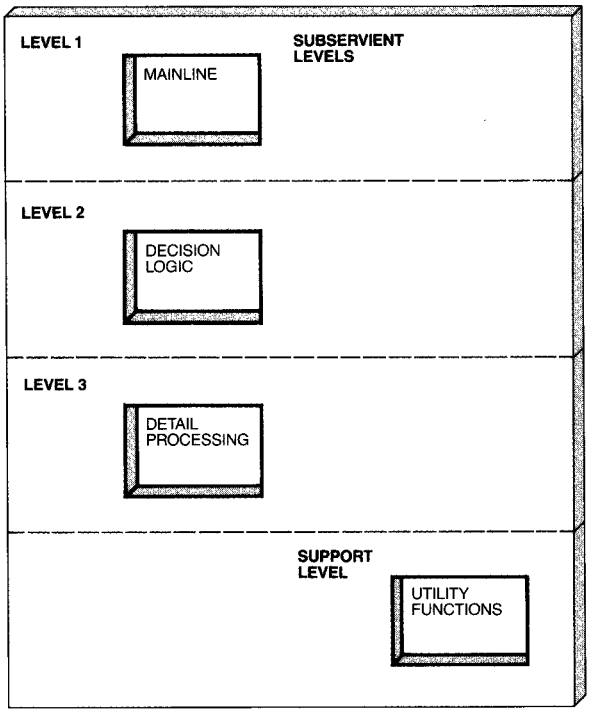
Although one of the values of this approach is that it promotes consistent program designs, development groups in the IBM Toronto Laboratory find it unsuitable in a highly volatile data-base/data-communication environment where data bases are very large and complex and user changes continually affect the structure and contents of the data bases. This type of decomposition forces programmers to discard a great amount of code whenever the data structures change, which is very expensive. Also, this technique may not provide modular programs.

For business application programs, the solution to the decomposition problem is much simpler than the above-summarized methodologies suggest. There are many similarities among such application programs, and herein a four-level design architecture is given to support them. The architecture is simple and consistent, makes use of high- and low-level common code, is highly suitable for an active program maintenance environment, works with software managers such as the Information Management System (IMS),[7] and can quickly aid the development of program design skills in new employees.

A significant difference between this method and other top-down design methods is that the upper two design levels are given, and design refinement, if necessary, involves only the bottom two levels. Identification of specific types of program functions at each design level contributes to design simplicity and is the key to design consistency.

The remainder of this paper describes the architecture and gives a simple program design example to illustrate the concepts. The technique addresses the programming design aspects of computer system implementation. It is appropriate for many general business application programs written in high-level procedural languages.

Figure 2 Basic program structural architecture



The application program as a four-level hierarchy

Procedural language application programs have four key parts: iteration, decision control, data detail processing, and support operations. These functions are organized into a modular structural hierarchy of four logical levels to form the basic program structural architecture, as shown in Figure 2.

Iteration appears at the top of the hierarchy in the mainline module. This module controls the main-event processing for the program. It directs initial and final program processing and evaluates the conditions under which the program operates. With some products, such as IBM's Application Development Facility,[2] the functions of this module are provided as part of the supplied control system. For other environments, programmers may use a standard version of the mainline module, described later in this paper.

The next level in the hierarchy is the decision logic level. It is invoked from the main processing section of the mainline module. This level identifies the type of application program and makes the major program logic flow decisions. The decision logic for most application programs belongs to one of four categories: control break, balance line, single-panel interactive (update or inquiry), or multipanel interactive.

Detail-processing modules are at the next level in the hierarchy, where data construction and manipulation take place. These functions are generally unique to each program.

Acting for all levels in the hierarchy are the utility function modules. These operate at the support level and provide common functions usable by all levels in the application program hierarchy. Utility function modules can support other utility modules.

In general business application programs, many functions occur frequently.

Decisions affecting overall program flow appear in the higher levels of the program hierarchy, whereas data processing is concentrated at the lower levels.

Classical functions

A reusable function is a programming function that is developed once and used in many different programs. Since Stevens, Myers, and Constantine[8] theorized that composite design would result in less new code being developed, many authors have discussed the subject of reusable functions. However, there has not been much specific documentation on reusable functions. Existing documentation tends to be project-specific, and the documented code occurs at a low hierarchical level in programs.

In business application programs, great amounts of data, often contained in complex structures, must traverse most of the key functions. Therefore, if there are to be reusable functions of a generally applicable nature, they must be able to handle the different elements and data structures that occur in various programs.

The data variations make it difficult for programmers, without access to code generators, to create ready-to-run reusable functions in object code form. The practical alternative is to specify the functions in pseudo code. The programmer adds his data references to the pseudo code to complete the function designs. He can then code the functions in his selected programming language and compile them to produce the object code.

In general business application programs, many functions occur repeatedly. The author has identified some of these functions and developed specific reusable code which fits in the above-described design architecture at *high and low levels* in programs. The author has chosen to call these "classical" functions and in a handbook[9] has provided pseudo PL/I code for them. (Although the programming language of our development department is PL/I, the code is simple and has been written to be understood by readers who work in other high-level procedural programming languages.)

In the following section, each of the logical levels in the structure of a program is discussed. Also given is a descriptive overview of the "classical" functions that can be used at the various levels.

Program structural levels

Level 1 — Mainline. One of the biggest problems facing maintenance programmers is determining the exact conditions under which a program executes. Often maintenance programmers examine several levels in the hierarchy of a program and are perplexed as to why the program runs, given certain input conditions. When the logic for primary control is widely dispersed, the initial developer and subsequent maintenance programmers may have difficulty fully understanding all the interactions of the program. It is extremely important for maintenance programmers to know under what circumstances the program functions, and this knowledge should be easily obtainable. When the run conditions of the program are buried in the hierarchy, programmers can easily make changes at the wrong places, causing unpredictable results.

The primary purpose of the mainline logic is to make the key operative conditions clearly visible. With consistent documentation standards, programmers should be able to comprehend readily the reasons for program execution.

Figure 3 Mainline module

```
MODULE MAINLINE [(system parameters)] MAIN.

  CALL INITIAL (initialization variables).

  DO WHILE condition for data to be processed is valid.
     CALL DECIDE (decision logic variables).
  ENDDO.

  CALL ENDJOB (end of job variables).

ENDMODULE.
```

This mainline module is the root of the program at the top of the program hierarchy. It is usually written by the programmer. Sometimes it is supplied as part of a software manager, as with the special processing option of ADF.[2]

For designing the mainline logic, a "classical" function called the *standard mainline*[9] has been developed. It is a very simple piece of code that controls three sequential functions: initialization, main processing, and end-of-job processing. The logic for the standard mainline module is shown in Figure 3.

Initialization uses a utility-level module to establish starting values and to handle other processing (such as control record validation) that takes place at the beginning of the program.

Following initialization is main processing, the major component of the mainline module. This function checks the validity of the condition or conditions under which the program operates and iteratively invokes the processing decision logic at the next lower level in the program. An input data stream that has not yet reached its end or a terminal user who has not yet told the program he is finished are valid conditions for continued program operation.

The final step in the mainline module is to call a utility-level module to handle the end-of-job processing. The end-of-job module may print grand totals, produce end-of-job summary reports, write audit totals onto a log file, and perform orderly program shutdown.

There is limited data processing and handling at this top level in the program hierarchy. A condition or conditions controlling execution of the program, data to be initialized, data passed to the end-of-job processing module, and data base and data communication work areas (which may be parameters when control is passed to the mainline module) are the likely data to be handled.

In an interactive system, the programmer may want to control the processing flow from data panel to data panel.

Level 2 — Decision logic. Decision logic is the second level in the program structural hierarchy. Here the characteristic logic of the processing solution appears. Again, practically no data are processed at this level. However, the amount of data handling increases significantly. The decision logic provides, in a sense, a message switching/control function and ensures that the appropriate data are correctly passed among the subservient program levels.

When programmers acquire some experience, they begin to notice that many programming problems repeatedly appear. Most general business application programs can be classified in one of four groups: control break, balance line, single-panel interactive (update and inquiry), or multipanel interactive. "Classical" reusable functions are defined for each of these groups.[9]

Control break logic is a very common programmer-written application logic. It is used for batch report programs, batch validation programs, and data reduction programs (where data are condensed and put on a file or data base for further processing by subsequent programs) and is based on the concept of changes in sequence keys of a sequenced input data stream. A control break occurs when a key field changes value. At this time, related processing such as totaling takes place. A zero control break program is a copy program.

Balance line logic concerns the matching of sets of streams of sequential items. The match may be called a file update, a merge, or a selection; one stream of data is matched against another similarly sequenced stream of data. The input consists of two streams of data in ascending, similarly sequenced order. The input data may come from sequential files, data bases, or tables. The output is a single stream of sequential data containing the results of the matches. The output data stream may also be a sequential file, data base, or table.

The *single-panel interactive* update transaction occurs in many simple on-line update systems. Each user-terminal transaction is treated as a single panel with associated source code. A simplified version of this logic can be used for inquiry-only applications.

In an interactive system, the programmer may want to control the processing flow from data panel to data panel. Some of the panels are simply selection (menu) panels, others are data-only panels, still others may be a mixture of the two types. The decision logic of the *multipanel interactive* transaction controls the flow from panel to panel and from an input/output device to a panel.

Level 3 — Detail processing. At this level, the program should primarily process the data. The actual modules that appear are defined by the decision logic that invokes them.

Programs should have simplified control logic and should maximize the data processed. For these reasons, the modules at this level in the structure of a program will be much larger than those found at the mainline and decision logic levels.

Since the detail processing level of the program deals with specific data manipulation, this level tends to be unique for each program. Therefore, no "classical" functions have as yet been identified for this level.

Level 4 (support) — Utility function. Utility functions occur in most business programs. They perform tasks that are called from various locations throughout the program. Utility functions can be supported by other utility functions. Therefore, although the program structure has four logical levels, it may have more than four physical levels.

In general, utility functions tend to be relatively simple modules that process data rather than affecting control logic. "Classical" functions that have been identified[9] at this level are print report detail, simple sequential read, sequential read and table load, Zeller's Congruence, "bubble" sort, and binary search.

The *print report detail* logic is a simple, common module used to print report data. It formats and outputs the appropriate data and sets up and prints report page heading lines for each new page. Often this particular piece of logic is incorrectly created by programmers.

The *simple sequential read* logic reads a sequential stream of data coming from a file or data base. In addition to providing physical access to the data, it verifies the sequence of the incoming stream of data. Failing to check sequence is a common programming error.

When sequential input data appear in sets of records or segments, it is convenient to load related groups into a table for subsequent group handling. Programmers may use the logic of the *sequential read and table load* to perform this function.

The best method of sorting files and data streams is to use system-provided "stand-alone" sort facilities.

In data processing, programmers may need to know the day of the week that a particular event takes place. For example, certain functions may have to be executed every Wednesday or a program may have to account for employees not working on Saturday or Sunday. To convert a date (e.g., October 21, 1982) into the day of the week (e.g., Thursday), a formula known as *Zeller's Congruence*[10] is available. Month, day, and year are the input parameters, and the procedure produces an alphabetic and a numeric representation for the day of the week (Sunday through Saturday).

The best method of sorting files and data streams is to use system-provided "stand-alone" sort utilities; however, from time to time, programs are required to order small in-memory tables. A simple method for this operation is to use a function known as *"bubble" sort.*

Numerous programs require data table searches. These searches are normally accomplished by using a search verb or a similar function provided by the programming language. Sometimes programmers write their own sequential search modules. If a programmer must search a *very large* sequenced in-memory table, he may use the *binary search* "classical" function.

In addition to these reusable functions, businesses have additional sets of utility functions characteristic of the enterprise. These functions will be identified by the organization itself. In the following example program, the module for user profile evaluation is probably a reusable function for all users in a company.

An example program

The decomposition of an IMS data-base/data-communication program illustrates the method of designing application programs. This example is less complex than real-life programs, but it does illustrate the decomposition technique and shows how the functional modules are interconnected to form the program. The design process is taken through the decomposition phase to the identification and high-level pseudo coding of the functional modules. From that point onward in the development cycle, the coding is, to a large extent, a relatively mechanical exercise.

The example program is an on-line program used to update a data base containing existing employee name and address data. A display terminal device is used to enter transactions for processing by the program.

IMS sign-on and user validation procedures are used, and, after signing on to the program, the user will be presented with a selection menu. The user then selects a transaction code and, depending on the choice, the program ends, a name and address display is shown, or a job description display is

shown. The layout of these displays can be found in the Appendix to this paper.

For name and address data, the user is prompted to enter an employee number. The program then accesses an employee data base and returns the name and address information. With proper authorization, the name and address data can be changed by the terminal user.

For job description data, the user is also prompted to enter an employee number. The program subsequently accesses the employee data base and returns with a display showing employee name, job title, and job description. Proper authorization enables the user to change the job description data.

There are two major types of multitransaction interactive programs.

Both data panels have an option for experienced users, allowing them to transfer to another display or end the program without recourse to the menu display.

All programs in the user organization must access a common utility function for data base update authorization.

The program uses an option of IMS known as Message Format Services (MFS) for management of the physical display data. However, in this example, only the application logic is designed. High-level English pseudo code for the solution is shown in the Appendix.

There are two major types of multitransaction interactive programs: those which "see" a complete session for each connected terminal and those which process only one interaction for any terminal at a time. IMS data communication programs belong to the second category. When continuity is required between subsequent terminal interactions, IMS provides storage areas for data retention. Each storage area is associated with a particular terminal.

For the design, the programmer begins the module identification process at the top of the program hierarchy. Since the program is to operate under the control of the IMS data communication monitor, without the services of a menu management system, the programmer must provide his own mainline module. He provides the module by taking the "classical" function of Figure 3 and modifying it to show the unique program data references.

The mainline code shows, without further design refinement, the modules that are invoked by it at lower levels in the program hierarchy: initialization, decision logic, and end-of-job processing.

The initialization module performs both terminal session initialization and, subsequently, transaction initialization. It first requests a work area for the terminal from IMS. The work area contains an indicator showing whether this is the first interaction for a terminal or a subsequent interaction. On the first interaction the program starting values must be set.

The decision logic module controls the overall program flow. Although the mainline shows a DO statement, the decision logic will be invoked only once for each passage of control from IMS. The DO format is retained to be consistent with the architecture and other multipanel interactive programs that "see" complete terminal sessions. In some circumstances, for performance reasons, the programmer may elect to take advantage of the IMS option of processing all available transactions from all users before control reverts back to IMS. This option is achieved with the same mainline logic having iteration control based on the availability of messages from all users, rather than just one.

The end-of-job processing module performs both end-of-transaction processing and end-of-terminal session processing. The terminal user enters an input parameter (detected by the menu and transaction processing functions) when he desires to end his session. For end-of-transaction processing, this module has IMS save necessary data. For end-of-terminal session processing, the module sets a value in the work area to end the program.

Figure 4 shows the first step in the design process.

The programmer proceeds to the design of the decision logic (second level in the program hierarchy). Because the sample program has three interactive transactions, it can be categorized as a multipanel interactive transaction "classical" function.

The pseudo code is modified with the data references to finalize the design at this level in the program hierarchy, without further refinement. The code identifies all dependent modules at lower levels in the module hierarchy: user profile evaluation, receive data from user, send data to user, menu processing, name and address processing, and job description processing.

The code shows that this type of interactive program cannot be decomposed using the source/transform/sink approach. The structure of the solution is very different from the structure of the problem. The problem structure is a hierarchy of panels. Menus can invoke transactions or other menus; transactions can invoke menus or other transactions. The solution is a single-level iteration allowing the problem hierarchy to be substantially modified with minimal impact on the program.

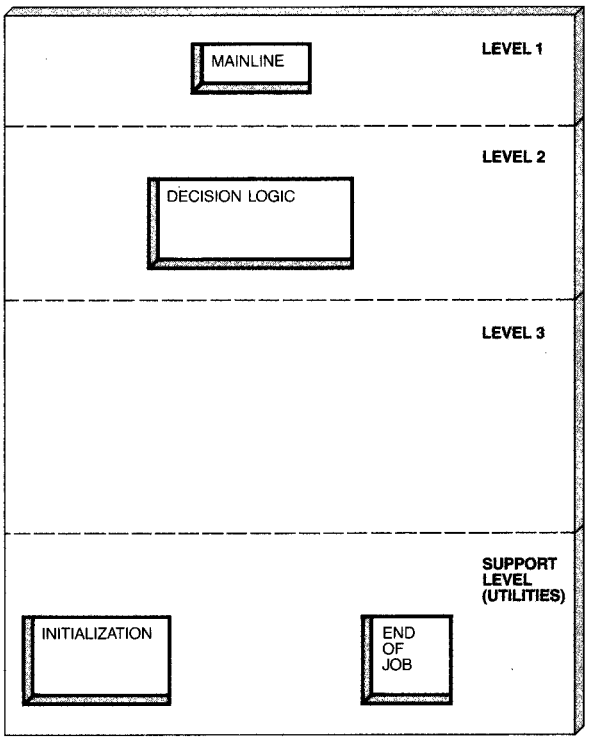
The user profile evaluation module verifies user access to the transaction. It is a utility function operating at the support level. When a particular transaction is selected, the transaction processing module checks that the code returned by the profile evaluation module is acceptable. If the code is not acceptable, the module sends a message to the terminal user rejecting the update transaction. User profile data are checked each time the decision logic is invoked. If the application is long-running, this check allows instant recognition of changes in user profile data without having to terminate the application.

The send and receive data modules are utility functions that provide the access path from the program to the physical terminal device.

The menu-processing module is another transaction operating at the detail-processing level in the program. It verifies the selection option and responds with a parameter that will cause the decision logic to pass control to the selected transaction.

The name and address processing evaluates selection options, prompts the user for data, and updates the data base (when the user has the proper authorization).

Figure 4 First design step



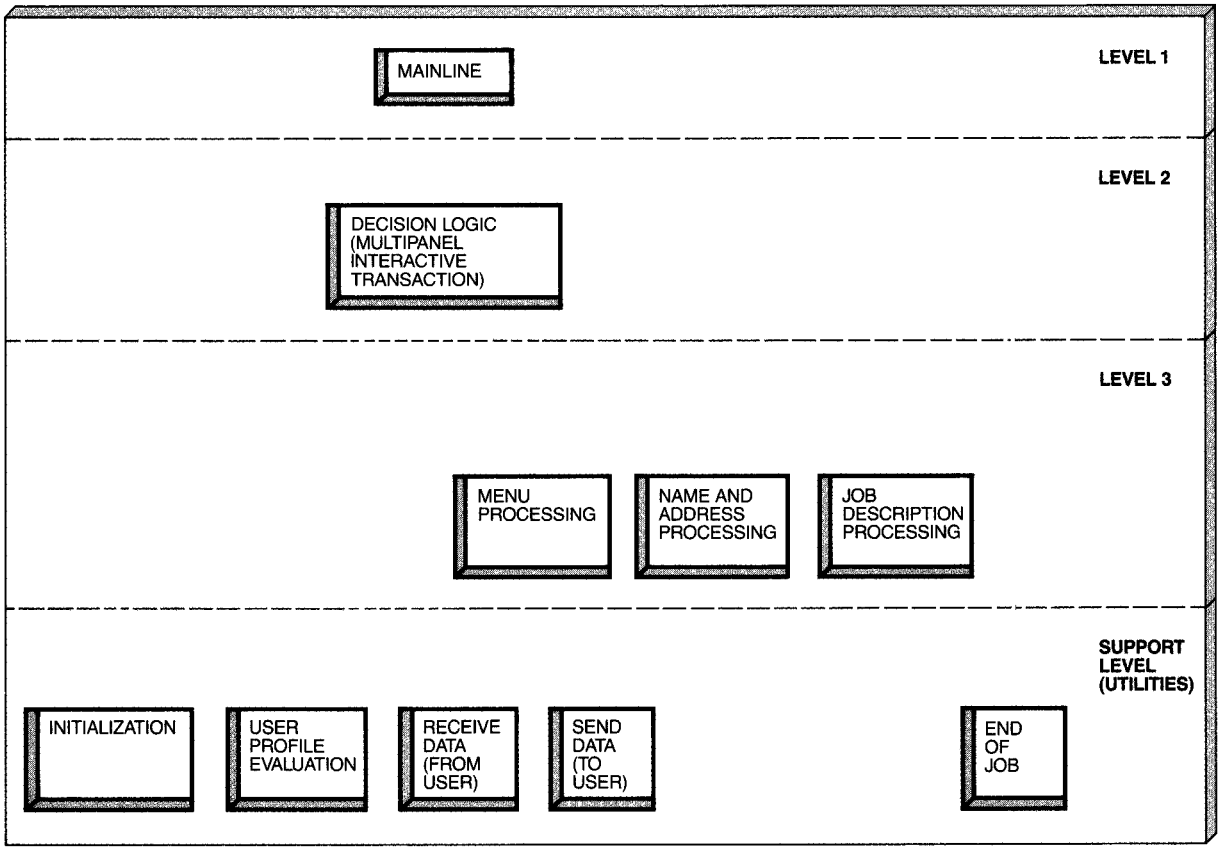
The job description processing is very similar to name and address processing, the job title and job description fields being the minor differences.

The second step of the design process is now complete and is shown in Figure 5.

If module design refinement is needed, it is done at the detail-processing and support level phases of the design evaluation process. The detail-processing level is evaluated first to determine if any supporting functions are needed. The menu processing module is simple and does not need any supporting functions. Because the name and address processing module and the job description processing module are so similar, we may initially conclude that they are both likely to use the same supporting functions. The functions they require are one to read a data base and one to update a data base.

In the final step, the programmer analyzes the support level utility functions (initialization, receive, send, data base read, data base update, and end-of-job) to determine if they need to be decom-

Figure 5 Second design step



posed to give better modularity. Because the example program is simple, no further decomposition is necessary.

The final structural design is shown in Figure 6.

As each level in the program design is established, the pseudo code for the associated modules can be written. At the completion of the structural design, program coding in the selected programming language becomes a mechanical exercise.

Other benefits

Although the major benefit of this decomposition approach is design consistency and thus greater productivity in a programming maintenance environment, the approach provides other significant benefits as well.

People who are novice programmers often have difficulty determining what a program should look like and, in particular, how code should be designed. This method is simple so that the decomposition technique and the code examples[9] provide a handy reference to show people how common business data processing problems can be solved. The problem of new programmers having to continually reinvent the work of past generations is avoided.

This design technique also affects the system design process. It simplifies the identification of the functional units known as programs. Business application programs can be identified as one of four types (control break, balance line, single-panel interactive, multipanel interactive). Along with these programs, there are system utility programs such as sort and copy (file to file, file to data base, data base to file, data base to data base). With use of these

utility programs and application programs, system design entails making selections from among known functions and arranging them in the proper sequence to provide the desired system data flow characteristics.

Concluding remarks

Consistency and maintainability of application programs, designed using the techniques described in this paper, are a result of the simplicity of the approach. Simplicity is achieved via the single architecture, the stratification of the program into four distinct layers, and "classical" functions that show programmers how to design frequently occurring program code.

The concepts are easy to teach, thus reinforcing the notion that they are simple. New employees in the Supply Management System area of the IBM Toronto Laboratory are taught on a tutorial basis, and university continuing-education students are taught in a classroom format.

The author continues to attempt to identify additional "classical" functions relevant to business programming environments. Until procedural language programming is completely replaced by application code generators, there will be a substantial demand for techniques to improve the productivity of business application programmers.

Acknowledgment

My gratitude goes to those of the staff and management of the IBM Toronto Laboratory who gave their support and encouragement and spent many hours reading and constructively criticizing my manuscripts.

Figure 6 Final design step — Sample multipanel interactive program

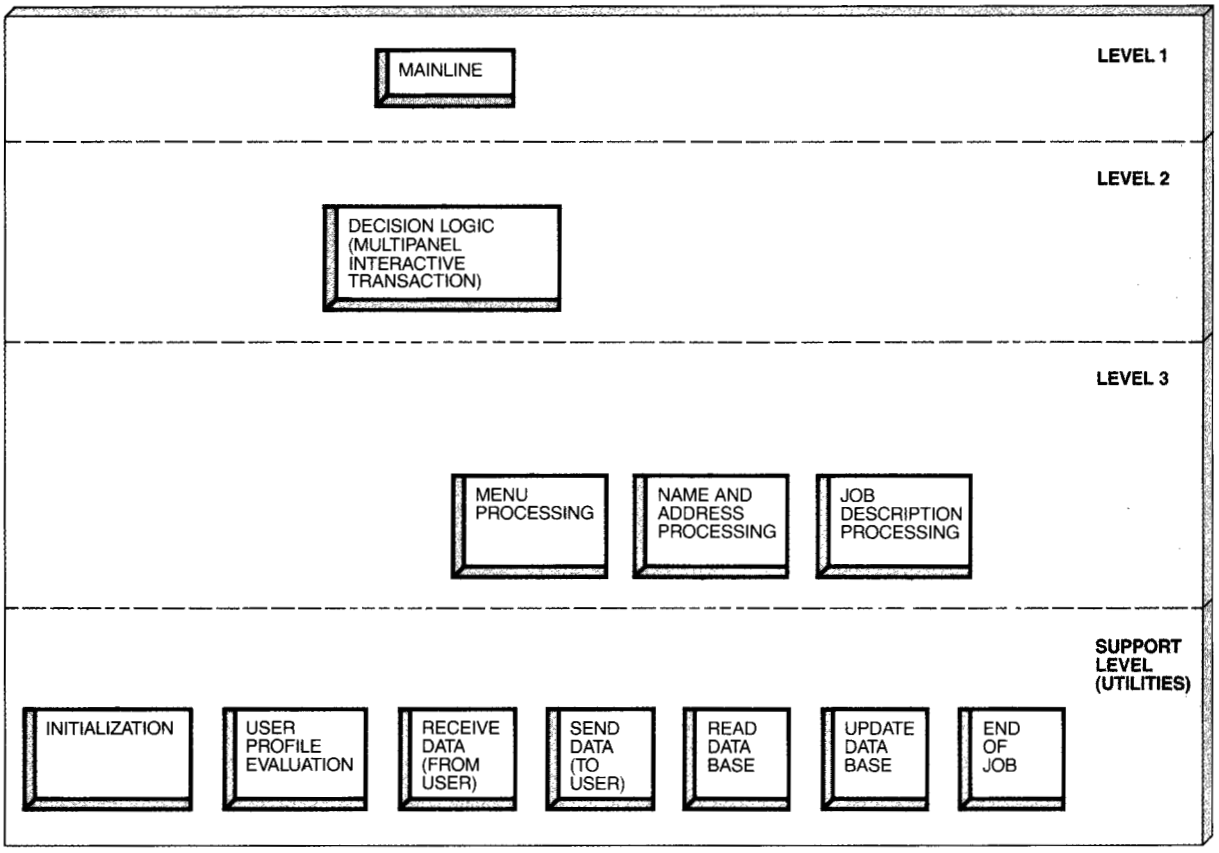


Figure 7 Selection menu

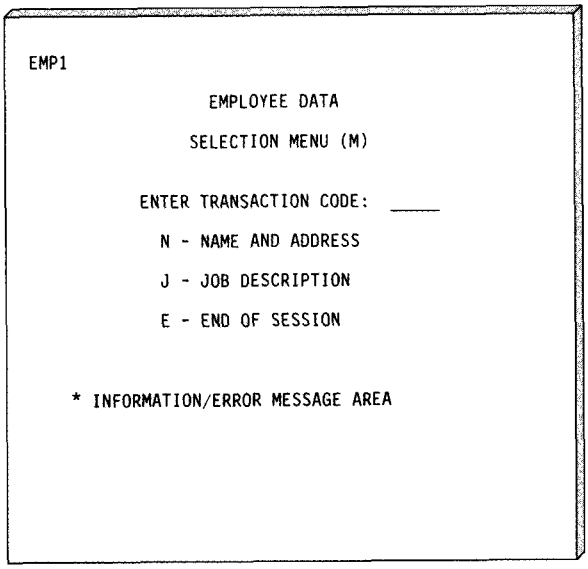
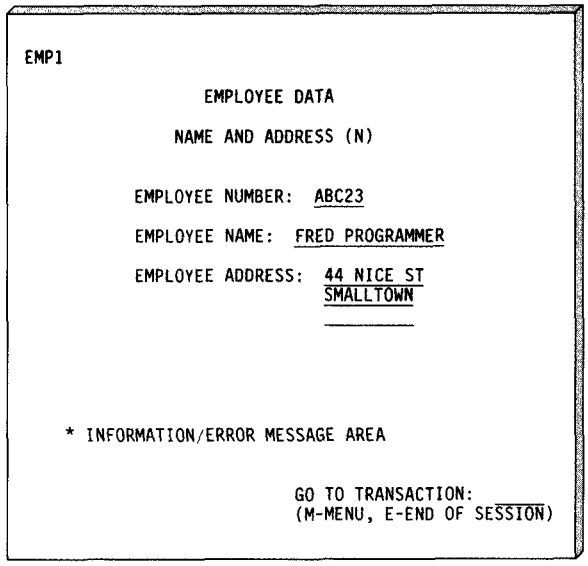


Figure 8 Name and address display



Appendix

In this appendix, the pseudo code for each module of the example program is given. Figures 7, 8, and 9 depict the displays that will be shown depending on a user's choice of transaction code as described in the example.

Mainline module

```
MODULE MAINLINE (IMS parameters /* including user id */)
                MAIN.
  CALL INITIAL  (IMS parameters,first time indicator,end of
                session indicator,panel code,previous panel
                code,employee number,previous employee
                number,name,address,title,description).
  DO once for each terminal interaction.
    CALL DECIDE (IMS parameters,first time indicator,end
                of session indicator,panel code,previous
                panel code,employee number,previous
                employee number,name,address,title,de-
                scription).
  ENDDO.
    CALL ENDJOB (IMS parameters,first time indicator,end
                of session indicator,panel code,previous
                panel code,employee number,previous
                employee number,name,address,ti-
                tle,description).
ENDMODULE.
```

Initialization module

```
MODULE INITIAL (IMS parameters,first time indicator,end of
               session indicator,panel code,previous
               panel code,employee number,previous em-
               ployee number,name,address,title,descrip-
               tion).
  get work area for terminal containing IMS parameters,first
    time indicator,panel code,previous panel code,employee
    number,previous employee number,name,address,ti-
    tle,description.
  IF first time indicator shows first use for this terminal
    THEN
      set panel code to "m".
      set previous employee number to blank.
  ENDIF.
  set end of session indicator to "no".
ENDMODULE.
```

End-of-job module

```
MODULE ENDJOB (IMS parameters,first time indicator,end
              of session indicator,panel code,previous
              panel code,employee number,previous
              employee number,name,address,title,de-
              scription).
  IF end of session indicator = "yes"
    THEN
      set IMS parameter to end terminal session.
  ENDIF.
```

insert work area for terminal containing IMS parameters,first time indicator,panel code,previous panel code,employee number,previous employee number,name,address,title,description.

ENDMODULE.

Decision logic module

```
MODULE DECIDE (IMS parameters,first time indicator,end of
               session indicator,panel code,previous
               panel code,employee number,previous em-
               ployee number,name,address,title,de-
               scription).
  CALL PROFILE (program code, user id, database id,
                access authority).
  IF first time indicator shows first time for this terminal
    THEN
      reset first time indicator.
      set previous panel code to blank.
    ELSE
      CALL RECEIVE (IMS parameters,panel code,em-
                    ployee number,name,address,ti-
                    tle,description,transaction code).
  ENDIF.

  set processing control to "select".
  DO WHILE processing control = "select".
    CASE panel code.
      WHEN "m"
        CALL MENU (IMS parameters,end of session indi-
                   cator,panel code,previous panel
                   code,message,transaction code,pro-
                   cessing control).
        set previous panel code to "m".
      WHEN "n"
        CALL NAME (IMS parameters,end of session indi-
                   cator,panel code,previous panel
                   code,employee number,previous em-
                   ployee number,name,address,ti-
                   tle,description,transaction code,ac-
                   cess authority,processing control).
        set previous panel code to "n".
      WHEN "j"
        CALL JOB (IMS parameters,end of session indica-
                  tor,panel code,previous panel code,em-
                  ployee number,previous employee num-
                  ber,name,address,title,description,
                  transaction code,access authority,pro-
                  cessing control).
        set previous panel code to "j".
      OTHERWISE
        set panel code to previous panel code.
        set message to "invalid transaction code".
        set processing control to "send".
    ENDCASE.

  ENDDO.

  CALL SEND (IMS parameters,panel code,employee num-
             ber,name,address,title,description,mes-
             sage,transaction code).
ENDMODULE.
```

Figure 9 Job description display

```
EMP1

                    EMPLOYEE DATA

                 JOB DESCRIPTION (J)

          EMPLOYEE NUMBER:  ABC23

          EMPLOYEE NAME:  FRED PROGRAMMER

          TITLE:  JUNIOR PROGRAMMER

          JOB DESCRIPTION:  WRITES NEW PROGRAMS
                            AND MAINTAINS EXISTING
                            PROGRAMS

    * INFORMATION/ERROR MESSAGE AREA

                              GO TO TRANSACTION:
                              (M-MENU, E-END OF SESSION)
```

Receive module

```
MODULE RECEIVE (IMS parameters,panel code,employee
                number,name,address,title,descrip-
                tion,transaction code).
  set up IMS/MFS terminal input parameters.
  get terminal message segments containing employee num-
      ber,name,address,title,description,transaction code
      for the particular panel code.
ENDMODULE.
```

Send module

```
MODULE SEND (IMS parameters,panel code,employee num-
             ber,name,address,title,description,mes-
             sage,transaction code).
  set up IMS/MFS terminal output parameters.
  insert terminal message segments containing employee
    number,name,address,title,description,message,trans-
    action code for the particular panel code.
ENDMODULE.
```

Menu processing module

```
MODULE MENU (IMS parameters, end of session indicator,
             panel code, previous panel code, message,
             transaction code, processing control).
  IF previous panel code not = "m" or transaction code =
    blank
    THEN
      set message to blank.
      set transaction code to blank.
```

```
      set processing control to "send".
      RETURN.
  ENDIF.
  IF transaction code = "e"
    THEN
      set employee number,name,address,title,description,
        transaction code to blank.
      set message to "good bye".
      set end of session indicator to "yes".
      set processing control to "send".
    ELSE
      set panel code to transaction code.
      set processing control to "select".
  ENDIF.
ENDMODULE.
```

Name and address processing module

```
MODULE NAME (IMS parameters,end of session indi-
            cator,panel code,previous panel code,em-
            ployee number,previous employee number,
            name,address,title,description,transaction
            code,access authority,processing control).
  IF previous panel code not = "n"
    THEN
      set employee number,name,address,transaction code
        to blank.
      set message to "enter employee number".
      set processing control to "send".
      RETURN
  ENDIF.
  IF transaction code = "e"
    THEN
      set employee number,name,address,title,descrip-
        tion,transaction code to blank.
      set message to "good bye".
      set end of session indicator to "yes".
      set processing control to "send".
      RETURN.
  ENDIF.
  IF transaction code not = blank
    THEN
      set panel code to transaction code.
      set processing control to "select".
      RETURN.
  ENDIF.
  IF employee number not = previous employee number
    THEN
      CALL READ (IMS parameters,employee num-
                 ber,name,address,title,description,mes-
                 sage).
      IF message = blank
        THEN
          set previous employee number to employee num-
          ber.
      ENDIF.
      set processing control to "send".
      RETURN.
  ENDIF.
  IF access authority shows updates are allowed
    THEN
      CALL UPDATE (IMS parameters,employee num-
                   ber,name,address,title,descrip-
                   tion,message).
    ELSE
      set message to "update not authorized".
  ENDIF.
  set processing control to "send".
ENDMODULE.
```

Job description processing module

```
MODULE JOB (IMS parameters,end of session indica-
           tor,panel code,previous panel code,employee
           number,previous employee number,name,ad-
           dress,title,description,transaction code,ac-
           cess authority,processing control).
  IF previous panel code not = "j"
    THEN
      set employee number, name, title, description, transac-
        tion code to blank.
      set message to "enter employee number".
      set processing control to "send".
      RETURN.
  ENDIF.
  IF transaction code = "e"
    THEN
      set employee number,name,address,title,descrip-
        tion,transaction code to blank.
      set message to "good bye".
      set end of session indicator to "yes".
      set processing control to "send".
      RETURN.
  ENDIF.
  IF transaction code not = blank
    THEN
      set panel code to transaction code.
      set processing control to "select".
      RETURN.
  ENDIF.
  IF employee number not = previous employee number
    THEN
      CALL READ (IMS parameters,employee num-
                 ber,name,address,title,description,mes-
                 sage).
      IF message = blank
        THEN
          set previous employee number to employee num-
          ber.
      ENDIF.
      set processing control to "send".
      RETURN.
  ENDIF.
  IF access authority shows updates are allowed
    THEN
      CALL UPDATE (IMS parameters,employee num-
                   ber,name,address,title,descrip-
                   tion,message).
    ELSE
      set message to "update not authorized".
  ENDIF.
  set processing control to "send".
ENDMODULE.
```

Data base read module

```
MODULE READ (IMS parameters,employee number,
             name,address,title,description,message).
  set up IMS data base access parameters.
  issue get unique with employee number key to get
    employee name, address, title, description.
  IF data base segment found
    THEN
      set message to blank.
    ELSE
      set message to "employee information not found".
  ENDIF.
ENDMODULE.
```

Data base update module

```
MODULE UPDATE (IMS parameters,employee num-
               ber,name,address,title,description,mes-
               sage).
  set up IMS data base access parameters.
  issue get hold unique with employee number key to get
    employee name, address, title, description.
  IF data base segment not found
    THEN
      set message to "employee data not available for
        update".
      RETURN.
  ENDIF.
  issue replace with employee number key to update
    employee name,address,title,description.
  IF update is successful
    THEN
      set message to "data updated".
    ELSE
      set message to "update unsuccessful — call admini-
      strator".
  ENDIF.
ENDMODULE.
```

Cited references and note

1. This method has not been submitted to any formal IBM test. Potential users should evaluate its usefulness in their own environment prior to implementation.
2. *IMS Application Development Facility Program Description/Operations Manual*, SH20-2634, IBM Corporation; available through IBM branch offices.
3. *IBM System/38 RPG III Reference Manual and Programmers Guide*, SC21-7725, IBM Corporation; available through IBM branch offices.
4. *OS and DOS PL/I Language Reference Manual*, GC26-3977, IBM Corporation; available through IBM branch offices.
5. G. J. Myers, *Composite/Structured Design*, Van Nostrand Reinhold, Toronto (1978).
6. M. A. Jackson, *Principles of Program Design*, Academic Press, Inc., New York (1975).
7. *IMS/VS General Information Manual*, GH20-1260, IBM Corporation; available through IBM branch offices.
8. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal* **13**, No. 2, 115–139 (1974).
9. G. R. Rogers, *Program Design Handbook*, TR-74.024, IBM Canada Limited; available through the IBM Canada Limited Laboratory Librarian, Toronto (1982).
10. J. V. Uspensky and M. A. Heaslet, *Elementary Number Theory*, McGraw-Hill Book Co., Inc., New York (1939).

Gary Robert Rogers *IBM Canada Limited Laboratory, 1150 Eglinton Avenue East, Don Mills, Ontario, Canada M3C 1H7.*

Mr. Rogers is a staff development analyst. He joined IBM in 1974 and has worked in IBM Canada Headquarters and the Toronto Laboratory. For the past three years, he has been involved with the development of the IBM Americas/Far East Corporation Supply Management System. Prior to joining IBM, he completed numerous application software development projects with a software house and a large corporation. He received a B.A.Sc. in electrical engineering from the University of Toronto in 1965, is a member of the Association of Professional Engineers of Ontario, and is a lecturer at the University of Toronto School of Continuing Studies.

Reprint Order No. G321-5191.