Software Quality and Coupling and Cohesion

12/12/1993
By Vince Kellen
vjk@kellen.net
http://www.kellen.net

We all know how to write clean, elegant, efficient code, right? After all, if we didn't we wouldn't be the great programmers we think we are. Oh yes, you'd also gladly let others review your code, rip it apart and endure their back-seat driving as they second-guess all the hundreds of small but important decisions you made in your code. Of course.

The previous passage was written tongue-in-cheek. None of us really enjoys getting code mercilessly critiqued, and most of us believe that we really are competent PAL programmers. Or at least they we are improving our skills. So, naturally we avoid being closely reviewed.

If you're a serious programmer, you should gleefully welcome such a review. Why? Because over the years the software industry has accumulated bits of wisdom that give us clues how to write solid, maintainable code. Taking a hard, critical look at your own code and applying these pearls of wisdom can help make order out of chaos. Two such nuggets are the concepts of coupling and cohesion.

Help Wanted

Coupling and cohesion were discussed at great length by Larry Constantine and Edward Yourdon in their 1976 landmark book, *Structured Design*. Since these concepts have been tied to structured programming, in this object-oriented day it seems that many programmers are forgetting what they mean. In many ways, the concepts are still extremely important.

We will tackle cohesion first, since it is a bit easier to comprehend. Cohesion refers to how clearly-defined a particular module or procedure is. A module with high cohesion does one or a few things exceedingly well. Let's use a job description as an example. Suppose you placed the following ad:

> Wanted: Programmer with solid skills in the following areas: System analysis, design, testing methodologies, EDI, software auditing, COBOL, BASIC, PAL, SmallTalk and C++ required. Strong real-time data acquisition skills, scientific applications experience and accounting knowledge a must. Candidates must be experienced, energetic, willing to work long hours, be able to lead groups of programmers, get along with others and have the patience of Job.

How many qualified candidates would you get? And if you found one how long would he or she last? How many skills can one person excel in at one time? If I had that job, after a month or so I'd probably be wondering what my name is.

If a particular module is responsible for a diverse set of tasks, chances are the module will be, like our hapless candidate, a mess. So what constitutes a module which exhibits poor cohesion? I offer code listing 1 for your consideration.

Code Listing 1. An example of poor cohesion

```
; ------------------------------------------------------------------------
; PostPayroll.l() - Post The Payroll Data
;    Input:   Void
;    Return:  TRUE  - Payroll Posted
;             FALSE - Posting Cancelled
; ------------------------------------------------------------------------
Proc CLOSED PostPayroll.l()
```

```
UseVars AutoLib,
        DefaultPayPeriod.n,
        DefaultDiff.n,
        DefaultAcctNuma,
        MonitorType.a,
        ApplicationRev.a,
        DeveloperMode.l,
        MinWage.n,
        PWord.a

Private ProcName.a,
        FieldDate.y,
        DOW.y,
        Week.y,
        Ok.l,
        Title.a,
        Prompt.a,
        Period,
        S.d,
        E.d,
        W1,
        W2,
        W3,
        W4,
        D,
        X,
        Y,
        Ret.l,
        WeekDate.r

ProcName.a = "PostPayroll.l"
Ret.l = FALSE
MaxPer.n = CMax("PayPer", "Period #") - 2


SHOWDIALOG "Post Payroll Data"
   @7,15 HEIGHT 14 WIDTH 51

   ; PaintPAL_Frame_Begin
   FRAME SINGLE FROM 1,1 TO 7,47
   PAINTCANVAS BORDER ATTRIBUTE 65 1,1,7,47
   ; PaintPAL_Frame_End

   ; PaintPAL_Static_Text_Begin
   PAINTCANVAS FILL " " ATTRIBUTE 79 2,2,6,46
   @2,2
   ?? "Payroll Posting should not be done until ALL"
   @3,2
   ?? "of the Data has been entered.  It needs to be"
   @4,2
   ?? "done prior to generating the Payroll Report"
   @5,2
   ?? "or Exporting Data to Cougar."
   @6,2
   ?? ""
   PAINTCANVAS ATTRIBUTE 79 2,2,6,46
   ; PaintPAL_Static_Text_End

   PUSHBUTTON @9,1 WIDTH 17
      "~O~k - Post Data"
      OK
```

```
            DEFAULT
            VALUE TRUE
            TAG ""
            TO Ok.l

        PUSHBUTTON @9,23 WIDTH 18
            "~Cancel Posting"
            CANCEL
            VALUE FALSE
            TAG ""
            TO Ok.l
    ENDDIALOG


    If RetVal Then
        ClearAll

        Title.a = "Weekly Payroll Processing"
        Prompt.a = "Enter Pay Period:"

        Period = GetPayPeriod.n(Title.a, Prompt.a, DefaultPayPeriod.n, 1,
MaxPer.n)
        If Period = 0 Then
            Return FALSE
        EndIf

        ChangePayPeriod.u(Period, FALSE)

        Message "PROCESSING PAYHIST FOR PERIOD " + StrVal(Period) + " ....
PLEASE WAIT."

        DynArray FieldDate.y[]
        DynArray DOW.y[]
        DynArray Week.y[]
        Array WeekDate.r[4]

        WeekDate.r[1] = BlankDate()
        WeekDate.r[2] = BlankDate()
        WeekDate.r[3] = BlankDate()
        WeekDate.r[4] = BlankDate()

        IF NOT DeleteBadRecords.l() Then
            Beep
            Message "WARNING: Exclusive Access to TIME is required"
            Sleep 5000
            Return
        EndIf

        ClearQueries.u()
        Menu {Ask} {Payhist}
        "FAST DELETE"
        MoveTo [Period #]
        TypeIn StrVal(Period)
        Do_It!

        ClearQueries.u()
        Message "PROCESSING PAYPER .....  PLEASE WAIT."


        Query
```

```
        Payper |    PERIOD #    | START DATE | END DATE |
               | Check ~PERIOD | Check      | Check    |

       EndQuery

       Do_It!

       S.d = [ANSWER->START DATE]
       E.d = [ANSWER->END DATE]

       Message "PROCESSING TIME FOR PERIOD " + StrVal(Period) + " ....  PLEASE
WAIT."

       Query

        Time |    PERIOD #    | EMPLOYEE # | START DATE | END DATE |
             | Check ~PERIOD | Check      | Check      | Check    |

        Time | NDAYS |
             | Check |

        Time |  TIME 1            |  TIME 2            |
             | CALC SUM AS TIME 1 | CALC SUM AS TIME 2 |

       Endquery

       Do_It!

       Rename "Answer" "PaySums"
       ClearQueries.u()

       Query

        Time |    Period #    | Employee # |
             | Check ~PERIOD | Check      |

        Time | Actual Pay Rate | Time Total                              |
             | Check _R        | Check _H, calc _H * _R As Base Pay |

       Endquery

       Do_It!

       Rename "Answer" "PayHours"

       ClearQueries.u()

       Message "Calculating Composite Pay for Period " + StrVal(Period) + "
....  PLEASE WAIT."

       Query

        Payhours | PERIOD # | EMPLOYEE # |        TIME TOTAL        |
                 | Check    | Check      | Calc Sum As Total Hours |

        Payhours |         ACTUAL PAY RATE          |        Base Pay       |
                 | Calc Sum As Composite Pay Rate | Calc Sum As Base Pay |

       Endquery

       Do_It!
```

```
Rename "Answer" "CompPay"

ClearAll

Query

 Comppay | Total Hours | Composite Pay Rate | Base Pay |
         | _h          | changeto _p / _h   | _p       |

Endquery
Do_It!

ClearAll

DeleteTable.u("PayHours")                    ; Delete temporary table

If NOT ComputePayHours.l() Then
   Beep
   Message "ERROR: Could NOT Compute Payroll Hours - Cancelling Posting"
   Sleep 5000
   Return FALSE
EndIf
ClearAll

; Delete any Existing PAY records for this period.

Message "CLEARING EXPORT FILES ......  PLEASE WAIT."

If NOT EmptyTable.l("Pay") Then
   Beep
   Message "ERROR: PAY Table Does NOT Exist - Cancelling Posting"
   Sleep 5000
   Return
EndIf

Menu {Ask} {Pay}

; Create PAY records from corresponding PAYHIST records for this period.

MoveTo [Pay(Q) -> Period #]
Del
CtrlHome
"FAST INSERT"
MoveTo [Period #]
Example "PERNO"
MoveTo [Employee #]
Example "EMPNO"
MoveTo [T1]
Example "T1"
MoveTo [T2]
Example "T2"

Menu {Ask} {PayHist}
CtrlHome
MoveTo [Period #]
Example "PERNO, ~PERIOD"
MoveTo [Employee #]
Example "EMPNO"
MoveTo [T1]
Example "T1"
```

```
MoveTo [T2]
Example "T2"

Do_It!

ClearAll
Query

 Payhist | Employee # | Period # |
         | _e         | _n       |

 Payhist | Composite Pay Rate |
         | changeto _p        |

 Comppay | Period # | Employee # | Composite Pay Rate |
         | _n       | _e         | _p                 |

Endquery
Do_It!
ClearAll

DeleteTable.u("CompPay")  ;  Delete temporary table
Message "PREPARING PAYROLL FOR EXPORT .....  PLEASE WAIT"

If LockStatus("Pay", "ANY") = 0
AND LockStatus("Emp2", "ANY") = 0
AND LockStatus("PayHist", "ANY") = 0 Then
   Lock "PayHist" FL, "Emp2" FL, "Pay" FL
   If RetVal Then
      Ret.l = TRUE
      View "PAYHIST"
      View "EMP2"
      Edit "PAY"
      Home
      Scan For [Period #] = PERIOD
         CopyToArray PayData.r
         Message "Computing Pay for Employee #: " + Format("w10, al",
PayData.r["Employee #"])

         EmployeePayRate.n = 4.25

         MoveTo "EMP2"
         MoveTo [EMPLOYEE NUMBER]
         Locate PayData.r["Employee #"]
         If NOT RetVal Then ; No Matching Employee Master - Skip this
One
            EmployeePayRate.n = 0
            MoveTo "Pay"
            Loop
         EndIf

         MoveTo "PayHist"
         MoveTo [Period #]
         Locate PERIOD
         If NOT RetVal Then ; No Matching Period Data - Cancel
Processing
            Ret.l = FALSE
            QuitLoop
         EndIf
         Locate PayData.r["Employee #"], PERIOD
```

```
                If NOT RetVal Then ; No Matching Employee Pay History - Skip
this One
                   MoveTo "Pay"
                   EmployeePayRate.n = 0
                   Loop
                Else
                   EmployeePayRate.n = [Composite Pay Rate]
                   MoveTo "Pay"
                EndIf

                [t1$] = EmployeePayRate.n * [T1]
                [t2$] = EmployeePayRate.n * [T2]

                [TOTAL T] = [T1] + [T2]
             EndScan
             Do_It!
             UnLock "PayHist" FL, "Emp2" FL, "Pay" FL
          Else
             Ret.l = FALSE
          EndIf
       Else
          Beep
          Message "ERROR: Exclusive Access to PAYHIST, EMP2, and PAY required"
          Sleep 2000
          Ret.l = FALSE
       EndIf
    EndIf

    Return Ret.l
EndProc  ;  PostPayroll.l
```

As you can see, the first sign, *usually*, that a module exhibits poor cohesion is its length. The longer a programmer rambles on in a module, the drearier and less comprehensible the module becomes. The module lacks zing, punch, pizzazz, which of course, is a good thing for keeping those who maintain the code awake. The module above is performing too many different things: getting input from the user (on whether to continue or not, messaging the user on error conditions, performing several queries and setting up the query images and locking tables. Each of these tasks can be delegated down in the hierarchy to a specialized module. This would improve the module's cohesiveness.

Now of course, there are exceptions to the rule. It is possible for a lengthy module to still have a high degree of cohesion, *provided the module excels in one or two things only.* For example, some modules contain code which executes a series of commands or a series of evaluations. The latter case is especially common in PAL, particularly when the module needs to evaluate a list of menu choices or a list of events. Code listing 2 shows such a module.

Code Listing 2. A Long Switch Case module

```
; ------------------------------------------------------------------------
; This proc processes the user's menu selection
; ------------------------------------------------------------------------
proc PROCESS_MEMO_MENU(CHOICE)
    private RV
    RV = true

    switch
       case CHOICE = "M-Read" :
          MEMO_READ()
       case CHOICE = "M-Write" :
          MEMO_WRITE()
       case CHOICE = "M-Block" :
          MEMO_BLOCK()
```

```
        case CHOICE = "M-Save" :
            SHOWMSG("Saving memo...",false)
            Menu {File} {Save}
            SHOWMSG("Memo saved",false)
        case CHOICE = "M-Print" :
            SHOWMSG("Printing memo...",false)
            Menu {File} {Print}
            print "\f"
            CLEARMSG()
        case CHOICE = "M-Copy" :
            CLIP_COPY()
        case CHOICE = "M-Append" :
            CLIP_APPEND()
        case CHOICE = "M-Paste" :
            CLIP_PASTE()
        case CHOICE = "M-Goto" :
            MEMO_GOTO()
        case CHOICE = "M-Show" :
            MEMO_SHOW_CLIP()
        case CHOICE = "M-Search" :
            MEMO_SEARCH()
        case CHOICE = "M-Next" :
            MEMO_NEXT()
        case CHOICE = "M-Replace" :
            MEMO_REPLACE()
        case CHOICE = "M-IndentOn" :
            menu {Options} {AutoIndent} {Set}
            SHOWMSG("AutoIndent is set",true)
        case CHOICE = "M-IndentOff" :
            menu {Options} {AutoIndent} {Clear}
            SHOWMSG("AutoIndent is now off",true)
        case CHOICE = "M-WrapOn" :
            menu {Options} {WordWrap} {Set}
            SHOWMSG("WordWrap is now set",true)
        case CHOICE = "M-WrapOff" :
            menu {Options} {WordWrap} {Clear}
            SHOWMSG("WordWrap is now off",true)
        case CHOICE = "M-CaseOn" :
            menu {Options} {CaseSensitive} {Set}
            SHOWMSG("CaseSensitivity is now set",true)
        case CHOICE = "M-CaseOff" :
            menu {Options} {CaseSensitive} {Clear}
            SHOWMSG("CaseSensitivity is now off",true)
        case CHOICE = "M-CancelYes" :
            menu {Cancel} {Yes}
            RV = "Cancel"
        case CHOICE = "M-Exit" :
            do_it!
            RV = "Exit"
        otherwise :

    endswitch

    echo normal
    return RV

endproc
```

In this example, a long series of case statements evaluate the user's menu choice. Since the module does exactly one thing, *evaluate a menu choice*, it qualifies as a highly cohesive module.

Now, we can shrink this module down somewhat by using a dynarray to process the menu choices, as shown in code listing 3. While this makes examining the part of the module which actually does the work (the switch-case statement) easier because it is shorter, the cohesion has not been reduced drastically, only the size of part of the module. The reason is that the module still does the exact same thing, no more, no less. In fact, some people can argue that by using a Dynarray to hold switch-case commands, you are actually trading a long switch-case statement for a long dynarray declaration. In essence, you are trading a control structure for a data structure. Less experienced programmers often perceive data structures used in lieu of control structures as harder to comprehend, and hence, maintain.

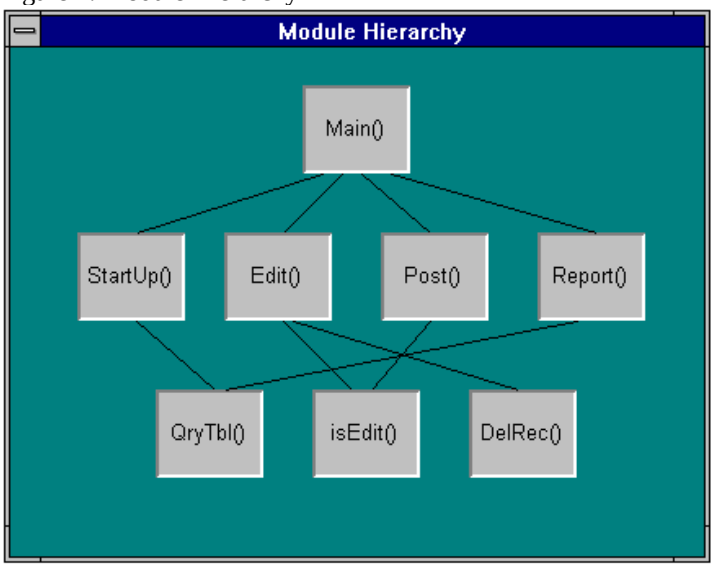
Code Listing 3. Using a Dynarray to Replace a long Switch/Case Statement

```
;-------------------------------------------------------------
; this proc starts the application up
;-------------------------------------------------------------
proc MAIN_MENU()
   private  CHOICE,            ; user's main menu choice
            EXIT_CHOICE,       ; variable holding user's exit method
           APP_PROCS           ; dynarray holding proc names to execute

   dynarray APP_PROCS[]

   ; initialize the application and load the APP_PROCS dynarray
   if STARTUP_APP(APP_PROCS) <> true then
      return "Paradox"
   endif

   while true

      ; display the main menu and enable/disable menu items
      MAIN_PULLDOWN()

      ; get the user's menu selection
      getmenuselection to CHOICE

      ; clear the pulldown
      showpulldown
      endmenu
      prompt ""

      switch
         case CHOICE = "Esc" :
            loop
         case CHOICE = "EXIT" :
            EXIT_CHOICE = GET_EXIT_CHOICE()
            if EXIT_CHOICE = "Paradox" or EXIT_CHOICE = "DOS" then
               quitloop
            endif
         otherwise :
            ; execute the procedure contained in APP_PROCS[]
            execproc APP_PROCS[CHOICE]
      endswitch

   endwhile

   SHUTDOWN_APP()
   return EXIT_CHOICE

endproc
```

So how do you go about building highly cohesive modules?

Well, in PAL programs, it is a good idea to decompose the programming problem into a hierarchy of cooperating procedures (see figure 4). Clearly identify the job description for each module. Try to keep each module as painfully ignorant as possible. After all, most other programmers aren't the genius you are. In fact, tomorrow you may not be the genius you are today. How many times have you looked at your own code, written months ago, baffled by what it does? If you clearly identify a *narrow* job description for each procedure, the chances for confusion later are less.

Figure 1. Module Hierarchy



To help you in writing module job descriptions, it might be helpful to keep in mind the following Rule of Module Intelligence (just made up, of course):

Bosses are ignorant. Workers are smart.

Just like in real life (ahem), endowing your various modules with different skill helps keep cohesion high. For example, the higher you go in a module hierarchy, the less nitty-gritty work the module actually does. In fact, higher level modules usually bark out commands, as the module in listing 4 does.

Code Listing 4. High-Level Module

```
procedure BIG_BOSS()

    if UNIVERSE_CREATE_THYSELF() = false then
        return false
    endif

    PROCESS_MAIN_MENU_SELECTION()

    if UNIVERSE_GO_AWAY() = false then
        SHOW_MESSAGE("Something bad happened during universe shut-down")
        return false
    endif

    return true

endproc
```

In this example, BIG_BOSS() initializes the application, calls the procedure which displays the main menu and handles the user's selection and then shuts the application down. This module does very little except issue a few key commands to subordinates.

Main procedures like BIG_BOSS() are fairly easy to write without losing cohesion. The real challenge is writing in lower-level procedures properly. Often, programmers use the procedure they are working on as a sort of scratch pad to explore possible alternatives to a solution. And often, a less experienced programmer will try to patch-up the scratch pad or cobble together the random bits and pieces of code into a working single module.

This can be dangerous.

If you find yourself exploring implementation alternatives in this ad-hoc manner, I don't want to discourage you. Just remember that when you are done experimenting, take a good hard look at your module and examine its cohesion. Most modules written after a long experimentation phase are ripe for problems. Why? First, the module was not trivial because it required some thought. Second, non-trivial modules are non-trivial because they try to do *too much for a person to comprehend at one time*. Our mission in life is to make *all modules trivial*. When all modules are trivial, cohesion is high and maintenance is easy.

After a scratch-pad session, I often stand back, take a look at the module, and break it up into a group of highly cohesive procedures, even if this can potentially break a working module. When it comes to programming cohesive, maintainable code, the old adage *if it ain't broke, don't fix it* does not apply. In fact, if anyone tells me that about a particular module, I get suspicious and find myself inevitable re-writing it so that it is understandable.

These lower-level modules which perform the important work take time to craft properly. As I write these modules, I make sure that each worker-bee procedure is specialized. That is, the module, even if somewhat complex, *does one thing and one thing only*. If the module has a simple job description, the chances of it doing the job exceedingly well increases.

Quit Talking!

The other issue, coupling, refers to what information needs to be communicated between modules.

In real life jobs, most bosses prefer to have workers that can perform their duties without excessive communication, either spurious or required. The more *self-reliant* the worker, the less maintenance the worker will require.

In programming in PAL, procedures need to communicate data between themselves. In fact, communicating data between modules is necessary to write anything other than a stupid little program, such as a "Hello world" program. Procedures transform data for the user. To do that, they must share data.

Writing modules with low coupling is desirable. Why? Because a module with low coupling is more independent and usually less complex because there is less data for the module to work on. The less a module has to know about data in other modules, or in any other part of the application, the better.

In PAL, procedures can share data via the following mechanisms:

1) Global variables
2) Argument lists
3) Tables
4) Text files, binary files

Let's discuss these in order. Global variables are perhaps the most common and pernicious source of coupling problems. If you carry around lots of global variables then at any given time a particular procedure *can* know about a variable. In reality, most modules can pretend that the global variables don't exist. In other words, a particular procedure may not *need* to know about a variable.

For this reason, if you have declared 25 global variables in the main procedure and procedure A() needs to know about three of the global variables, then only three variables are coupled to procedure A().

On the other hand, since global variables are not explicitly passed from the main procedure to procedure A(), then it may become unclear exactly what global variables procedure A() needs. In order to find out, you will have to read all the lines of code in A().

From this example you can see that it is not enough to have low coupling, but to have *explicitly stated* coupling. Global variables are *not* explicitly stated. Although you can list out the global variables a procedure needs as part of a comment above or just below the procedure declaration, the Paradox compiler can't prevent incorrect access to global variables.

For this reason it is a good idea to limit the number of global variables and institute rigorous procedures as to how you plan on accessing and updating global variables.

Passing variables from one procedure to another with an argument list (sometimes called a parameter list) is a better approach. With an argument list, the level of coupling is explicit and it is enforced by the runtime environment. However, if you declare module A() as it is shown below, you have a problem:

Procedure A(X,Y,W,H, Color, Amount, StatusFlag, isActual, EventHandler)

This procedure has nine arguments, which is a rather lengthy list. Let's take a closer look at the parameters in this fictitious procedure. StatusFlag is a variable which will cause A() to traverse a different path in the if statements inside A(). EventHandler is the name of a procedure which is executed whenever certain events occur. X,Y,W,H refer to the X and Y coordinates for the upper left-most corner of the display window and W and H are the window width and height. Color is the window color, Amount is a number which is displayed in the window and isActual is a flag which tells whether the number is an actual number or a budget or fake number.

As you can see from this parameter list, procedure A() is probably involved with the following tasks:

1) Setting window properties
2) Properly displaying user-defined data
3) Calling another procedure which handles certain events while the window is active.

Long and involved argument lists *usually* indicate trouble. It's much better form to replace this one-module-does-it-all procedure into three smaller procedures, one for each task listed above.

Tables, Tables Everywhere

In Paradox for DOS, as in most database applications, programmers use tables to store data that needs to hang around for a while. Paycheck information needs to be stored and reported on and it would be nice to keep several quarters of information on hand. This fact is so obvious that we tend to think of tables as the *whole enchilada* when it comes application development and forget that tables interact with source code in troublesome ways.

In Paradox for DOS, it is very common to have a table on the workspace while a series of very cohesive and apparently loosely coupled procedures work on the data in the table. From a purely programming perspective, working this way is rather odd, because the table represents a huge amount of global data. In fact, if you substitute the words *global variables* for *Paradox fields in table*, you can see that tables can frustrate attempts at reducing complexity.

Since Paradox for DOS language constructs operate on tables on an interactive workspace, this form of coupling is more obscure than a formal argument list. The reason is that you don't formally declare what fields or tables your procedure will require. Instead your simple use the VIEW command to put the table on the workspace.

Tables introduce several coupling problems. It is possible to lose track of what tables are on the workspace or in what order they were placed on the workspace. When this happens, the active procedure may not be able to see the fields it needs to see. I have seen several difficult bugs arise because the PAL programmer forgot to clear a table from the workspace, forgot to put it on the workspace, accidentally cleared it from the workspace or got confused as to the actual order of tables on the workspace.

Programmers can also get clever in accessing fields and tables on the workspace. They can write some short and tricky algorithms that refer to images by their number, rely on specific orderings of images or use Paradox keystroke commands such as DOWNIMAGE rather than an explicit MOVETO. Not only do tables increase the level of coupling, the ways the table are accessed can cause problems.

And since tables tend to have several if not many fields, the amount of coupling is much higher than the amount of coupling created by parameter lists and even global variables. For this reason, it is a good idea to follow some rigorous rules about workspace management. Here are some suggestions:

1. Clean up after yourself. Don't leave unneeded tables on the workspace.
2. Assume that other procedures forgot rule number 1 and verify the workspace's status.
3. Access tables and fields explicitly by using the table and field name rather than cursor movement commands such as RIGHT, LEFT, DOWN or UP.
4. Maintain rigorous, strict and consistent workspace management rules. Similar procedures should access the workspace in similar ways.
5. Generic coding techniques which violate the above rules are not forbidden. Just be prepared to spend lots more time on testing and debugging these procedures..

Other minor forms of coupling are text files and binary files. Since these files represent data that needs to be shared and since these files have far less significant structures than tables, the level of coupling these files introduce is far less than tables and global variables.

Variable Span and Life

Although not directly related to coupling and cohesion, but one that PAL programmers often forget is the concept of variable span and variable life. The average span of a variable is the average distance, in lines of code, between lines of code which refer to a variable. In the following procedure, variable X is used in one half of the procedure's lines of code, so it's average span is 1.

```
procedure COUNT_EM()
private X

  view "ORDERS"
  X=1
  scan
    X=X+1
  endscan

  ? X

endproc
```

A low average span is good, because that means that a variable is used frequently in a procedure. This indicates that the procedure's cohesion is high. If the variable span was high, say 30 lines of code, that would beg a question: What was going on in those 30 lines of code? Did those 30 lines of code have anything to do with the variable. Probably not. If this is the case than the procedure's cohesion is not very high.

Variable life refers to the number of lines of code between the first reference to a variable and the last reference to it. The larger this number is, the more lines of code the programmer has to keep in his or her head as he or she tracks

the variable through the procedure. If a variable's life is short, the procedure is probably short and hence, probably highly cohesive.

Visual Programming

Although the concepts of coupling and cohesion grew out of structured programming research and discussions in the 70's, they should not be confined to structured programming only. In this day and age of object-oriented programming and reusable code engineering, some people conclude that structured programming concepts don't apply.

Nothing could be further from the truth.

I like to look at coupling and cohesion in terms of visual programming metaphors. Two figures represent two different modules, Figure 2 is a pictoral representation of a loosely-coupled, highly cohesive program. Figure 3 is a representation of a highly-coupled, loosely cohesive program. Which figure is easier to discern and remember?

Figure 2. Loosely-Coupled, Highly Cohesive

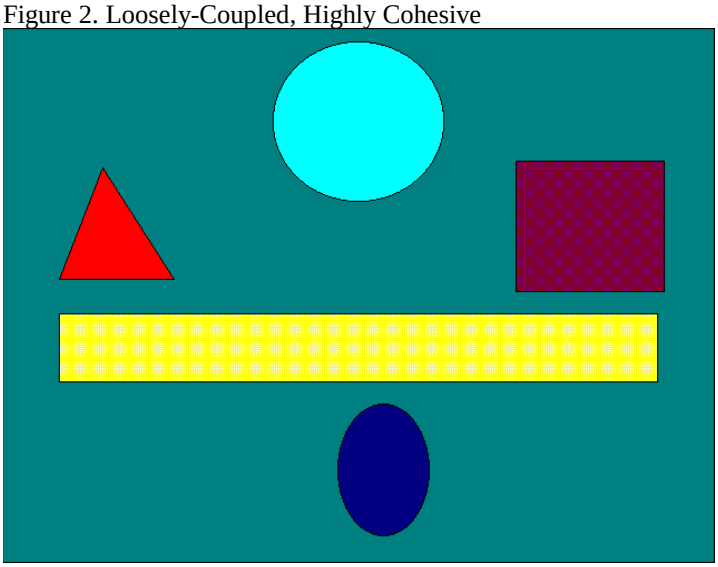
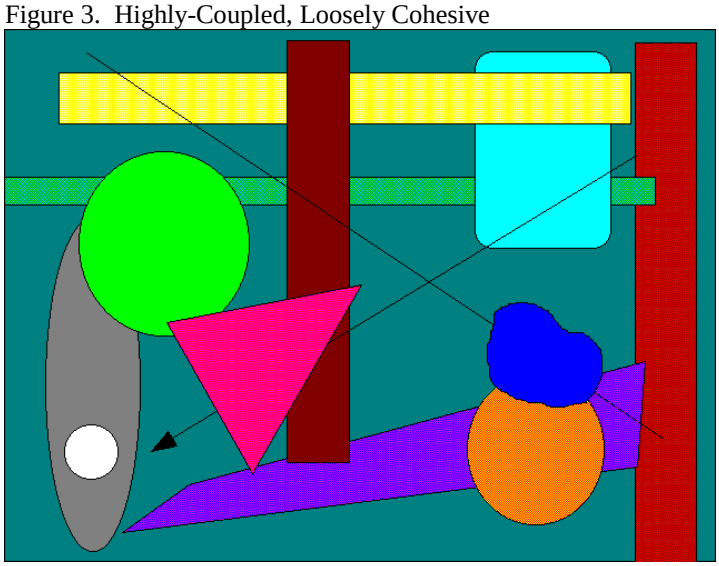


Figure 3. Highly-Coupled, Loosely Cohesive



It sometimes helps to look at these old problems in a new light -- a visual programming light. Since human visual processing capabilities far exceed abstract reasoning capabilities, framing a problem into purely visual terms increases comprehension of the problem. In addition, if a visual metaphor is used to describe a problem, abstract reasoning about the problem is enhanced. A visual metaphor frees us from messy details and lets us abstract the problem even further.

To me, coupling, cohesion, variable span and life can be discussed in purely visual terms. The problem can be stated this way:

1. *Smaller procedures are easier to understand because they contain less visual and hence logical information. This taxes the programmer's brain less.*

2. *The more information about a program that is immediately available to the programmer's eye at one time, the easier the program is to understand.*

3. *The more order and structure inherent in the information about a program, the easier it is to understand. The more random or chaotic the information in a program, the more difficult it is to understand.*

In the case of poorly cohesive programs, the information in the procedure may appear haphazard and unstructured to another programmer, hence the program is harder to understand. In much the same way it is harder for us to understand what is going on in a picture the more the picture contains random and haphazard features. Driving home at night in a heavy rainstorm is harder than driving home on a clear, sunny day. An obscure or complicated job description is harder to understand than a succinct and precise job description.

In the case of highly coupled programs, the amount of information in the program might be too high, as in the case of long parameter lists. Or, some information may not be immediately visible to the programmer, as in the case of global variables and tables.

If you think about programming in terms of information that is immediately and easily discernible to the programmer, it becomes clear that programs occupy "space" in programmers' minds. The larger and more chaotic that space, the more difficult it is to understand the program. For Paradox for DOS programmers, managing this "cognitive space" is extremely important.

For ObjectPAL programmers, the new visually-based metaphors in Paradox for Windows "shrink" the cognitive space for us by giving us visual objects rather than abstract ones and by applying a new, concrete order to the problem domain, increasing it's structure. But that is another story.