MAINTAINABLE CODE II: Clarity

Lincoln Stoller, Ph.D

4/9/93

Overview

This article analyzes the problem of writing clear 4th Dimension code, and discusses how in-line documentation, layout and logical flow lead to clearer code. It is the second of four articles on writing maintainable code.

Computer code is a sparse narrative that requires documentation to make its intention clear. As in any narrative the golden rule is that the author must make a commitment to the reader. Decide who you are going to write for -- whether it be for yourself six months from now or for someone completely unfamiliar with your code -- then continue to write for that reader. Every line of code must explain itself to your ever-present, always-interested imaginary reader.

Writing clear code, like maintaining your personal health, requires an appreciation of its long term value. In many cases it takes a code management crisis to develop such a sense of value. Incidentally, writing for yourself does not excuse you from communicating clearly. Just because you are the author of your own code does not mean you'll understand it in the future, as anyone who's had to decipher their own notes can attest!

Most of the examples in this article relate to a single code segment. The first version of this code, shown in Figure 1, represents what's produced by an author with little interest in clarity or communication. In the course of this article I'll use some of the naming conventions discussed in Part One of this series.

```
`Procedure pGL_Do_H
$0:=0
If (fGL_CK_F_OS($1) = 55)
    Repeat
        $Var1 := fAC_10(4)
        $Var2:= fDo_F_T($Var1)
        pDo_L_MT(1;$Var1)
        Case of
        :(($Var2 = 1) | ($Var2 = 2))
            pAC_12(54)
        :($Var2 = 3)
            Repeat
                pAC_5(»gF_Val01)
            Until (gF_Val01 <= 0)
            pAC_12(54)
            gF_Val12 := True
        Else
            $0 := -1
        End case
    Until (gF_Val12 | ($0 = -1))
Else
    If (gF_Val12)
        pDo_L_MT(2;0)
    Else
        pDo_L_MT(3;0)
    End if
End if
```

Figure 1. Raw, unfathomable code as formatted by 4D's procedure editor.

Strategies for documenting code

In-line comments

A well-documented procedure begins with a header consisting of a number of nonexecuting lines giving its name, purpose, date of creation and last modification, a list of passed parameters and a description of each. If necessary, the header can contain a list of modifications describing when they were done, what was done and why. It can contain a list of global variables used in the procedure, specifications of which variables can be modified and which are invariants, prerequisites the procedure assumes are satisfied before it executes, the conditions that obtain after the procedure is finished and possibly the names of the calling procedures or scripts.

Avoid lengthy, detailed header comments that your reader might find tedious or obscure. The comments themselves can become a maintenance nightmare if they need to be rewritten each time a change is made. To guard against reiterating the code in words, my policy is only to use a sentence in the header for a description of what I intend to be the immutable purpose of the procedure. I place technical descriptions or those relating to the mechanics of the code adjacent to the lines where the actions are performed. Although this scatters the documentation throughout the procedure, it enables me to change the code and update the comments with a minimum of effort.

If the code is being used by people other than the code's author, it is useful to limit the header to descriptions of the purpose, context, prerequisites, side effects and error conditions. Limit in-line commentary to technical explanations about how the code achieves these effects.

Self-documenting code

Several years ago I asked a programmer at a large financial institution how the dozen programmers employed there were able to stay on top of their multi-thousand line securities management program. "We write self-documenting code," was the brusque reply. Since that time I have concluded that for real-world problems, self-documenting code does not exist and most likely never will. The concept is almost inherently contradictory since people require documentation that removes the details and emphasizes major ideas while computers require the mechanical connection of every detail. In spite of this we can use the latitude available to us to organize our code to reflect its logical structure.

Simple layout issues and the value of blank space

There are basic guidelines for clear coding that have more to do with visual composition than with logic or verbal explanation. For example, the way we arrange our lines of code can help the reader's eye catch the major ideas on a visual level. Contrary to some samples I've seen, blank space is not beneficial in and of itself; it's only the contrast of dense code with blank lines that has meaning.

As a rule I leave a blank line before every If, Else, Case of statement, procedure call or block of consecutive calls. I leave two blanks lines between loosely connected

sections of code. Figure 2 shows how a few blank lines can make the structure code more evident.

```
`Procedure pGL_Do_H
$0:=0

If (fGL_CK_F_OS($1)=55)
    Repeat
        $Var1:=fAC_10(4)
        $Var2:=fDo_F_T($Var1)
        pDo_L_MT(1;$Var1)

        Case of
        :(($Var2=1) | ($Var2=2))
            pAC_12(54)

        :($Var2=3)
            Repeat
                pAC_5(»gF_Val01)
            Until (gF_Val01 <= 0)
            pAC_12(54)
            gF_Val12:=True

        Else
            $0:=-1
        End case
    Until (gF_Val12 | ($0=-1))

Else
    If (gF_Val12)
        pDo_L_MT(2;0)

    Else
        pDo_L_MT(3;0)
    End if
End if
```

Figure 2. Well-spaced code.

Logical flow versus sequential execution

A logical pattern is a chain of cause and effect. To display the logic of your algorithm you should do your best to place the code that embodies the cause adjacent to code that handles the effect. This means minimizing the number of lines and the complexity of

the code that stands between where an action is indicated and where it is performed. This may mean adding comments that restate the actions that are the cause of the code that's currently being handled, or it may mean shuttling causes and effects into separate procedures so that the details are hidden and only the larger scheme is expressed.

Unfortunately, sequential coding does not allow all causes immediately to precede their effects. Instead, text-based environments, including 4D's text editor, force you to string out logical 'trees' into a linear sequence. Branching statements, like the Case of and If-Else statements that present simultaneous possibilities, are forced to display an order of precedence. This works contrary to the self-documenting ideal and is illustrated in Figure 3.

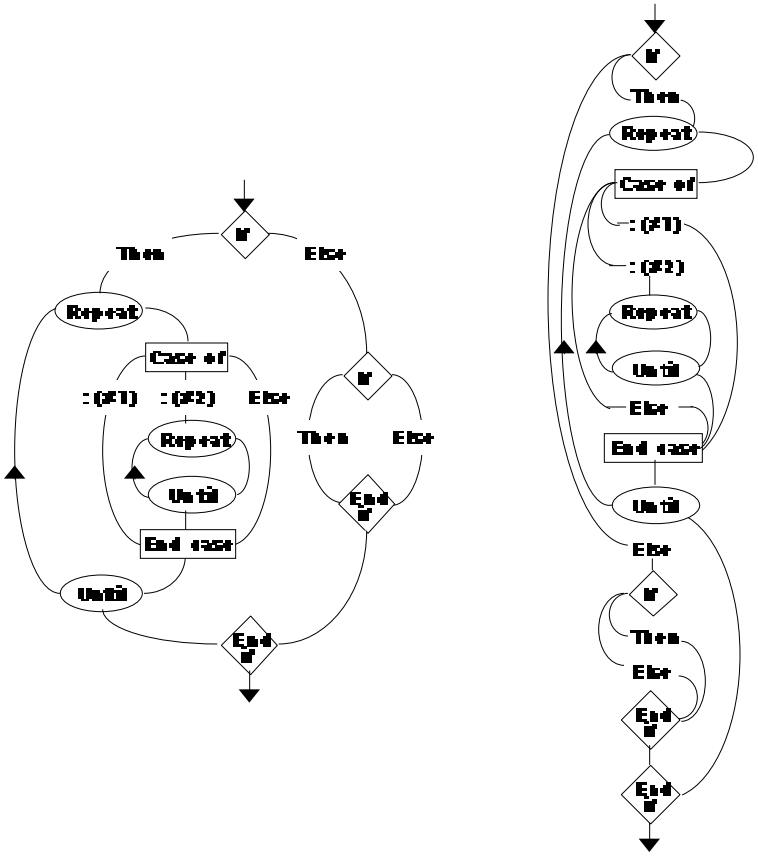

Figure 3. Displaying the logical flow of the code in Figure 1 in a linear sequence.

Sequential coding is a necessity but this shouldn't stop us from recognizing how it affects code maintenance. When complex nesting and branching turns our code into visual spaghetti, the sequential model becomes the greatest factor in obscuring our

intent. Once we recognize the problem we can better determine what it will take to untangle it.

A simple way to clarify branching and multiply-nested If statements is to restate the conditions at the end of the structure. This is especially important in situations where you need to remain informed of the order in which the Else and End If clauses are interleafed.

The more powerful way to clarify code logic is to create procedures that hide supporting code and encapsulate branching structures. For example, if you have an If-Else statement with two large sections of code, you can condense this to four lines by putting the If-code and the Else- code into separate procedures. The resulting structure is manifest in a single glance, as you can see by comparing Figure 3 with its decomposition in Figure 4.

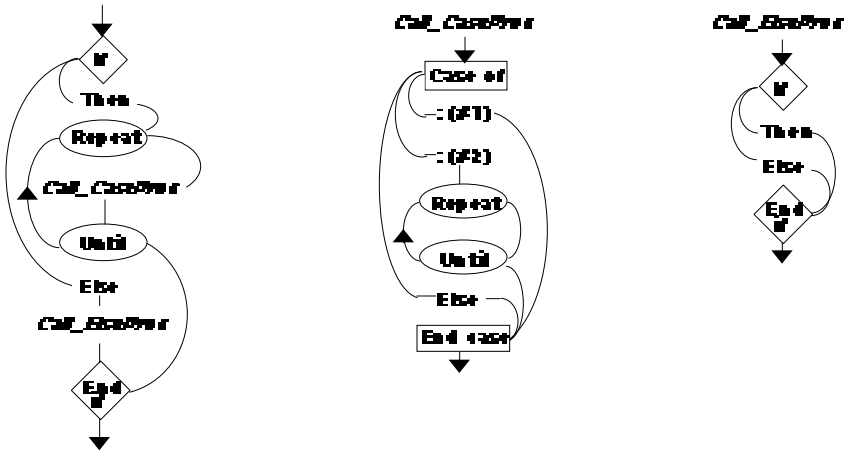

Figure 4. Improving the ‘logical map’ by adding subordinate procedures.

However, subdividing code can also lead to poor memory usage and slower execution for the following reasons:

- nested procedure calls use up limited amounts of RAM and stack space;
- procedures inside a loop add significant memory management overhead and degrade performance;
- tightly related procedures will likely share many variables and will require information to be passed either explicitly through parameter lists or implicitly by using additional global variables.

In addition to performance issues, adding procedures fragments the big picture making it harder to integrate the details. The problem is that our logic operates on various scales: sometimes we need focus, sometimes we need generality. It may seem that we can't ever "have it both ways," but the fact remains that we think both ways.

My guideline is to subdivide code into conceptually bite-sized chunks. Each procedure has a title that describes its function. I condense this title into a few words and squeeze these into a 16 character name. Before I code the procedure, I write a line or two to document what the procedure does. If I need to handle tasks outside the scope of this description, I consider splitting these tasks off into separate procedures.

I try to limit my procedures to one dominant control structure, such as a series of nested If-Else statements to test for a series of related conditions, or a Case of structure to execute one of several parallel conditions. However I may combine several control structures within a single procedure in cases where there is a lot of widely shared information. In these cases it is often better to use a single, involved procedure than it is to use several procedures that are so interrelated that you need a significant amount of code just to pass information from one to the other. In these cases I preceed each control structure with a summary of its purpose and how it relates to the procedure as a whole.

Modularize

Modular software is divided into more or less independent parts that communicate with each other in a controlled manner. Dividing your program into modules means more than just chopping it into segments, as I'll discuss at length in the third article of this series. For the present it's sufficient to say that you should divide your program into functional areas and identify the area to which each procedure and variable belongs. Any grouping of programming components is a step in the right direction. The objective is to divide and conquer, to cut the program up into pieces so that you can better rememeber and integrate the inevitable profusion of parts that appears when you create a complex program.

Employ a naming convention to distinguish procedures and variables of one module from another. This is an additional label -- most likely a prefix -- that identifies a

component's general function in addition to the more precise identification given by its unique name. Module tags should also be applied to the naming of global variables, as I'll describe in the next section.

As an example, consider a personal banking program that's divided into a transactions module and a services module. Procedures and variables belonging to the former could be prefixed with the characters TR, and those belonging to the latter with the characters SR. Additionally, you'll probably have general use variables and procedures that should be placed in a system's module. Here is a list of possible procedure names and their functions:

TRAddDpstWdrw	Add a deposit or withdrawal transaction
TRCekBalance	Check the account balance an alert if too low
TRPrtRunBal	Assemble and print a running balance report
SRFndCost	Find the cost of a given bank service
SRChgMonthFee	Subtract the monthly service charge from the balance
SRCekFundAvail	Check amount of funds available for withdrawal
SYAsgnID	Assign a unique ID for a new record in some file
SYStripBlanks	Remove leading and trailing blank spaces from text
SYInitGenVars	Initialize value of general system variables

Naming procedures in this manner will segregate them in the sorted procedure list, allowing you to avoid scanning a long list of names or tracing through nests of procedure calls. The appearance of a different prefix will also alert you to the performance of an action that's not part of the current module. Finally, modularizing will make documenting your program much easier since you can describe your program module by module.

Putting it into practice

Many readers are probably inclined to find programming conventions boring, unnecessary or both. We often feel that convention, like etiquette or correct spelling, is something that we want to be free from in the privacy of our own code. Proper spelling is a good analogy because poorly worded, choppy, misspelled text can be impossible to read (have you ever tried to read someone else's notes?) Deciphering misspelled words often requires a context, and context is just what code often lacks.

Consequently, meaningful and consistent syntax is extremely important. Consider the results shown in Figure 12 of these documentation and composition conventions, along with the naming conventions discussed previously, compared to the same code shown in Figure 1.

```
`Function: InBearsHouse
`Called to manage Goldilock's actions after she enters the house of the three bears.
`$1=string giving Goldilock's location;
`$0=error code, 0 if no error has occurred.
`Created: 11/10/92
`Modified: 11/29/92

c_Longint  ($StandAtTabl;$TooHot;$TooCold;$JustRight;$Anywhere;$FreeOfErrs)
c_Longint  ($TasteResult;$SeatChosen;$OatsNotHere;$ErrorCode;$0)
c_String  (30;$1;$Location)
$Location:=$1  `Give names to the numerical and boolean codes.
$HasErrors := False
$StandAtTabl := 55
$TooHot := 1
$TooCold := 2
$JustRight := 3
$OatsNotHere := -1
$Anywhere := 0
$ErrorCode := 0

If (Goldilocks($Location) = $StandAtTabl)
    Repeat      `Keep choosing seats and tasting until the oatmeal is eaten.
        $SeatChosen := PickSeat("AtTable")
        MoveToLocation("SitDown";$SeatChosen)
        $TasteResult := TasteOatmeal ($SeatChosen)

        Case of      `Check oatmeal to find the one that's just right.
        :(($TasteResult = $TooHot) | ($TasteResult = $TooCold))
            StandUp("AtTable")    `No good, try another seat.

        :($TasteResult = "JustRight")  `Eat it all.
            Repeat
                EatOatmeal(»lWhatsLeft)
            Until  (lWhatsLeft<= 0)
            bHasEaten:= True
            StandUp("AtTable")

        Else      `This is a programming error; exit the procedure.
            $ErrorCode := $OatsNotHere
            $HasErrors := True
```

```
            End case
        Until  (bHasEaten | $HasErrors)

    Else
        If (bHasEaten)    `Then she's already entered the house before.
            MoveToLocation ("Upstairs"; $Anywhere)

        Else      `She hasn't eaten and she's not at the table.
            MoveToLocation ("StandingAtTable"; $Anywhere)
        End if    `(bHasEaten)
    End if    `(Goldilocks($Location) = $StandAtTabl)
    $0 := $ErrorCode
```

Figure 12: The original code of Figure 1 rewritten using naming, documentation and composition conventions.

Conventions help our code convey its underlying concepts. They save us from those "what the hell did I do here?" situations. The basic object is to write code that helps to clarify our thinking.

Acknowledgments

Thanks to Scott Ribe, Tracy Harms and Michael Billesback. Thanks also to Jennifer Fox for editorial direction.