Software Engineering with Reusable Designs and Code

ROBERT G. LANERGAN AND CHARLES A. GRASSO

Abstract—**For over six years Raytheon's Missile Systems Division, Information Processing Systems Organization has used a successful approach in developing and maintaining business software. The approach centers on the fact that 60 percent of all business application designs and code are redundant and can be standardized and reused. This approach has resulted in significant gains in productivity and reliability and improved end-user relations, while providing better utilization of data processing personnel, primarily in the maintenance phase of the software life cycle.**

INTRODUCTION

IT is common practice when writing scientific programs to use prewritten subroutines or functions for common mathematical operations. Examples of these logarithmic or trigonometric subroutines. The computer manufacturer usually writes, supplies, and documents these subroutines as part of his software. For instance, they usually come with the Fortran compiler. The functions are universal. Square root is square root regardless of the computer, company, or application.

In business programming it is common belief that each system application is so unique that it must be designed and coded from the beginning. For instance, it has been the belief that the coding scheme that our company, even our plants, use for material classification code or make or buy code, or vendor code or direct labor code, and the algorithms used for processing these data elements are unique to the company or plant. Therefore, prewritten reusable modules cannot be designed, coded, and reused.

A close examination of this reasoning has led us to believe that there are two fallacies in it. The first is that, contrary to common belief, there are at least a few business functions that are sufficiently universal to be supplied by the manufacturer of a Cobol compiler. There are many others applicable to a company, plant, functional area, or application area that could be prewritten.

How many manufacturers supply a Gregorian date edit routine with their compilers?

How many manufacturers supply a Gregorian to Julian date conversion routine or vice-versa, with their compiler?

How many manufacturers supply a date aging routine for such applications as acounts receivable?

In every one of the above cases the application is probably written and rewritten in every business shop in North America.

Manuscript received August 1, 1983.

The authors are with the Missile Systems Division, Raytheon Company, Bedford, MA 01730.

In addition to universal routines, there are company-wide applications. Examples in our company include:

- part number validation routines,
- manufacturing day conversion routines,
- edits for data fields used throughout the company, such as employee number.

Within a functional area in any company, such as manufacturing or accounting, there are routines that can be prewritten, tested, documented, and then copied into a program.

Within a system such as payroll there are often routines that also can be prewritten, such as tax routines.

Yet we believe that the false notion of uniqueness still persists to such a degree that this approach is used at about one-tenth of its potential. We will discuss later the way we have used this concept to produce programs that have an average of 60 percent reusable code.

The second fallacy, in our opinion, involves the program as a whole. It is commonly believed that each business program (as well as each data field) is so unique that it must also be designed and developed from the start. In our opinion there are only six major functions you can perform in a business application program. You can sort data, edit or manipulate data, combine data, explode data, update data, or report on data. By identifying the common functions of these six types of programs, we have produced seven "logic structures." These logic structures give the programmer a head start and provide a uniform approach that is of value later in testing and maintenance.

REUSABLE MODULE DESIGN APPROACH

Our reusable module design approach strategy separates reusable modules into two distinct categories, functional modules and Cobol program logic structures.

Functional Modules

Functional modules are designed and coded fro a specific purpose. Then they are reviewed, tested, documented, and stored on a standard copy library. As mentioned earlier, some of the business routines have universal application, such as date aging; tax routines and others have more limited application to a company, plant, functional area, or system application.

Within our company we classify these functional modules in several Cobol language categories. These categories are:

- file descriptions, i.e., FD's
- record descriptions, i.e., 01 levels in an FD or in working storage

0098-5589/84/0900-0498$01.00

- edit routines, i.e., the data area and procedure code to edit a specific data field
- functional routines, i.e., the data area and procedure code to perform some function, such as left justify and zero fill data elements
- database I/O areas
- database interface modules
- database search arguments
- database procedure division calls.

As can be seen from the above list, we have some modules that are solely data related, such as 01 level record descriptions. The majority of the modules involve both data areas and procedure code. For instance, a database call paragraph, designed to retrieve a specific series of segments, works in conjunction with a program control block module, a segment search argument module, and a database I/O module.

There are approximately 3200 modules in the above categories, supporting over 50 system applications at three plants. By using these functional modules and logic structures, we have been producing programs that average 60 percent reusable code. This produces more reliable programs, with less testing and coding. The maintainability and documentation associated with these applications has also improved substantially because the code is not physically contained in each program.

Cobol Program Logic Structures

A Cobol program logic structure has a prewritten identification division, environment division, data division, and procedure division. It is not a complete program because some paragraphs contain no code, and some record descriptions are also empty, consisting only of the 01 level. It does not however, contain many complete 01 levels and procedure paragraphs.

To illustrate the concept behind logic structures we will describe three types.

The *update* is designed for the classical, sequential update. There is a version with an embedded sort and a version without. The update is designed for situations where the transaction record contains a transaction type field (add, change, or delete). The update logic structures is also designed to accommodate multiple transactions per master record. Error messages to a transaction register are provided for standard errors such as an attempt to add an already existing record. Final totals are also provided, as well as sequence checking.

The *report* logic structure is also written in two versions, one with and one without a sort of the input records prior to report preparation. Major, intermediate, and minor levels of totals are provided for, but more may be added if needed. If multiple sequences of reports are desired, the record can be released to the sort with multiple control prefixes. Paragraphs are also provided for editing, reformating and sequence checking.

The *edit* logic structure is also written in two versions, with or without a sort of the input records. This logic structure was designed for two purposes. One is the editing of input records. In effect the input records are examined based on some criteria and written to the selected (good records) or nonselected (error) files. Another use for this logic structure is the selection of records from a file, based on some criteria, for later use in a report.

Construction of Logic Structures

For each type of logic structure there is a central supporting paragraph.

- For the update program it is the high-low-equal comparison.
- For the report program it is the paragraph that determines which level of control break to take.
- For the selection program it is the select/nonselect paragraph.

Let us consider the report program as an example.

Prior to the control beak paragraph we can identify support functions that must occur in order for the control break paragraph to function. Examples are: get-record, sequence-check-record, edit-record-prior-to-sort, and build-control-keys. These are supporting functions. Other functions such as major-break, intermediate-break, minor-break, roll-counters, build-detail-line, print-detail-line, page-headers, etc., are dependent on the control break or central paragraph.

Obviously many of these paragraphs (functions) can be either completely or partially prewritten.

Our report program logic structure procedure division contains 15 paragraphs in the version without a sort, and 20 paragraphs in the version with a sort.

To further clarify what we mean when we talk about logic structures, it might be helpful to specify some data and procedure division areas in a report logic structure, without an embedded sort.

```
Identification Division
Environment Division
Data Division
  File Section
  Working Storage Section
    01  AA1 – CARRIAGE-CONTROL-SPACING
    01  BB1 – CONSTANTS- AREA
    01  BB2 – TRANSACTIONS- STATUS
    01  BB4 – FILES-STATUS
    01  CC1 – COUNT-AREA
    01  DD1 – MESSAGE-AREA
    01  EE1 – TRANSACTION-READ-AREA
    01  FF1 – KEY-AREA
    01  GG1 – HEAD-LINE1
    01  GG2 – HEAD-LINE2
    01  GG3 – HEAD-LINE3
    01  HH1 – DETAIL -LINE
    01  LL1 – TOTALS-AREA
    01  SS1 – SUBSCRIPT-AREA
    01  TT1 – TOTAL-LINE-MINOR
    01  TT2 – TOTAL LINE-INTER
    01  TT3 – TOTAL-LINE-MAJOR
    01  TT4 – TOTAL-LINE-FINAL
  Procedure Division
    0010 – INITIALIZE
    0020 – MAIN-FLOW
    0030 – WRAP-IT-UP
    0040 – CHECK-CONTROLS
```

0050 – FINAL-BREAK
0060 – MAJOR-BREAK
0070 – INTER-BREAK
0080 – MINOR-BREAK
0090 – PRINT-TOTAL
0100 – ROLL-COUNTERS
0110 – FILL-DETAIL-LINE
0120 – WRITE-PRINT-LINE
0130 – NEW-PAGE-HEADING
0140 – GET-TRANSACTION-RECORD
0150 – TRANSACTION-FORMAT-OR-EDIT
0160 – SEQUENCE-CHECK

The above area, combined together as a program, provide the programmer with a modular functional structure on which to build a report program very easily.

For the update logic structure, the central paragraph is the hi-low-equal control paragraph. Prior to the central control paragraph there must be supporting functions such as get-transaction, sequence-ckeck-transaction, edit-transaction, sort-transaction, get-master, sequence-check-master, build-keys, etc. As a result of this central paragraph you will have functions such as add-a-record, delete-a-record, change-a-record, print-activity-register, print-page-heading, print-control-totals, etc.

Our update logic structure procedure division contains 22 paragraphs in the nonsort version and 26 paragraphs in the version with an embedded sort.

Benefits of Logic Structures

We believe that logic structures have many benefits.

- They help clarify the programmer's thinking in terms of what he is trying to accomplish.
- They make design and program reviews easier.
- They help the analyst communicate with the programmer relative to the requirement of the system.
- They facilitate testing.
- They eliminate certain error-prone area such as end of file conditions since the logic is already built and tested.
- They reduce program preparation time, since parts of the design and coding are already done.

However, we believe that the biggest benefit comes after the program is written, when the user requests modifications or enhancements to the program. Once the learning curve is overcome, and the programmers are familiar with the logic structure, the effect is similar to having team programming with everyone on the same team. When a programmer works with a program created by someone else, he finds very little that appears strange. He does not have to become familiar with another person's style because it is essentially his style.

Research Strategy Used to Test the Concept of Reusability

In August 1976 a study was performed at Raytheon Missile Systems Division to prove that the concept of logic structures was a valid one. Over 5000 production Cobol source programs were examined and classified by type, using the following procedure.

Each supervisor was given a list of the programs that he was responsible for. This list included the name and a brief descrip- of the program along with the number of lines of code.

The supervisor then classified and tabulated each program using the following categories:

edit or validation programs
update programs
report programs

If a program did not fall into the above three categories, then the supervisor assigned his own category name.

The result of classification analysis by program type was as follows:

1089	edit programs
1099	update programs
2433	report programs
247	extract programs
245	bridge programs
161	data fix programs
5274	total programs classified

It should be noted that the bridge programs were mostly select (edit and extract) types, and the data fix programs were all update programs. The adjusted counts were as follows as a result of this adjustment.

1581	edit programs
1260	update programs
2433	report programs
5274	adjusted total programs classified.

The average lines of code by program type for the 5274 programs classified were as follows:

626 lines of code per edit program
798 lines of code per update program
507 lines of code per report program

The supervisors then selected over 50 programs that they felt would be good candidates for study. Working with the supervisors, the study team found that approximately 40-60 percent of the code in the programs examined was redundant and could be standardized. As a result of these promising findings, three prototype logic structures were developed (select, update, and report) and released to the programming community for selective testing and feedback. During this time a range of 15-85 percent reusable code was attained. As a result of this success, it was decided by management to make logic structures a standard for all new program development in three data processing installations. To date over 5500 logic structures have been used for new program development, averaging 60 percent reusable code when combined with reusable functional modules. It is felt at this time that once a programmer uses each logic structure more than three times,that 60 percent reusable code can easily be attained for an average program. We believe this translates into a 50 percent increase in productivity in the development of new programs.

In addition, programmers modifying a logic structure written by someone else agree that, because of the consistent style, logic structure programs are easier to read and understand.

This is where the real benefit lies since most data processing installations are using 60-80 percent of their programming resource to support their maintenance requirements.

To summarize: the basic premise behind our reusability methodology is that a large percentage of program code for business data processing applications is redundant and can be replaced by standard program logic.

By supplying the programmer standard logic in the form of a logic structure we can eliminate 60 percent of the design, coding, testing, and documentation in most business programs. This allows the programmer to concentrate on the unique part of the program without having to code the same redundant logic time and time again.

The obvious benefit of this concept is that after a programmer uses a structure more than three times (learning curve time) a 50 percent increase in productivity occurs. The not so obvious benefit is that programmers recognize a consistent style when modifying a program that they themselves did not write. This eliminates 60-80 percent of the maintenance problem that is caused by each programmer using an individual style for redundant functions in business programs. This is one of the basic problems in maintenance programming today and is causing most programming shops to spend 60-80 percent of their time in the modification mode instead of addressing their new application development backlog.

Conclusion

After studying our business community for over six years, we have concluded that we do basically the same kind of programs year in and year out and that much of this work deals with redundant programming functions. By standardizing those functions in the form of reusable functional modules and logic structures, a 50 percent gain in productivity can be attained and programmers can concentrate on creative problems rather than on redundant ones. In addition to the one time development benefit, the data processing organizations can redeploy 60-80 percent of their resources to work on new systems development applications.

References

[1] R. M. Armstrong, *Modular Programming in COBOL.* New York: Wiley, 1973.

[2] R. Canning, "The search for reliability," *EDP Analyzer*, vol. 12, May 1974.

[3] G. Kapur, "Toward software engineering," *Computerworld*, In-Depth Section, Nov. 1979.

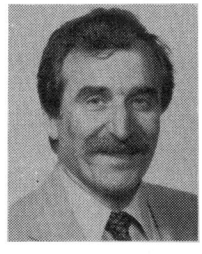
[4] R. Lanergan and B. Poynton, "Reusable code–The application development technique of the future," in *Proc. IBM GUIDE/SHARE Application Symp.*, Oct. 1979, pp. 127-136.

[5] D. Leavit, "Reusabie code chops 60% off creation of business programs," *Computerworld*, pp. 1-4, Oct. 1979.

[6] D. Schechter, "The skeleton program approach to standard implementation," in *Computer Programming Management*. Pennsauken, NJ: Auerbach, 1983.

Robert G. Lanergan attended Northeastern University, Boston, MA, from 1964 to 1967.

He developed and implemented a highly successful software reusability methodology that resulted in a significant reduction in the time and effort required to develop and maintain business systems. At present he is Manager of Information Services with Raytheon Missile Systems Division, Bedford, MA. He presented papers and lectured on the subject of reusability to many national and international organizations (ACM, DPMA, NCC, ACPA, DOD, IBM Guide and Share). In 1980 he was selected as a key employee by Raytheon's Board of Directors for software productivity contributions. His research interests include software productivity, reliability and maintenance issues. He is in the process of authoring a book entitled *How to Alleviate The Software Maintenance Dilemma: Past–Present and Future*.



Charles A. Grasso received the B.A. degree in business from New Hampshire College, Manchester.

He is currently the Manager of the Missile System Division Systems and Programming Organization, Raytheon Company, Bedford, MA. He has been instrumental in the development of major computer applications employing a reusable code/program generator methodology created internally at Raytheon. He is affiliated with the University of Lowell, Lowell, MA, where he teaches advanced structured Cobol.