Modularization: A Key Component of Structured Programming

by Rod Fulton

I have been programming for eleven years. The last ten of those have been in M and have involved mostly maintenance programming. As a result, I have become very familiar with unstructured spaghetti code. I've even written some of it myself. I did this for several reasons:

• The block structured DO was not available and consequently GOTO's were required to simulate it.

• There was no simple, straightforward way to scope variables. The NEW command and parameter passing took care of this.

• Efficiency constraints forced (or so I thought) the conservation of bytes by putting as many commands on a line as possible and minimizing the length of variable names and labels.

• I thought that comments were useless since they didn't have to be accurate. The interpreter ignored them, and producing them took up valuable programming time. I also believed that only wimps needed comments to understand code. Of course, I was wrong. Today, they are an integral part of my design and coding processes. If I can't describe an M process in English, then I am usually not clear about it, and I need to go back to the drawing board.

Why I Write Only Structured Code

Five years ago, in 1991, I realized the error of my ways and became a fanatic about writing structured code that is easily readable and maintainable. This happened when what an old timer had warned me about as I was just starting out as a programmer finally happened. I needed to modify a program I had written but hadn't touched in months, and I couldn't make any sense of it. *I couldn't read my own code!*

Since that day, I've changed the way I do things. I approach computer systems with the attitude that the most expensive component is the programmer and that everything possible should be done to maximize his use. This means using only structured programming techniques so that others can read, maintain, and modify programs as easily as possible.

Modularization

In producing structured code, one of the key components is modularization. To do this, you must be on the lookout for things you have done before or that you will do again. I'm not talking about single commands but sets of commands. Sometimes, but not often, these things are done exactly in the same way and can easily be made into a separate routine or function. Usually, however, they aren't exactly alike but are so close that if you think about them a little (or a lot), you can figure out how to abstract them to a level that will work in all instances.

These sets of commands or actions fall into two categories. There are those that are specific to a particular product, operating system, or company, and there are those that are universal and can be used by most M programmers on any system. While there are more of the first category on any given system, I will concentrate on the second category because it will allow me to present examples that are germane most anywhere. If you develop a frame of mind that is always looking for patterns, you will start seeing them everywhere and can start modularizing your system.

An Example

The Manual Method

Consider the situation where you need to set several variables to values of pieces of another variable. Usually this is a global variable but it doesn't have to be. It almost always has more than one level of delimiters, and these delimiters and their priorities are standard throughout the system. Let's assume that the program we are working in will be fed the variable *person* which, if referring to me, could have been created by the following line of code:

```
s person="Fulton;Rod^713 Heavens
Dr,Apartment 3; Mandeville;LA;70471^(604)
. . . 845-1892;(800)759-8074;(818)356-
0479"
```

The delimiters, in order of precedence, are: "^;,". Assume that we want all of the pieces of information placed in indi-

vidual variables so we can do whatever with them. The following is the way most people would do it:

```
s last=$p($p(person),"^"),";")
s first=$p($p(person),"^",",2)
s street1=$p($p($p(person,"^",2),";"),",")
s street2=$p($p(person,"^",2),";"),",",2)
s city=$p($p(person,"^",2),";",2)
s state=$p($p(person,"^",";",3)
s zip=$p($p(person,"^",2),";",4)
s country=$p($p(person,"^",2),";",5)
s phone1=$p($p(person,"^",3),";")
s phone2=$p($p(person,"^",3),";",2)
s phone3=$p($p(person,"^",3),";",3)
```

Anyone looking at this chunk of code would have a little bit of work to do to make sure that everything was coded correctly. There are eleven lines of code that have to be deciphered—one for each SET command. You need to make sure each source and target variable is spelled correctly. In addition, there are twenty-two $PIECE functions. You need to make sure that they each reference the proper piece. That's a lot of work. What do you do if these same variables are set the same way in other routines?

The First Pass at Abstraction and Modularization

The next step is to create a separate routine that does nothing but set these particular target variables. If *person* is not always the name of the source variable, you could parameterize it so that it can be called from anywhere.

We have actually come up with a good solution for unloading this particular source string. What about other source strings? You could write a similar routine for each one. But first, remember that I said that the source variable is actually a global variable. In the example I gave above, I didn't say so but if you look at what it contains, it's pretty clear that *person* was probably set to the value of all or part of a global. Now, stop and think about all of the individual global nodes in the system you work on. It's very likely that you have hundreds, if not thousands of them. Do you want to write individual routines for unloading each of them into standard variables? I doubt it.

Complete Abstraction and Modularization

What if we could write a routine that would work with any target string? We would have to tell it the source variable and the target variables. We would also have to figure out a way to tell it which pieces of the source string belonged to which target variables. I'll show you how it works later but the following will work:

```
d
^setvars("last;first^street1,street2;city;
state;zip;country^phone1;phone2;phone3^",
person)
```

The first parameter is a map of what variables we want and where they are located. The second parameter is the source variable. This certainly makes it much easier to read and write in-line code than the original manual method. This is similar in functionality to an extension to the language offered by InterSystems' DataTree product but it differs quite a bit in that it can deal with sub-delimiters and that it is portable to any ANSI M system. Notice that I've unloaded all of the variables. I could write the routine so that it only sets the variables I specify, which is not an uncommon thing to do. Suppose that we only want the name and phone numbers. The following is an example of how to use it to do this:

```
d^setvars("last;first^^phone1;phone2;
phone3^",person)
```

I mentioned the delimiters that are used in this system. I have written the routine to automatically use them. But what about those cases where different delimiters are used? Wouldn't it be nice to be able to override them? This can be done by allowing for an optional third parameter which would contain the delimiters, in order of precedence, to be used. If no third argument is supplied, the routine will automatically use the default delimiters.

The following routine (see next page) walks through the variable string, delimited piece by delimited piece. When it finds a non-null piece it uses that as a variable name and sets it to the corresponding piece of the source variable string. Null pieces of the variable string are ignored.

If the list of target variables includes "var," vars," "dat," "data," or delims, a forced error occurs. If these are variable names used in your system you can change them in ^setvars. You can also specify whatever default delimiters you want.

Benefits of Modularization

- **More readable programs.** Programs that use modularized code, whether in the form of programs, as shown here, or in the form of extrinsic functions, are easier to read. For one thing, they simply contain less code. For another, it is easier to see the big picture in what such programs do because the nitty-gritty details are not cluttering up the code.

- **Smaller programs.** As we write new programs, they will be smaller and quicker to produce and contain fewer logical and syntax errors.

- **Few errors.** Because you are writing less code, you are going to make fewer mistakes.

Source code for ^setvars

```
setvars(vars,data,delims) ;RLF;02:10 PM 22 Sep 1996
 ;
 ;This routine sets the variables in 'vars' to the corresponding values in 'data'. It runs off
of what is in 'vars'. Null pieces in 'vars' are skipped even if their counterparts in 'data' h a v e
data. This allows you      ;to retrieve only the variables you want.
Variables
 ;defined in 'vars' are set to null if their counterparts in 'data' are null.
 ;
 ;The variables 'vars' and 'data' are the strings to be processed.
 ;
 ;The variables var' and 'dat' are their respective heads as delimited by the first character
of 'delims'.
 ;
 ;The variable 'vars' can't contain the following strings: 'vars', 'var',       ;'data', 'data',
 or 'delims'. If you think about it you can see that this will cause nothing but trouble. Since
this routine is not interactive the only thing to do is have it self-destruct when this
;occurs.
 ;
 n var,dat                                    ; Protect these variables
 ;
 s:'$d(delims) delims="^;,"                   ; default delimiters
 ;
 s var=$p(vars,$e(delims))                    ; set the heads
 s dat=$p(data,$e(delims))                    ;
 ;
 s vars=$p(vars,$e(delims),2,511)             ; set the tails
 s data=$p(data,$e(delims),2,511)             ;
 ;                                            ;
 i var="vars"!(var="var") thendie             ; Force an error if var is pro
                                              ;hibited variable name
 i var="data"!(var="dat") thendie
 i var="delims" thendie
 ;
 i var=$tr(var,delims) s:var]"" Ovar=dat      ; Base case
 e d setvars(var,dat,$e(delims,2,255))              ; Recur
 d:vars]"" setvars(vars,data,delims)          ;Next piece
 q
```

• **Less code to manage.** By modularizing whenever possible, you will decrease the amount of code on your system. The number of programs will increase, but they will be smaller, simpler, easier to understand, and easier to modify.

• **Modifications are easier.** This is because you've isolated functionality. When you want to change something, you only have to do it in one place. This is not really clear from the example I gave because once you implement it on your system, you won't need to change it. However, most systems, including yours, have activities that are done the same way by similar code in several different routines. Modularization lets you identify, isolate, and standardize them.

Drawbacks

The only possible drawback may be efficiency but you shouldn't be concerned with it. The thing to do is use modularization with other structured programming techniques to produce an understandable and modifiable system. And sooner or later, every system will need to be modified. Usually sooner. And usually often. If you concentrate on efficiency, you will end up in over you head very quickly. You will have to tell users that you can't provide changes they want. You may not tell them the truth about why you can't make the changes. You may not tell yourself either. You will be afraid of the side effects because you won't be sure of all the things you will need to change. Even when you will be able to make desired changes they will take orders of magnitude longer than necessary. The company will end up spending more money on your low productivity than will be offset by purchasing faster computers. Besides, many efficiency techniques are dependent upon hardware platforms and the implementations and versions of M and the operating system.

If you do end up with a system that is unacceptably slow, you will probably have to break the rules by tweaking code for efficiency in only one or two places. In the example I have given here, ^setvars is thirteen to twenty times slower. But so what! If you implement it everywhere and your response time is too slow, you were either on the verge of overloading your system

anyway, or, more likely, you have one or two batch programs that call ^setvars thousands or hundreds of thousands of times. These programs could be moved to off times if possible. If not, then they could be tweaked by using the manual method of setting variables that ^setvars replaces. In such cases, document the source code to explain why you have done this.

Conclusion

Modularization is an important part of structured programming. It helps you create understandable and maintainable systems. I have given you an example of modularization that will work on your system. I have walked you through the process of creating it. If you put yourself in the frame of mind where you hate doing things more than once, you will start seeing possibilities for modularization in your own system. If you start taking advantage of them, in a few months you will find that your productivity has increased. M

Rod Fulton is a consultant specializing in M. He can be reached at: fulton@cs.tulane.edu or at 1-800-759-8074.
