Proceedings

Application Development Symposium (1979 : Monterey, Calif.)

Monterey Conference Center
Monterey, California
October 14-17, 1979

Sponsored by:

GUIDE International, Inc.
SHARE, Inc.
International Business Machines Corporation

Printed in the United States of America.

Reusable Code— The Application Development Technique of the Future

by Robert G. Lanergan and Brian A. Poynton
Raytheon Company
Burlington, Massachusetts

1. Introduction

During the last three years, Raytheon Missile Systems Division business data processing organization has developed, implemented and perfected a unique reusable code methodology to accelerate applications development.

Our approach (Figure 1) has been to establish a process which eliminates 40-60% of the redundancy in business applications development. By using reusable code to replace this redundancy, we have been able to develop new applications faster and at a lower cost than with traditional methods.

Our Approach

Establish a Process to Eliminate Redundancy and Accelerate Application Development

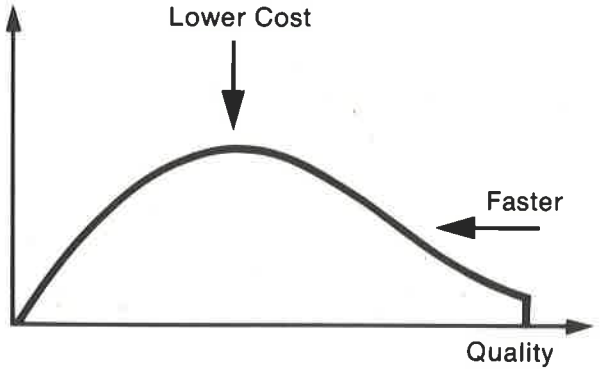

Figure 1.

Three years of experience (Figure 2) has proved to our organization that when people use reusable code and software tools and aids, application development and maintenance can be improved substantially.

In some cases, new programs have been developed forty times faster than by using the traditional development methods. In another case, a major IMS teleprocessing dock-to-stock inventory system estimated to take 126 man-months of effort took only 36 man-months. The 62 programs in this system averaged 65% reusable code.

Some of the other results our reusable code methodology has provided our organization are (Figure 3):

Three Years of Experience Prove . . .

When People Are Managed and Trained in:

- Structured Technology
- Reusable Code Techniques
- Productivity Aims and Services

. . . Application Development and Maintenance Can be Improved Substantially.

Figure 2.

Reusable Code Brings Results

- Averaged 60% Reusable Code for 5 New IMS Applications
- Averaged over 40% for 500 New Programs
- Increased Maintainability of 3000 Old Programs
- Built Library of over 1500 Reusable Code Modules
- Trained 120 Programmer/Analysts
- Reduced Learning Curve for New Employees

Figure 3.

- Five new IMS applications composed of over 250 programs have averaged over 60% reusable lines of code.
- Over 500 new non-IMS programs have averaged over 40% reusable lines of code.
- Over 1500 reusable code modules have been developed, certified and stored in a centralized library for future applications development.

- Our 120 programmer/analysts are producing more maintainable programs faster as opposed to using their own individual methods. As high as forty times faster in some cases.
- Entry level trainee programmers are more productive than most journeyman programmers at the end of three months experience.

Realizing that no one technique can solve the many problems facing application development today, we have developed other tools and aids that support the reusable code methodology. They are (Figure 4):

We Provide Support Technology

Indexing Scheme
Certification Procedures
"Where-Used" System
. . . to Foster Communication

Figure 4.

- Reusable Code Index System that enables the accessing of reusable code based on up to five levels of key words.
- Certification Procedures, Reusable Code Standards and a Centralized Library that enables reusable code to be used by separate physical locations.
- Where-Used System that provides information on reusable modules by program and by module.

Some additional tools and aids that have increased applications development productivity are (Figure 5):

Storeroom of Productivity Aids

Cobol Standardizer
Utility Tool
Test Generator
Reconciler
Interactive Cobol Generator
Disk Space Manager
Debugging Tool
Revision Controller
Revision
Controller

Figure 5.

- COBOL Reformat Software that standardizes source code and increases the readability of source programs.
- Test Data Generator that provides the ability to generate test data and files easily.
- Data File Reconciler that aids the reconciliation between two files or programs notating the differences.
- Disk Space Management System that monitors the utilization of library work space by supervisor.
- Program Revision Controller that maintains revision control over source and executable programs notating the differences.
- Debug Tool that identifies most common ABEND errors in one run.

Our Interactive COBOL Report Generator has provided the nucleus for a "Programmer's Work Bench" (Figure 6). Through prompting, COBOL source code is developed, tested and executed in one interactive session, usually in less than one hour. This process is aided by our Test Data Generator and Debug Tools along with our reusable code modules.

The Programmer's Work Bench

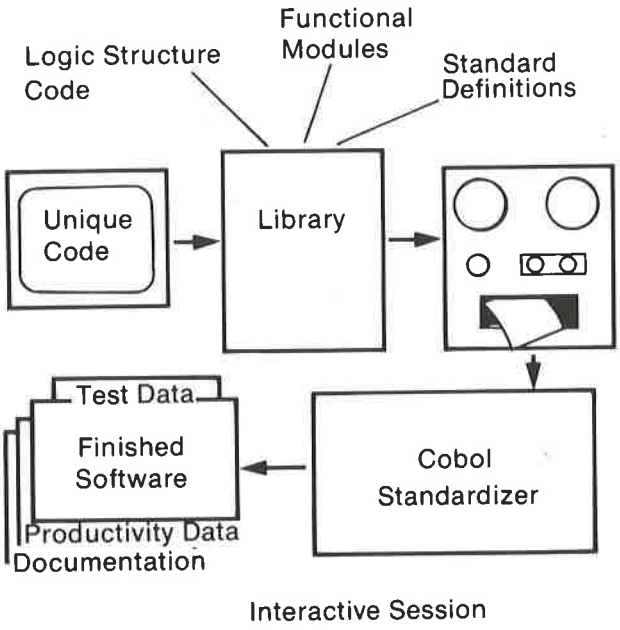

Figure 6.

To train both our 120 programmer analysts and our entry level programmers we developed two in-house training programs (Figure 7).

Experienced programmers are required to complete 10 half-day sessions on how to use and develop reusable code. In addition, they receive training in our

productivity tools and aids. We also provide follow up assistance after programmers complete the training program.

Built Two-Tier in-House Training Program

Retrofit Craftsman–Programmers

- Trained 120 Programmer Analysts
- Completed 10 Half-Day Sessions
- Provided Follow-Up Assistance

Bring On-Board New Employees

- Covers Two Months, Full Time
- Details Complete Inventory System
- Provides Top-Down Development, Using Structured Technology
- Comprehensive: 21 Different Features
- Reach High Level of Productivity in 3 Months

Figure 7.

Entry level programmers are put through a two month full time reusable code training program. They design a complete inventory system using structured technology and reusable code techniques. In addition, they receive training in our productivity tools and aids and reach a high level of productivity in 3 months. They out-perform most journeyman programmers in a short period of time.

Another component of our reusable code program is our multi-level productivity measurement system (Figure 8). This system provides various levels of management with the number of reusable modules and lines of code used by each programmer and supervisor. This system allows us to measure our progress relative to reusable code utilization and productivity tools.

Now that we have given you some background about some of the tools and aids that support our reusable code methodology, we would like to cover the following topics.

- Why reusable code is not used more often.
- Our implementation of this technique.
- The construction of logic structures.
- The benefits derived from logic structures.
- How we tested the concept of logic structures.
- Our conclusions after using reusable code over the last three years.

Provide Multi-Level Measurement

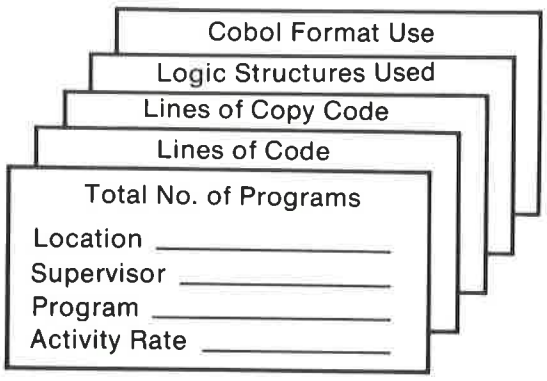

Figure 8.

2. Why Reusable Code Is Not Used More Often

In business programming, (Figure 9A) it is common belief that each application, each program, and each data field is so unique that it must be written from scratch. For instance, it has been the belief that the coding scheme my company or even my plant uses for Material Classification Code, Make or Buy Code, Vendor Code, Direct Labor Code, and the algorithms used for processing these fields are unique to the company or plant. Therefore, we cannot use any prewritten code.

Why Reusable Code is Not Used More Often

- Craftsman Programmer Believes That Each Program is Unique
- Our Premise

4 Major Functions in Business Programming

- Sort Data
- Select and/or Edit Data
- Update Data
- Report Data

40—60% of The Code in Business Programs Can Be Standardized

Figure 9A.

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 which are universal. There are many others which are applicable to a company, plant, functional area, application area, and which could be pre-written.

Examples of universal routine are:

- Gregorian date edit routine
- Gregorian-to-Julian date conversion routine, and vice-versa
- Payroll tax routines
- Date ageing routines

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

In addition to universal functional routines, there are company wide routines. 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, Purchase Order Number, etc.

The third group is called functional area routines. Some examples are customer, vendor and invoice number routines.

For those companies doing work on Government contracts there would be another group. Examples are: Federal Stock Number, Federal Vendor Code, Interchangeability Code and Shelf Life Code.

All of the above functional routines can be pre-written, tested, documented and reused for applications development. Yet, most data processing installations use little or no reusable code for their applications development. I will discuss later the way we have implemented this concept to produce programs which have 40-60% pre-written code.

The second fallacy, in our opinion, involves the program as a whole. It is commonly believed that each business program is so unique that it must be written from scratch. In our opinion, there are only four things you can do with a business application computer program: you can sort data, select and/or edit data, update data, or report on data. Since the manufacturer supplies the sort, we can eliminate that from consideration.

Every other program is either one of the remaining three, or a combination of them. By identifying the redundant functions of these three types of programs, we have produced three "logic structures", one for each type of program. These logic structures give the programmer 40-60% of the COBOL source code and help to provide a uniform approach which is of later value in testing, debugging and maintenance.

3. Implementation of the Reusable Code Technique

Reusable code is categorized into two types (Figure 9B), functional modules and logic structures.

What is Reusable Code?

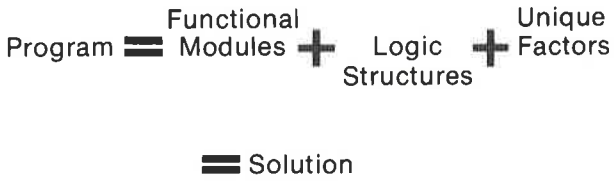

Figure 9B.

The first type, functional modules, are designed and written for specific purpose. They are tested, certified, documented and stored on a central library for use by three locations. They are not hand-crafted into each program. Maintenance is isolated from the programs because each module is copied into the program at compilation time. If a module is used in 30 programs and a modification is required, it only has to be modified once, not 30 times. The savings in maintenance time alone should be obvious.

Within our organization, we classify these routines into several categories. These categories and the number of modules in each category (Figure 10) are:

• File Definitions/Record Descriptions	275
• Data Field Edits	410
• Functional Modules	89
• Data Base Input Definitions	46
• Data Base Specification Blocks	68
• Segment Search Arguments	47
• Data Base Calls	567

Magnitude of Savings

• File Definitions	275
• Data Field Edits	410
• Functional Modules	89
• Data Base Input Definitions	46
• Data Base Specifications Blocks	68
• Segment Search Arguments	47
• Data Base Input Calls	567
	1502

➧ Maintenance Isolated from Program

Figure 10.

As you can see from the above list, we have some modules which are solely data related, such as file definitions. The majority of the modules involve both data areas and procedure code. For instance, a data base call module designed to retrieve a specific series of segments works in conjunction with a data field edit module and its associated work area, data base specification block, segment search argument module and a data base input definition. One example (Figure 11) would be the requirement to access a particular project segment in a part number data base—a typical two-level edit and data base call. To perform this task, an average of 150 lines of code must be written. Using the technique just described, only 10 to 15 lines of unique code must be written (Figure 12), an order of magnitude in terms of development savings. In addition, the code is written once, tested and proven by our best programmers and is available thereafter to other programs. This technique has been largely responsible for developing new IMS applications with over 60% reusable code with minimum maintenance after the system is operational.

Why Are We Unique?

Control of Data

IMS Data Base Management Routines

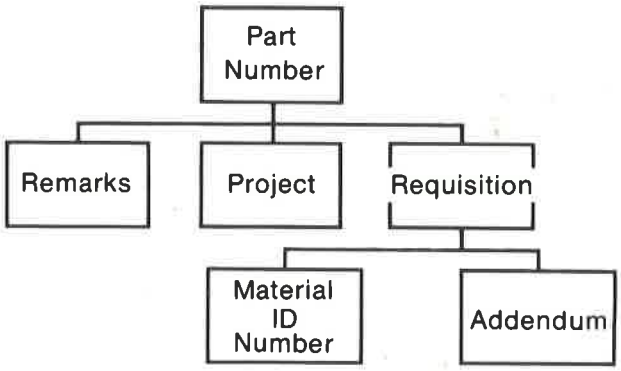

Figure 11.

The second type of reusable code is called logic structures. A logic structure (Figure 13) 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, however, contain many complete 01 levels and procedure paragraphs. It is designed hierarchically and contains small functional paragraphs. Figure 14A illustrates the number of functions and lines of code for each of our six logic structures.

Why Are We Unique?

Save Time and Dollars

IMS Call with Two Edits

Hand-Crafted Code	Reusable Code
150	10

- Standard Code Is Written Once, Tested and Proven by Best Programmers
- Available Thereafter to All Programs

Figure 12.

What is a Logic Structure?

- Four Cobol Divisions
- Not a Complete Program
- Not a Collection of Paragraph Names
- Hierarchical
- Functional

Figure 13.

How Many Types Are There?

	No. of Functions	Procedure Lines of Code	Nonprocedure Code
Edit	11	213	253
Edit/Sort	12	341	286
Update	22	517	390
Update/Sort	26	621	418
Report	16	263	259
Report/Sort	20	359	286

Figure 14A.

The *update* logic structure 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). This can be easily overridden for collator type updates. The update logic structure 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, reformatting and sequence checking.

The *select* and/or edit logic structure is also written in two versions, one with and one without a sort. 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 non-selected (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. In practice one of its most common uses has proven to be special purpose runs against master files when customers request specific non-standard modifications to existing data.

Combinations: We rank these logic structures in the order update, report and select (Figure 14B) when determining which one should be used for a combined program. The highest included function determines which logic structure to use. For instance, an update with editing of transactions would indicate adding edit code to the update, rather than adding update logic to a select logic structure.

How We Determine Which Logic Structure to Use

Ranked in the Order Of

- Update
- Report
- Select and/or Edit

Logic Structure That Provides the Most Reusable Code is the Deciding Factor

Figure 14B.

4. Construction of Logic Structures

Rather than explain this top down or bottom up, we'll describe the Logic Structures "inside-out".

In the 8th Century in Europe a house was supported by two sets of curved timbers. These were called "crucks".

For each type of logic structure there is a "crucks" paragraph, 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/non-select paragraph.

Let us consider the report logic structure (Figure 15) as an example. We can identify many functions such as major-break, intermediate-break, minor-break, roll-counters, build-detail-line, print-detail-line, page-headers, etc., which are dependent on the control break or central paragraph.

Prior to the control break paragraph, we can identify support functions which 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.

When you think of it this way, you can see that many of these paragraphs (functions) can be either completely or partially pre-written.

Our report program Logic Structure Procedure Division contains 16 paragraphs in the version without a sort, and 20 paragraphs in the version with a sort.

For the update logic structure (Figure 16), the "crucks" is the hi-low-equal control paragraph. 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. Prior to the central control paragraph there must be supporting functions such as get-transaction, sequence-check-transaction, edit-transaction, sort-transaction, get-master, sequence-check-master, build-keys, etc.

Our update logic structure Procedure Division contains 22 paragraphs in the non-sort version and 26 paragraphs in the version with an embedded sort.

5. Why Use Logic Structures?

Some of the benefits that logic structures provide are (Figure 17):

- Consistent organization of redundant logic
- 40-60% pre-written standardized code
- Standard names and labels
- Internal documentation
- Ease of maintenance
- Ease of learning transfer
- Elimination of one-man craftsmanship
- Ease of testing and debugging
- Facilitate communications between people

Report Logic Structure

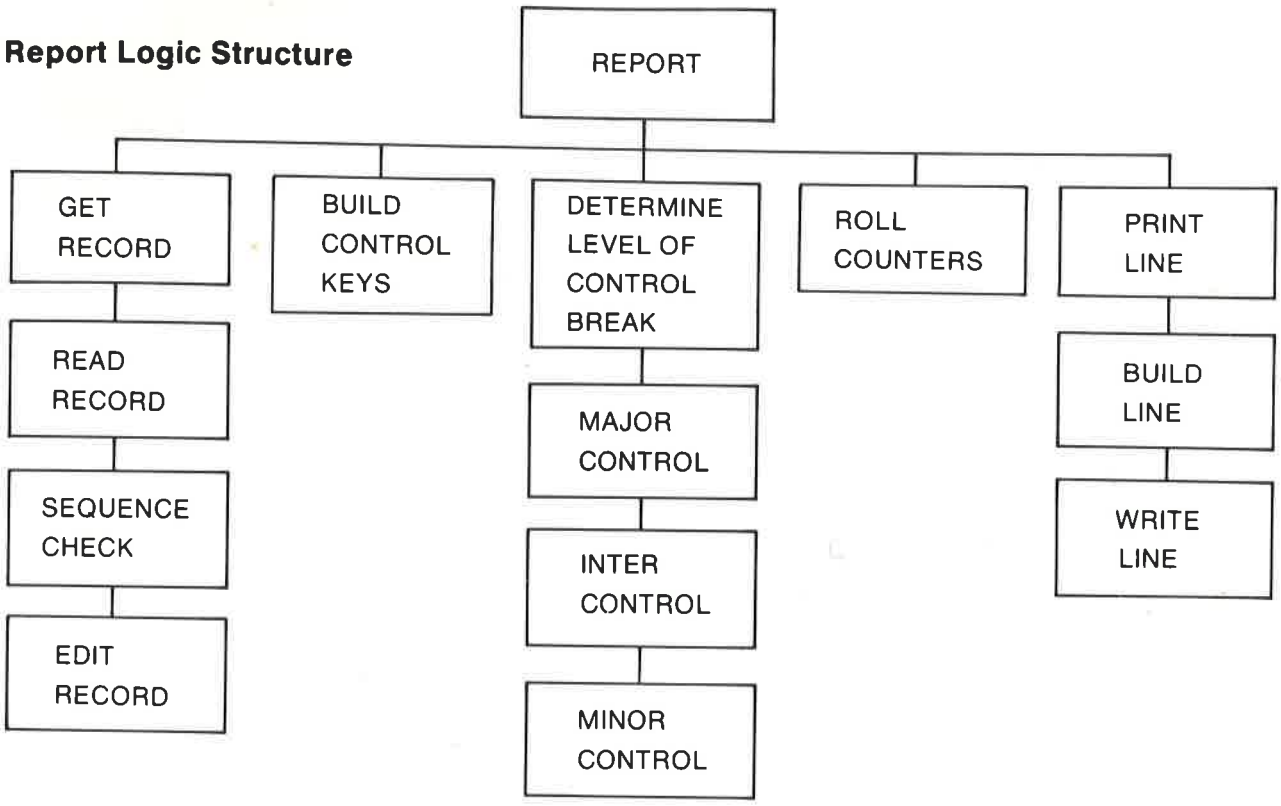

Figure 15.

Update Logic Structure

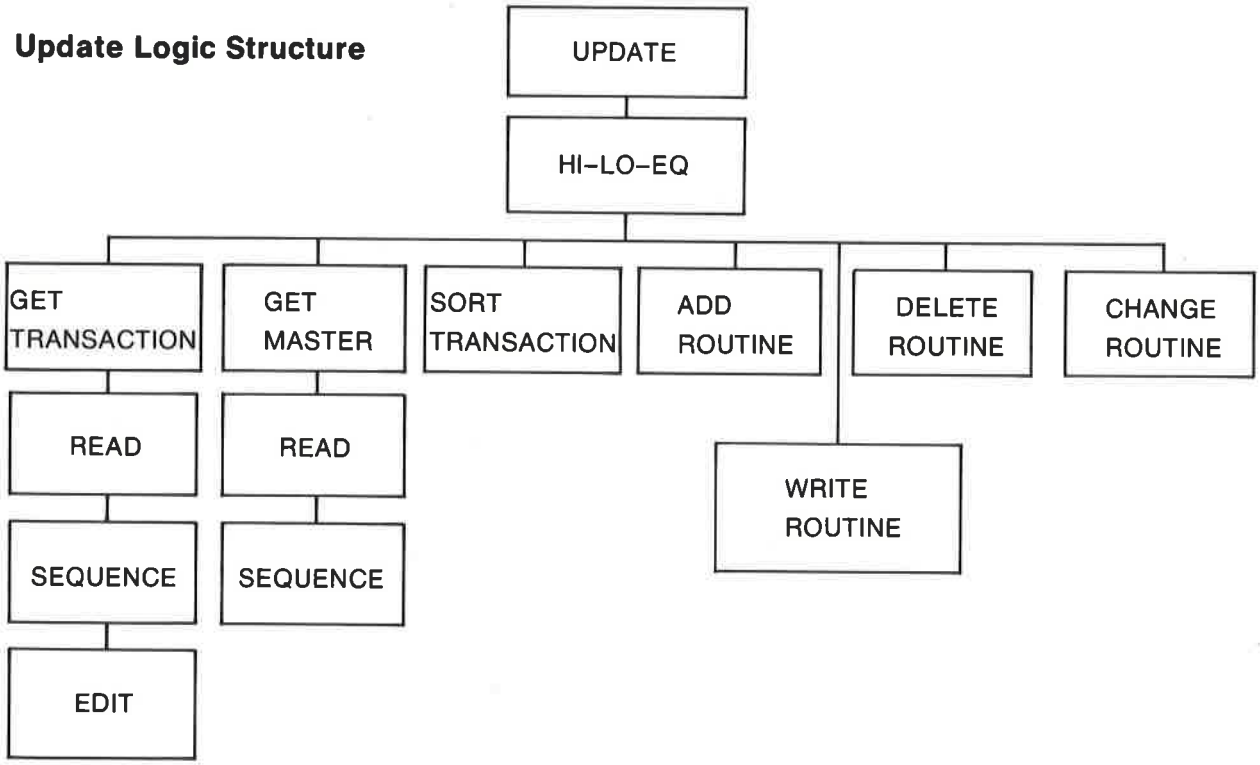

Figure 16.

Why Use Logic Structures?

- Consistent Organization of Redundant Logic
- 40–60% Prewritten Standardized Code
- Standard Names and Labels
- Internal Documentation
- Ease of Maintenance
- Ease of Learning Transfer
- Eliminates One-Man Craftsmanship

Figure 17.

However, we believe 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.

6. How We Tested the Concept of Logic Structures

In August of 1976, a study was performed at Raytheon Missile Systems Division to prove that the concept of logic structures was a valid one (Figure 18). Over 5000 production COBOL Source programs were examined and classified by type using the following procedure.

How We Tested the Concept of Logic Structures

- Classified Over 5000 Programs in 1976
  1761 Selection and/or Edits
  1260 Updates
  2433 Reports
- Analysis Showed Potential for 40–60% Reusable Code
- Developed and Tested 3 Prototype Logic Structures
- Attained 15–85% Reusable Code During Test Period

Figure 18.

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

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

- Selection and/or Edit 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 Selection and/or Edit Programs
1099 Update Programs
2433 Report Programs
427 Extract Programs
245 Bridge Programs
161 Data Fix Programs

5454 Total Programs Classified

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

1761 Select and/or Edit Programs
1260 Update Programs
2433 Report Programs

5454 Adjusted Total Programs Classified

The average lines of code by program type for the 5454 programs classified were as follows:

626 Lines of code per Select Programs
798 Lines of code per Update Programs
507 Lines of code per Report Programs

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% of the logic 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 - REPORT) and released to the programming community for selective testing and feedback. During this time, a range of 15-85% 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 our three data processing installations.

To date, logic structures have been used to develop over 500 new programs with an average of 42% reusable code (Figure 19). It is felt at this time that, once a programmer uses each logic structure three times, 40-60% reusable code can easily be attained for an average program.

What Are the Results of Logic Structures?

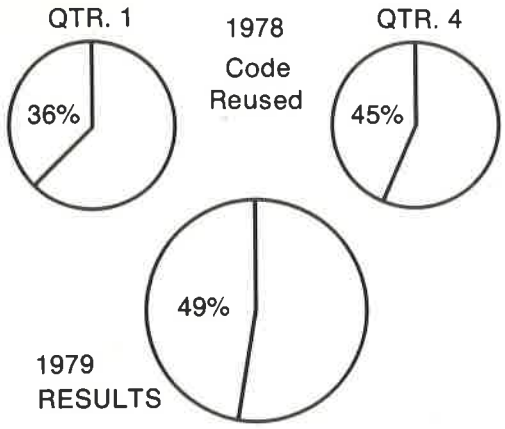

Figure 19.

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 D/P installations are spending a majority of their time in the maintenance mode.

To summarize: the basic premise behind logic structures is that a large percentage of program code for business data processing applications is redundant and can be replaced by standard program logic. This standard logic can be used to address the four fundamental functions that business programs can be classified under.

- Sorting
- Selection and/or Edit
- Updating
- Reporting

By supplying the programmer standard logic in the form of a logic structure, we can eliminate 40-60% of the design, coding, keypunching, 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 sharp increase in productivity and reliability occurs. This occurs because 40-60% of the programs do not have to be developed from the beginning. The not-so-obvious benefit is that programmers recognize a consistent style when modifying programs that they themselves did not write. This eliminates much of the craftsmanship 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 60-85% of their time fixing and enhancing old systems.

7. Our Conclusions After Using Reusable Code for Three Years

After using reusable code for over three years in three separate data processing installations, the results for both logic structures and reusable functional modules are:

- Logic structures have provided an average of 42% reusable code over a three-year time frame for over 500 new programs (Figure 19).
- Five new major IMS DATA BASE applications composed of over 250 programs have averaged over 60% reusable code (Figure 20).

Our conclusions after analyzing our business application programming community for over three years are (Figure 21):

What Are the Results of Functional Code Modules?

5 New IMS Data Base Applications Averaged Over 60% Reusable Code

- 251 Programs Composed of 351K Lines of Code
- Only 25K Lines of Code Were Required to Produce 21 5K Reusable Lines of Code

Figure 20.

The Results Are In

- Programming Remains a Craft
- Craftsmanship at Heart of Software Development
- 3 Basic Types of Programs
- 40–60% of Code Can be Standard
- Redundancy Causes Maintenance Dilemma
- Reusable Code = Productivity, Reliability

Figure 21.

- All programmers are craftsmen, and craftsmanship is the reason behind the software development productivity problem. Until management recognizes this and implements a methodology such as our "REUSABLE CODE METHODOLOGY", no substantial gains in productivity can be attained.
- Functional modules and logic structures can provide 40-60% of reusable code and will accelerate applications development substantially if used correctly.
- Redundant functions used by individual programmers cause much of the maintenance dilemma because of the individual styles that each programmer uses to solve the same problem.
- Reusable code increases productivity, decreases maintenance, reduces craftsmanship, and provides a way of measuring programming productivity.
- Trainee programmers trained in the reusable code methodology can out-perform most journeyman programmers after three months experience.
- A plan, management commitment and a measurement system is necessary if you are going to be successful in implementing any kind of productivity program.

We would like to end on this note. It has been our experience over the last three years that no real gains can be made in programming productivity until data processing managers enforce more discipline over the way software is developed and maintained. Our experience and success with the reusable code methodology has proved to us that this is one way to enforce that discipline.