SOFTWARE ENGINEERING WITH STANDARD ASSEMBLIES

R. Lanergan and B. Poynton

Raytheon Company, Missile Systems Division
Bedford, Massachusetts 01730

The techniques of functional modularity are utilized in a business data processing environment to prepare modules for use in multiple applications. These modules are tested, documented, and stored in a library for later use. Three major applications using this technique averaged 60 percent re-used code. In addition, business programs have been categorized into three types and pre-written "logic structures" have been prepared for each type, to serve as a framework to be completed by the addition of copied code and unique code.

Key words: Logic structures; re-usable code; standard modules.

1. Introduction

It is common practice when writing scientific programs to use pre-written subroutines or functions for common mathematical operations. Examples of these are logarithmic subroutines, 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 application 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 or Make or Buy Code, or Vendor Code or Direct Labor Code, and the algorithms used for processing these fields are unique to the company or plant. Therefore, I cannot use any pre-written code.

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 sufficiently universal to be supplied by the manufacturer with the COBOL compiler. There are many others which are applicable to a company or plant or functional area or application area and which could be pre-written.

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 Aged Accounts Receivables?

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

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 which can be pre-written, tested, documented and then copied into a program.

Within a system such as configuration management there are often routines which also can be pre-written.

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 implemented this concept to produce programs which

have 40 to 60 percent pre-written 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 be written from scratch. In our opinion there are only four things you can do with a business computer program. You can sort data, select 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 common features of these three types of programs, we have produced three "logic structures", one for each type of program. These logic structures give the programmer a running start, and help to provide a uniform approach which is of later value in debugging and maintenance.

We will explain our implementation of these concepts, the construction of logic structures, the benefits derived from the use of these concepts and the procedure used to test the concept.

2. Implementation of This Concept

2.1. Pre-Written Modules

These are modules which are written for a specific purpose. Then they are walked through, tested, documented and put in a library. Examples in a scientific environment would be routines for trigonometric functions. As mentioned earlier, some of the business routines would have universal application, such as date aging, and others would have more limited application to a company, plant, functional area or system application.

Within our company we classify these routines into several categories. These categories are:

File descriptions i.e. FDs

Record descriptions i.e. 01 levels in an FD or in Working Storage

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 Base I/O area specifically IMS I/O area descriptions

Data Base interface module, specifically IMS Program Control Blocks

Data Base Search Arguments, specifically IMS Segment Search Argument

Data Base Procedure Division Calls

As you can see from the above list we have some modules which 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 data base 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 Data Base I/O Module.

We have approximately 1000 modules in the above categories, for several data bases applications at multiple plants. By using those we have been producing programs which contain approximately 60 percent re-usable code. Naturally this produces more reliable programs with less testing and coding. Maintainability and documentation are also improved.

2.2. Logic Structure

A logic structure has a pre-written 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.

We have written six Logic Structures:

The update (Figure 1) 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 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. In practice one of its most common uses has proven to be special purpose runs against master files when customers request specific nonstandard modifications to existing data.

Combinations. We rank these logic structures in the order - update - report - select - when determining which one should be used for a combined program. (Figure 2) 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.

3. Construction of Logic Structures

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

In the eighth century in Europe a house was (Figure 3) 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 (Figure 4).

For the selection program it is the select/nonselect paragraph.

Let us consider the report program as an example. We can identify many functions (Figure 4a) such as major break, intermediate break, minor break, roll counters, built 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. (Figure 4b)

When you think of it this way you can see that 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.

For the update logic structure the "crucks" is the high-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 nonsort version and 26 paragraphs in the version with an embedded sort. By the way, I should mention at this point that a logic structure should be tailored to the audience. The paragraph shown is recommended as the right way to code in more than one text on structured programming (Figure 5). But if your people have not had the training necessary to let them appreciate this approach they may prefer or feel more comfortable with a slightly different but equally effective type of code. (Figure 6)

4. 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 walk-throughs easier.

They help the analyst communicate with the programmer relative to the requirements of the system.

They facilitate debugging.

They eliminate certain error prone areas 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 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.

5. Procedure Used To Test Concept

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. 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 description 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
5454	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.

1761 Select 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 to 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 - report) and released to the programming community for selective testing and feedback. During this time a range of 15 to 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 our three data processing installations. To date over 200 logic structures have been used for new program development with an average of 42 percent reusable code. It is felt at this time that once a programmer uses each logic structure three times that 40 to 60 percent reusable code can easily be attained for an average program. We believe this results in a 30 to 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 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
Updating
Reporting

By supplying the programmer standard logic in the form of a logic structure we can eliminate 15 to 85 percent 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 15 to 85 percent of the program does 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 ownership 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 to 85 percent of their time fixing and enhancing old programs.

6. Conclusion

After studying our business programming community for over two years, we have concluded that we do the same work year in and year out and much of this work deals with redundant programming functions. By standardizing those functions in the form of reusable code and logic structures, gains in productivity and reliability can be attained and the programmers can concentrate on the real creative problem rather than the redundant one. In addition to the one time development benefit, the programming community can eliminate much of the programmer's individual styles which are the root of the maintenance dilemma that most business data processing installations are faced with today. (Figures 7 through 23)

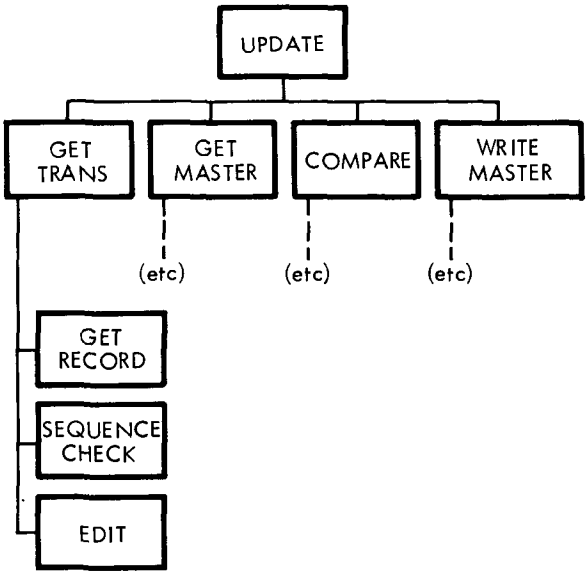

Figure 1 - Update Logic

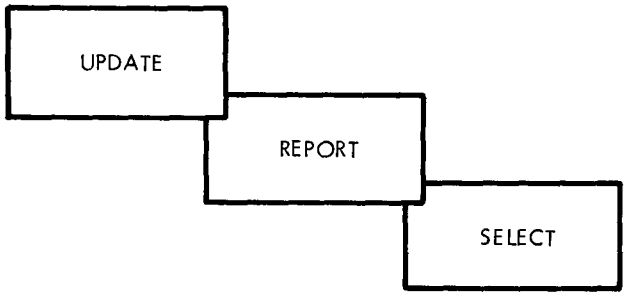

Figure 2 - Combined Structures

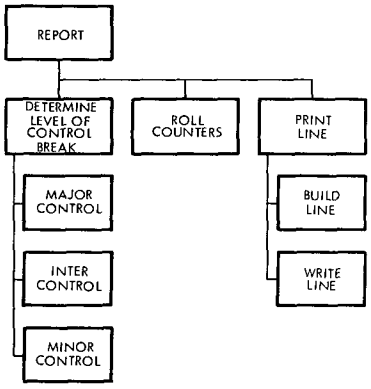

Figure 4a - Report Logic

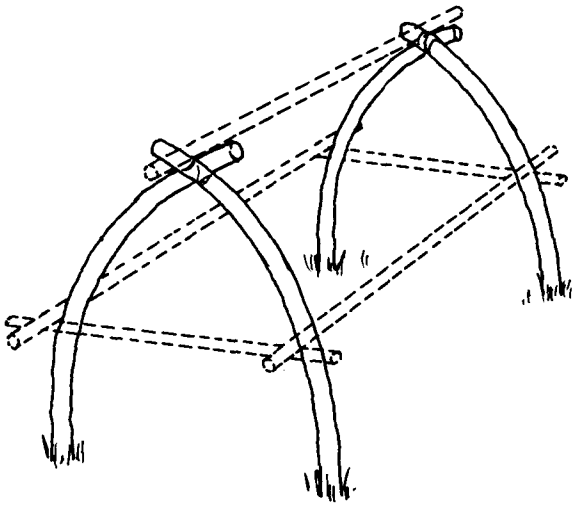

Figure 3 - 8th Century Cruck

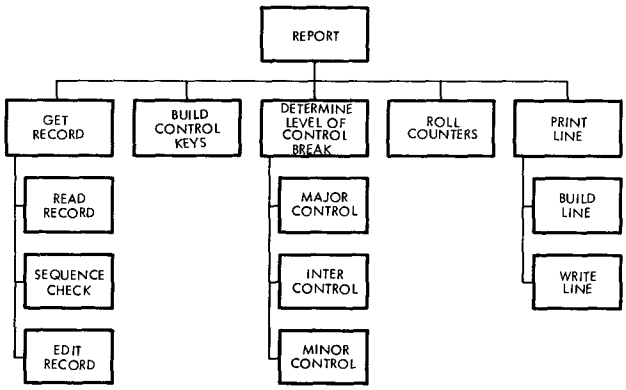

Figure 4b - Report Logic

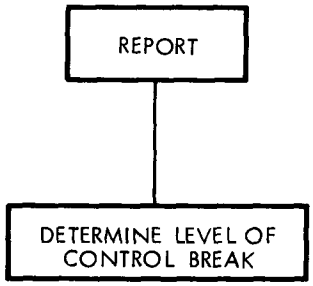

Figure 4 - Report Logic

```
                0070-PROCESS-COMPARE.
000300              IF  FF1-CURR-MASTER-KEY IS LESS THAN FF1-CURR-TRANS-KEY
000400                  PERFORM 0080-TRANS-HIGH THRU 0085-TRANS-HIGH-EXIT
000500              ELSE
000600                  IF  FF1-CURR-MASTER-KEY IS EQUAL TO FF1-CURR-TRANS-KEY
000700                      AND FE1-DELETE-CODE
000800                      PERFORM 0090-DELETE-A-RECORD
000900                              THRU 0095-DELETE-A-RECORD-EXIT
001000                  ELSE
001100                      IF  FF1-CURR-MASTER-KEY IS EQUAL TO
001200                              FF1-CURR-TRANS-KEY
001300                          AND FE1-CHANGE-CODE
001400                          PERFORM 0110-CHANGE-A-RECORD
001500                                  THRU 0115-CHANGE-A-RECORD-EXIT
001600                      ELSE
001700                          IF  FF1-CURR-MASTER-KEY IS GREATER THAN
001800                                  FF1-CURR-TRANS-KEY
001900                          AND FE1-ADD-CODE
002000                          PERFORM 0100-ADD-A-RECORD
002100                                  THRU 0105-ADD-A-RECORD-EXIT
002200                      ELSE
002300                              PERFORM 0130-TRANS-IN-ERROR
002400                                      THRU 0135-TRANS-IN-ERROR-EXIT.
                0075-PROCESS-COMPARE-EXIT. EXIT.
```

Figure 5 - Structured Modular Code

```
000300 0080-PROCESS-COMPARE.
000100     IF FF1-CURR-MASTER-KEY IS LESS THAN FF1-CURR-TRANS-KEY
000200         PERFORM 0090-TRANS-HIGH THRU 0095-TRANS-HIGH-EXIT
000300         GO TO 0085-PROCESS-COMPARE-EXIT.
000400     IF FF1-CURR-MASTER-KEY IS EQUAL TO FF1-CURR-TRANS-KEY
000500         AND EE1-DELETE-CODE
000600         PERFORM 0100-DELETE-A-RECORD THRU
000700                 0105-DELETE-A-RECORD-EXIT
000800         GO TO 0085-PROCESS-COMPARE-EXIT.
000900     IF FF1-CURR-MASTER-KEY IS EQUAL TO FF1-CURR-TRANS-KEY
001000         AND EE1-CHANGE-CODE
001100         PERFORM 0120-CHANGE-A-RECORD THRU
001200                 0125-CHANGE-A-RECORD-EXIT
001300         GO TO 0085-PROCESS-COMPARE-EXIT.
001400     IF FF1-CURR-MASTER-KEY IS GREATER THAN FF1-CURR-TRANS-KEY
001500         AND EE1-ADD-CODE
001600         PERFORM 0110-ADD-A-RECORD THRU 0115-ADD-A-RECORD-EXIT
001700         GO TO 0085-PROCESS-COMPARE-EXIT.
001800     PERFORM 0150-TRANS-IN-ERROR THRU 0155-TRANS-IN-ERROR-EXIT
000300 0085-PROCESS-COMPARE-EXIT. EXIT.
```

Figure 6 - Less Structured Modular Code

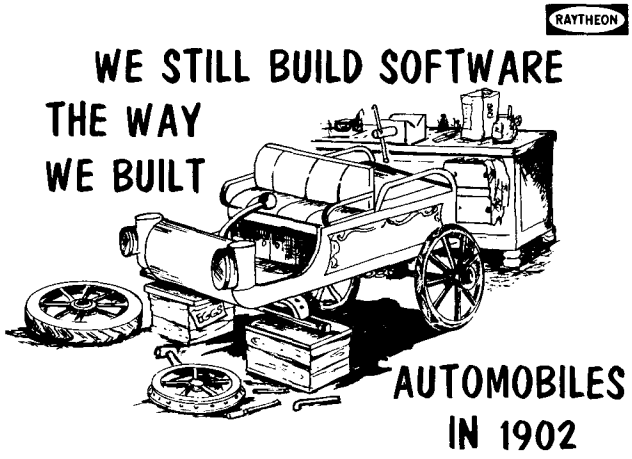

Figure 7

THAT MEANS...

- *No Formal Design*
- *One Man Operation*
- *Piece by Piece*
- *Hand Made*

Figure 8

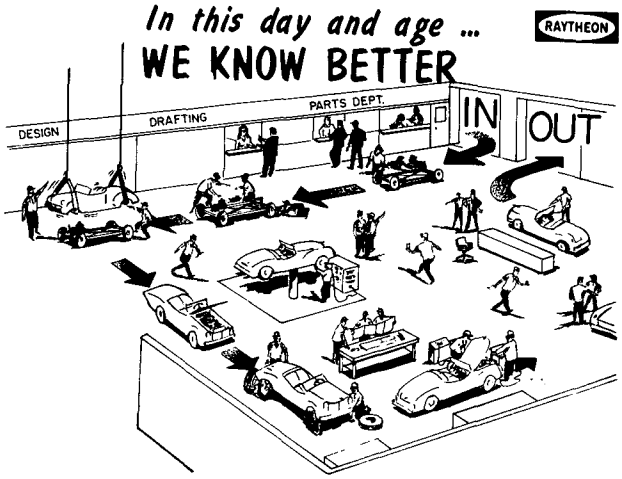

Figure 9

BECAUSE WE KNOW...

RAYTHEON

- *How to Design*
- *How to Manage Specialization of Labor*
- *How to Pre-Build Components*
- *How to Plan And Work in Parallel*

Figure 10

AREAS AFFECTED....

RAYTHEON

- *Design*
- *Coding*
- *Testing*
- *Support*
- *Documentation*
- *Maintenance*

Figure 11

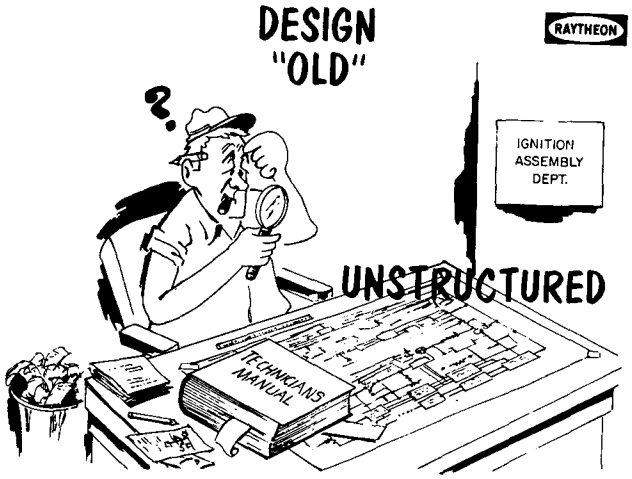

Figure 12

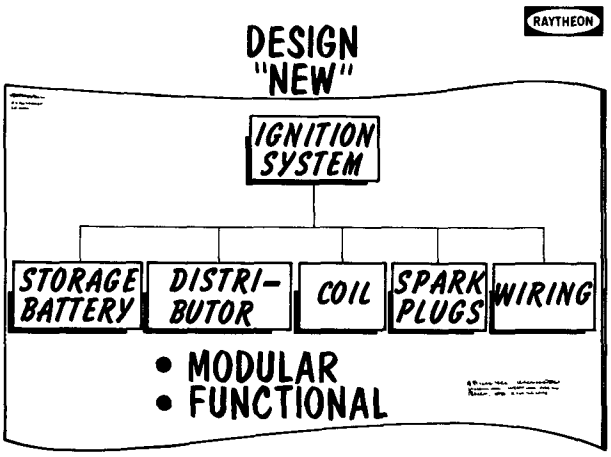

Figure 13

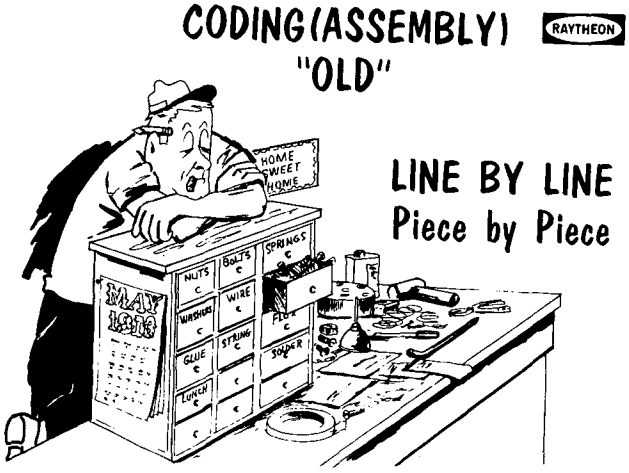

Figure 14

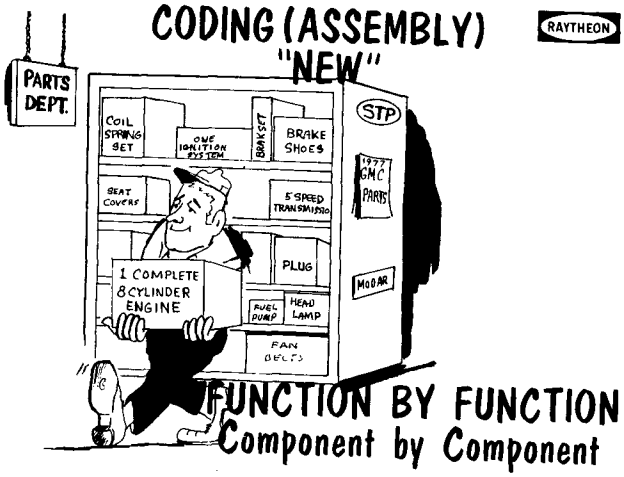

Figure 15

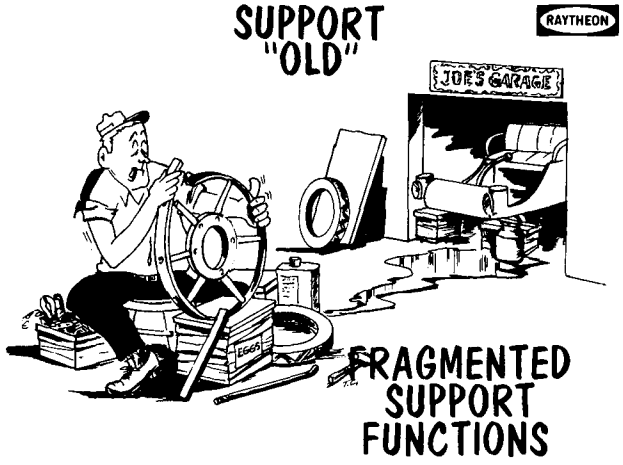

Figure 16

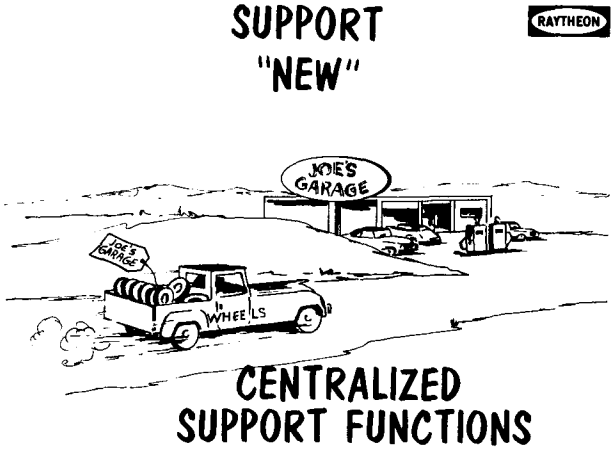

Figure 17

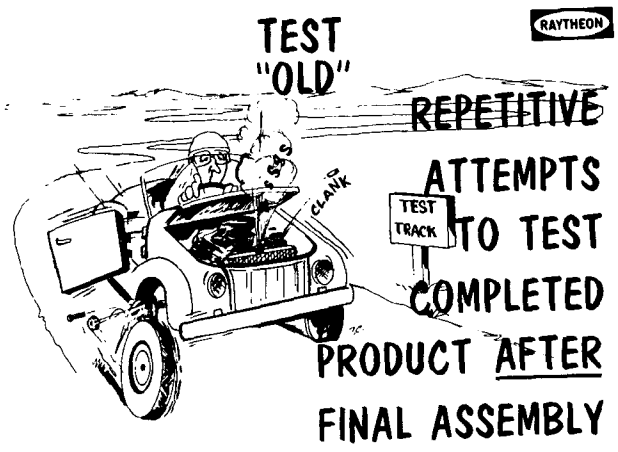

Figure 18

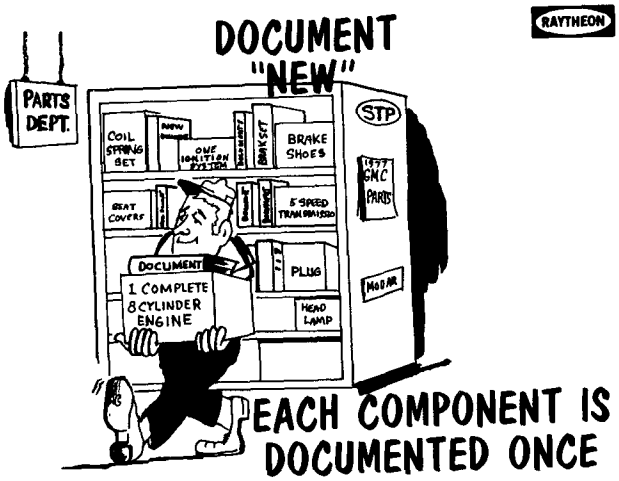

Figure 21

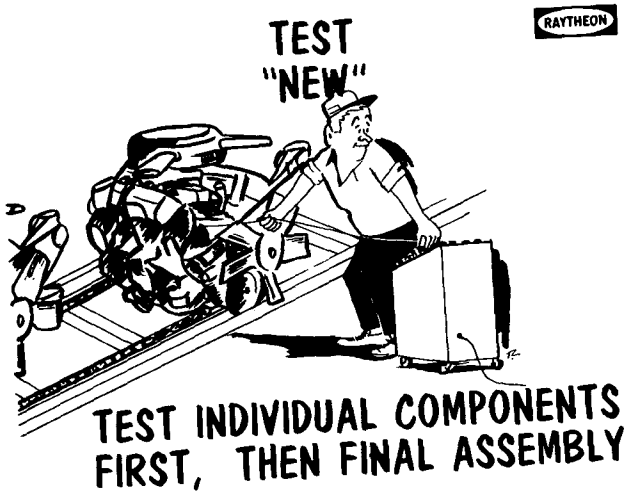

Figure 19

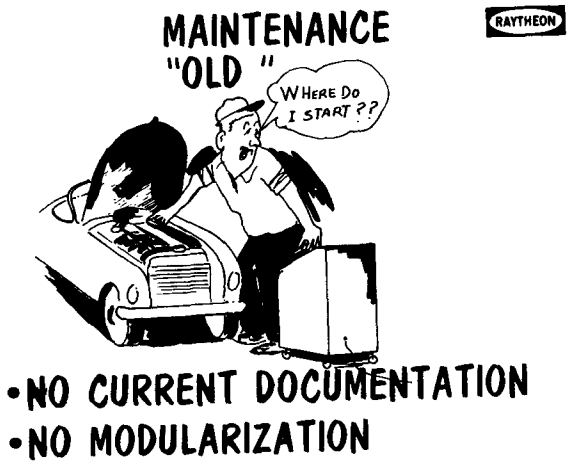

Figure 22

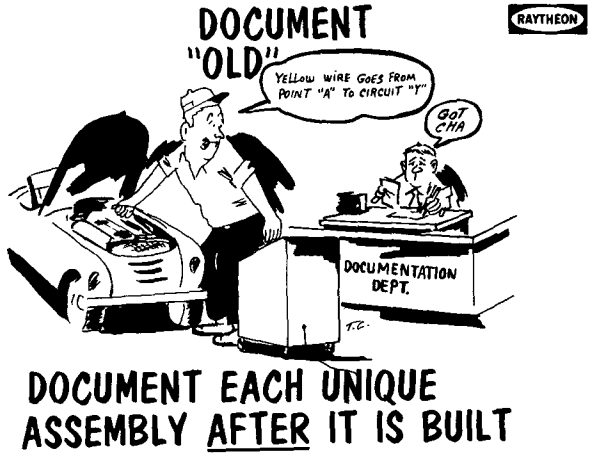

Figure 20

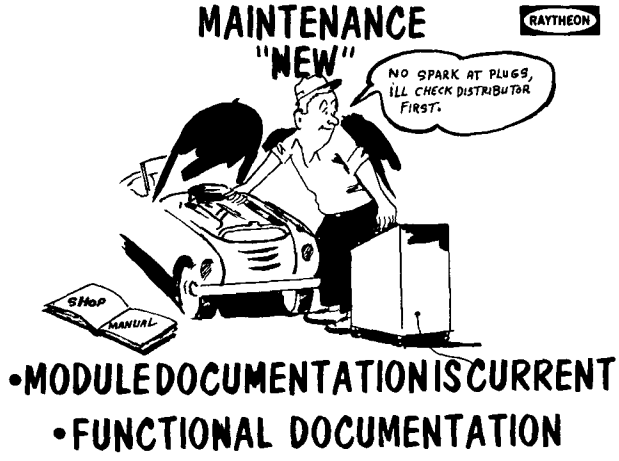

Figure 23