THE APPLICATION OF SOFTWARE ENGINEERING PRINCIPLES IN A DATA MIGRATION PROJECT

AN ARTICLE SUBMITTED BY GUY SIMONIAN
OF COTAL SYSTEMS, INC.

THE APPLICATION OF SOFTWARE ENGINEERING PRINCIPLES IN A DATA MIGRATION PROJECT

AN ARTICLE SUBMITTED BY GUY SIMONIAN OF COTAL SYSTEMS 860-523-4567, COTAL@DELPHI.COM 03/26/97 10:57 AM

keywords: software engineering, data migration, Year 2000, CASE, C++, Oracle

SCOPE

In 1988 I had the privilege of being one of the charter members of Digital Equipment's Software Engineering partner program. In that capacity and for following 5 years I dealt on a regular basis with process, products, and people involved in building new systems and reengineering old ones. At that time, my efforts were split between; 1) designing the architecture of the SEE's (software engineering environments) for large system proposals, and 2) conducting software quality assurance audits. As an independent consultant, I have been fortunate to have been able to apply in practice many of the principles I espoused. This document how those principles were applied, *what* I did, *how* it was done, and *why* it was done.

A QUALITY DIRECTIVE FOR SOFTWARE ENGINEERING

My engagement began in September of 1996. Fortunately, the director of data processing, who is also one of the founding members of the company, had designed several large systems and was an avid proponent of Relational System's Extended Relational Architecture (ERA). As W.E. Deming stated, quality starts at the top of an organization, and while managers may not actually perform the work, they must establish the goals and set the priorities.[1] The director understood that adherence the principles of the methodology, be it database architecture or software design for data migration was critical to the pursuit of quality. His directive to me at the time was to work in the area of the company that needed the most help, client conversions. I was to enable it with technology in such a way as to make the data migration process simple to perform and easy to replicate. I was to upgrade the quality of the systems involved while maintaining the exiting schedule for new customers. It was clear from that meeting that the company's growth and very existence hinged upon the ability of converting new customers quickly and efficiently. It was also apparent that to revamp the existing conversion process while dealing with a demanding and ever increasing workload would be a challenge. A considerable amount of effort had gone into the first three migrations that had taken place before I had arrived. Every new client however demanded a near heroic undertaking to achieve timely results.

SOFTWARE ENGINEERING PRINCIPLES APPLIED

The quality system I intended to build would have to adhere to many of the basic principles of good software engineering practices that have proven successful. The first four, Coupling, Complexity, Cohesion, and Shape generally form the basis for good design. Reuse is one result of good design, and is the principal component responsible for high productivity. Low Defect rates are another objective of good design, and is another metric used to indicate productivity.

COUPLING

Coupling of software modules refers to the quantity and type of interdependence between modules. 'Loose' coupling typically involves data coupling, for example, file sharing, while 'Tight' coupling would include parameter coupling, for example, global variable usage.[2] Thus low or 'loose' coupling is desirable.

COMPLEXITY

Complex programs are difficult to understand and maintain. Nested-If's, complex data structures and lack of naming conventions all contribute unnecessarily to complex routines. Complexity algorithms, for example Halstead's Software Science, McCabe's Cyclomatic Complexity Metric and others use rules based methods to produce a numerical representation of a module's complexity. Several re-engineering vendor tools actually can parse through an existing system, and build a color coded chart of the system. Overall size as measured in lines of code is another complexity metric. Thus, low complexity is desired.

COHESION

Cohesion of a software module refers to the ability of all of its tasks to perform one function well.[3] Highly functional modules are less error prone. Modules that perform 2 or more functions have low cohesion. Thus high cohesion is desirable.

SHAPE

Shape refers to the calling structure of the modules in the system. Psychological studies of "chunking" theory have suggested the number of 7 as optimal[4]. Also called *fan-out* or *span of control*, by experts[5], a practical range is between 5 and 10. A pyramidal shape where the top module calls 7 which in turn call 7 each is desired.

REUSE

The largest single factor that affects programmer productivity is the ability to reuse software modules. NCR for example, had an objective of 90% reuse on a recent project.[6] The projections for the OSI environment stated that a client conversion would be performed every two weeks meant effectively that a new software system had to be delivered every two weeks. Thus, highly reusability for both modules and the entire system is desired.

DEFECTS

Software defects can be caused by a variety of reasons, ranging from misinterpreting the business specifications to technical syntax errors. The cost or time it takes to repair software defects increases greatly with when they are identified along the process time line.[7] Obviously, low defects are desired, and defects found and corrected <u>early</u> in the process are desired over those handled later in the process.

OLD DATA MIGRATION PROCESS

The existing process for data migration that I was given consisted of five steps. Over time, I was able to eliminate three of these steps, thereby greatly reducing both the coding time and the data migration elapsed time.

OLD PROCESS DESCRIBED

1. Copy data from tape to disk.
2. Use SQL Windows® application, along with C++ dll's, Windows based initialization files to build a comma delimited flat file.
3. Use other C++ dll's and a different SQL Windows application or manually build scripts and run them using SQL Studio®, Wintalk®, or SQLPlus® to build the empty tables.
4. Manually define the SQL control files needed by SQL* Load® to populate the tables, column names, and data types.
5. Use a custom application, a front end that handles multiple files and tables to SQL* Load to populate the Oracle® schema.

OLD PROCESS & COMPLEXITY

The old process employed low level function calls that performed adequately, and which may have benefited by more use of structures and less use of pointers and bit level manipulation. A function that was most certainly more complex than needed were those that determined record types and called appropriate functions.

In Figure 1, one module switches processing based on a global structure defined elsewhere which carries the record type. It uses a symbol also defined elsewhere to branch to a set of instructions which perform the following;

a) determine which data structure to deal with based on a structure defined elsewhere

b) move data from the file buffer to a character data type

c) call one or more functions to unpack or manipulate that data,

d) open a data file using a stream pointer defined elsewhere

e) write that data element to the file.

This particular module went on for 858 lines, had 38 case statements many with nested if's and for-next loops, opened 38 different files, using approximately 800 different symbols. Note also the ambiguous naming conventions. On the positive side, it only called 6 different functions.

FIGURE 1

```
VOID convert_trailer_record_900_939(unsigned char ** p_trailer,long structure, int vli)
{

 char cBuffer  [200];
 unsigned char cBuffer2 [200];
 unsigned char cBufferArea [200];
 unsigned char *cBuffer3;
  unsigned char date[7];
 unsigned char *datework;
 unsigned char testdate[7];
 unsigned char testdateCC[9];
  int  L_vli;
 int  vli_indx;
 int  vli_indx_work;
 int  vli_indx_work2;
  datework = testdate;
 cBuffer3 = cBufferArea;

 switch (structure)
             {
               case DLRTRLR_900:
             sprintf(cBuffer,"%s,", G001_ACCTKEY);
             write_file(&dlrtrlr900stream,"900_dlrt.dat",cBuffer);
             sprintf(cBuffer,"%s,", G001A_BRNBR);
             write_file(&dlrtrlr900stream,NULL,cBuffer);
             sprintf(cBuffer,"%s,", G001B_SEQNBR);
             write_file(&dlrtrlr900stream,NULL,cBuffer);
             sprintf(cBuffer,"%s,", G001C_ACCTNBR);
             write_file(&dlrtrlr900stream,NULL,cBuffer);
             ncrK2Char( *(p_trailer) + 2, T900_DLRTRLR_data.C002_TXNST, 1 );
             sprintf(cBuffer,"%s,", T900_DLRTRLR_data.C002_TXNST);
             .
             .
             .
             ncrK2Char( *(p_trailer) + 107, cBuffer2, 3 );
             cBuffer3 = cBufferArea;
             cBuffer3=convert_to_float(cBuffer2,2);
             sprintf(cBuffer,"%s,", cBuffer3);
             write_file(&dlrtrlr900stream,NULL,cBuffer);

               case LNOPTRLR_904:
             sprintf(cBuffer,"%s,", G001_ACCTKEY);
             write_file(&lnoptrlr904stream,"904_lnop.dat",cBuffer);
             sprintf(cBuffer,"%s,", G001A_BRNBR);
             write_file(&lnoptrlr904stream,NULL,cBuffer);
             .
             .
             .
```

OLD PROCESS & SIZE COMPLEXITY

The old process used a lot of code to do a little work. The use of the fourth generation GUI for example added little value to the process. The 'end user' so to speak were the same programmers that were constructing the software application, so no training or orientation was required. The input file parameters found in the .INI file stayed fixed throughout the duration of the process, so the coding overhead associated with the calls to GetPrivateProfileString that accessed the .INI file were unnecessary. The amount of coding required to build this application included;

1. Step 1 required no coding.

2. Step 2 required 2 source modules lines, and 2 header files occupying 16,647 lines.

3. Step 3 was constructed manually, one line for each column definition, using 2000 lines of code.

4. Step 4 was comprised of 45 control files occupying 2192 lines of code.

5. Step 5 required no coding.

In total, 20,839 lines of code are required by this process.

OLD PROCESS & COUPLING

The old process employed almost all levels of coupling, though predominantly loose coupling. Data files were used to set program parameters, pointers to file input buffers determined the current section of data being migrated, header files indicated were overused to define symbols and global variables. Parameter passing in function headers ranged between 3 and 7 parameters. With the exception of the header files, loose coupling predominated. There was just too much coupling required by the current architecture.

OLD PROCESS & COHESION

The tasks of the low level modules performed one function, and therefore had high cohesion. See Figure 2 for an example. The higher level modules, like the one shown in Figure 1 performed more than one function. In fact to describe the Figure 1 module we would have to include four tasks as in, "Determine which record structure is currently being handled, then create the appropriate file. Process the data items as needed, then populate the open file with the translated data." This module therefore had low cohesion.

FIGURE 2

```
/* This function checks for invalid month end dates and adjusts the date accordingly.
  Currently  assumes YYMMDD format for input and return value.                    */

void check_mend(unsigned char *date)
{
int day;
 int month;
 int year;
 unsigned char cbuffer[3];
strcpy(cbuffer,date+4);
 day=atoi(cbuffer);
memmove(cbuffer,date+2,2);
 memset(cbuffer+2,'\0',1);
 month=atoi(cbuffer);
 memmove(cbuffer,date,2);
 memset(cbuffer+2,'\0',1);
 year=atoi(cbuffer);
switch(month)
     {

    case 2:
         if (day < 29)
              break;
         if ( ((year % 4 == 0) && (year % 100 !=0))  | |  (year % 400 ==0) )
              memmove(date+4,"29",2);
         else
              memmove(date+4,"28",2);
         break;
    case 4:
    case 6:
    case 9:
    case 11:
         if(day > 30)
            memmove(date+4,"30",2);
           break;
    default:
         break;
     }
}
```

OLD PROCESS & SHAPE

From a navigation perspective, the compiled module portion of this program consisted of two source files and two header files, one source that was 15,000 lines long, and the other with 1400 lines. The shape of this process could be described as series of columns standing alone, rather than a pyramid building upon its base.

OLD PROCESS & REUSE

Of the entire 20,839 lines of custom code written for the project, 2107 lines, or just **10% is reusable.**

OLD PROCESS & DEFECT CORRECTION

The existing process impeded the efficiency of unit testing. Unit testing is required to perform tasks like the implementation of routines to handle new data types, for example, sign packed data, Julian Date types, etc.

a) The presence of the SQL Windows in combination with the C code implemented as a DLL made use of the Visual C++ debugger difficult and impractical for step 2 testing. Physical examination of the comma delimited files produced by step 2 could reveal some of the defects, but not all of them.

b) Syntax errors in step 3 would be found immediately as the SQL executed to build the tables. Size or format errors induced here, however, would not be found until steps 4 or 5.

c) Errors would also crop up in the .CTL load files needed in step 4. An example of the control file syntax is found in Figure 3. Consequently, as testing was pushed out to the later steps, the time to correct defects went up exponentially.

d) Load errors uncovered after running step 5 for 2 or 3 hours may have been induced by coding errors in step 2. The C coding error would then be corrected, and steps 2 through 5 rerun.

FIGURE 3

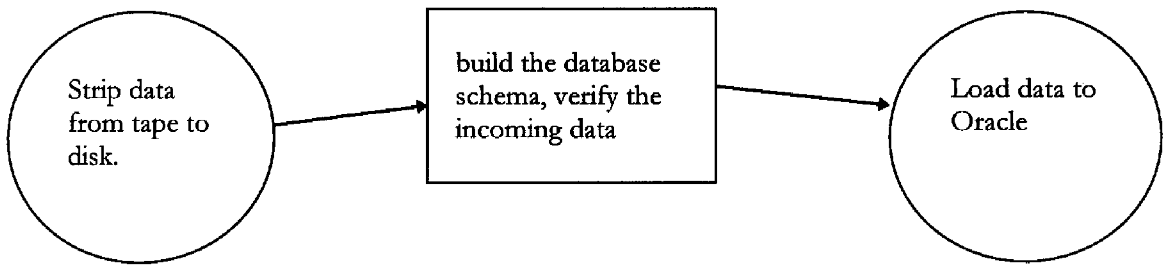
```
LOAD DATA INFILE 'SVFDF1.DAT' REPLACE
INTO TABLE SVFDF1 TRAILING NULLCOLS
(
  FILE_ID                          POSITION (1:6) ,
  SAVINGS_MAJOR_ACCOUNT_TYPE       POSITION (7:8) ,
  SAVINGS_ACCOUNT_NUMBER           POSITION (9:22),
  EFFECTIVEPLACED_DATE             POSITION (23:30),
  SEQUENCE_ID                      POSITION (31:33),
  TYPE_OF_FLOAT                    POSITION (34:37),
  FLOAT_AMOUNT                     POSITION (38:49) ZONED (12,2),
  TOTAL_AMOUNT                     POSITION (50:61) ZONED (12,2),
  SPLIT_FLAG                       POSITION (62:62),
           .
           .
           .
```

NEW DATA MIGRATION PROCESS

NEW PROCESS DESCRIBED

1. Copy data from tape to disk.

2. Use one C++ program to build the Oracle tables and columns, mine the incoming data, and load the data to that schema.


Strip data from tape to disk.
build the database schema, verify the incoming data
Load data to Oracle

NEW PROCESS & COMPLEXITY

The most highly complex modules like the one shown in Figure 1 were targeted for re-engineering since that could benefit the most from reduced complexity.

Figure 4 shows the new LAYOUT module. It's function could be described as the following;

a) Create the Oracle table for this particular record

b) Process the data items on the basis of offset, length, precision, and type

This particular module was 231 lines long, had two if-then-else statements, and called 5 different functions.

FIGURE 4

```
/*==========================================================*/
  VOID convert_lnd40(unsigned char *disk_record)
/*==========================================================*/

 {

static int second_Pass_Ora_Flag; //static to each routine

table_build("LND40",second_Pass_Ora_Flag);
rec_num(); //grab current record number
munge.xlate (Rec_Num_Key, 0,18, 'u',0,"Rec_Num_Key"); // write to database

// parameters below are disk buffer, starting position, length, type, precision, Field Name
munge.xlate(disk_record,2,4,'E',0,"BINARY_LENGTH");
munge.xlate(disk_record,6,2,'E',0,"SUBSYSTEM_NUMBER");
munge.xlate(disk_record,8,4,'E',0,"BANK_NUMBER");
munge.xlate(disk_record,12,20,'E',0,"ACCOUNT_NUMBER");
munge.xlate(disk_record,32,4,'E',0,"RECORD_TYPE");
munge.xlate(disk_record,36,12,'U',4,"PREVIOUS_RATE");
munge.xlate(disk_record,48,2,'E',0,"RECAP");
munge.xlate(disk_record,50,2,'E',0,"INSTRUCT_EXP");
munge.xlate(disk_record,52,2,'E',0,"TYPE_INQUIRY");
.
.
.
```

NEW PROCESS & SIZE COMPLEXITY

Seven programs now comprise the data migration program set:

xxxx.cpp This is the main function that determines the file set to be translated., 284 lines

conv.cpp sets the navigation logic to parse through the file being translated, 1044 lines

layout.cpp	'lays out' the offsets of the fields of the record being processed, 5975 lines
putout.cpp	handles optional comma delimited output files classes, 2242 lines
munge.cpp	Invokes the translation routines for the data being processed, 653 lines
func.c	houses specific routines for bit and character manipulation, 2037 lines
orai.c	has low level calls to the Oracle Call Interface, 235 lines

The total lines of code including these files and 3 header files (352 lines) is 12,822 lines. Several low level routines that performed similar tasks with slight variations (unpacking packed numbers, with or without signs, signs at the front or rear of the field), were combined into one module (see Figure 5A & 5B). This system is now 40% smaller than the old system yet performs more tasks.

FIGURE 5A

```
   /*==================================================*/
short int  Packed2Char( unsigned char *pPack, unsigned char *pChar, short nLen, short odd_length )
   /*==================================================*/

   /*****************************************************************
    GS -2/7/97 - generic unpacker of signed data, sign at front or back of the number
    Restrictions:
        precision must be less than length of field
        sign nibble can be first or second nibble of first character, or last nibble of last char
        typically D is negative sign, more can be added by changing case statement at bottom routine

        arguments:
            1.  unsigned char pointer to the packed decimal number
            2.  unsigned char pointer to receive the unpacked digits
            3.  short int length of argument 1, the packed decimal number.
            4. short int flag indicating nibble at the end of pPack

    Generically can handle the follow 8 packed datatypes;

    Even length signed packed with precision      e.g. S9(3)V99   (the sign counts as 1)
     "          "         without               "     S9(3)
    Odd length    "          with               "     S9(2)V99
     "          "         without               "     S9(2)
    Even length unsigned packed  "      "         9(2)
     "          "         with                  "       9(2)V99
    Odd         "          "                    "       9(3)V99
     "          "         without               "       9(3)

   returns; character buffer with number, including decimal point, sign at end if present
   ******************************************************************************/
{
  int error=0;
  char cBuffer[200];
  unsigned char cBufferArea [200];
  char savebuff[200];
  char buff[200];
    short int          iW;
    short int          iI;                   /* loop index */
    unsigned short int   iFirstNib;       /* first nibble */
    unsigned short int   iLastNib;        /* last nibble */
    unsigned short int   iSignNib = 0; /*Nibble that contains the sign*/
    unsigned char        *pWork;         /* work pointer for packed number */
    unsigned char        *cBuffer3;      /* work pointer for packed number */
    int nDigits=0;
    int nlen;                                /* pointer offset to stream */
    nDigits=0;
     //   continued in FIGURE 5B
```

FIGURE 5B

```
memset( pChar, '\0', (nLen * 2) + 1 );      /* initialize char array */
pWork = pPack;                              /* copy pointer to work */
for( iI = 1; iI <= nLen; iI++ )
{
    iFirstNib = ( *pWork & 0xf0 ) >> 4;  /* obtain high order 4 bits */
    iLastNib = *pWork & 0x0f;             /* obtain low order 4 bits */
if ( iFirstNib > 0x09 )                     /* check first nibble */
       {                                    // must be a number if not a sign
       iSignNib = iFirstNib;
        if ((iI > 1) && (iI < nLen) )
         w_EDC_error_msg("Sign character found where it does not belong... ",1);
       }
if   ( (iI == nLen) && (odd_length==1) )   //ignore cases where bogus nibble
      iLastNib =0;                           // is found in last nibble of odd length field
 if (iLastNib > 0x09)
             iSignNib = iLastNib;
if (!(iFirstNib > 0x09))                            //if First nibble is not a number
    *(pChar + nDigits++) = iFirstNib | 0x30;  //          then skip it
if (!(iLastNib > 0x09))                             //if First nibble is not a number
*(pChar + nDigits++) = iLastNib | 0x30;         // make ascii character and bump outputpointer
pWork++;                                            // point to the next input byte
}                                                   // end for loop
switch ( iSignNib )                                 // make negative if needed
{
    case 0x0f:
            *(pChar + nDigits) = '+';
        break;
    case 0x0e:
            *(pChar + nDigits) = '+';
        break;
  case 0x0d:
     *(pChar + nDigits) = '-';          //negative number
        break;
  case 0x0c:
     *(pChar + nDigits) = '+';          // anything else is positive number
        break;
    case 0x0b:
            *(pChar + nDigits) = '+';
        break;
    case 0x0a:
            *(pChar + nDigits) = '+';
        break;
    case 0x00:
            *(pChar + nDigits-1) = '+';
       break;
    default:
            *(pChar + nDigits-1) = '+';
       break;
}
return( 0 );
}
```

NEW PROCESS & COUPLING

The elimination of the extra 3 steps in the old process reduced the 'loose' coupling associated with the file based data coupling. This was by far the largest factor contributing to the coupling metric. In addition, global variables are no longer being used to define data structures associated with the field processing. Parameter passing in function headers is the same, ranging from none to 8 variables. Default values for function header parameters has reduced this coupling factor.

NEW PROCESS & COHESION

The module shown in Figure 4 performed two functions, and therefore had relatively high cohesion. The 7 current files each perform one function, therefore they have high cohesion.

NEW PROCESS & SHAPE

The *fan-out* associated with the new process approaches the pyramidal shape desired. Of the 7 program files in the system, 4 are typically modified during the coding process. File editing navigation using the Visual C++® environment, or other code editors like Code*w*right® is simplified by invoking the control TAB keystroke to move from file to file. This is preferred over the setting of bookmarks, or searching for names in one or two large files. A piece of the call tree is shown in Figure 6.

FIGURE 6

```
main(),  in xxxx.cpp, select file to process
    |
 disk_input, in conv.cpp sets record counters, record type
      |
  convert_abc(), in layout.cpp  selects field offset, length, type
        |   └table_create(), in orai.c builds table in Oracle
        |
   munge.xlate(), in munge.cpp extracts specified data
            |  └check_date(), in func.c checks format & compliance
            |
    Packed2Char(), in func.c unpacks signed field
```

NEW PROCESS & REUSE

Module reuse includes 90% of the xxxx.cpp function, 70% of conv.cpp, 30% of layout.cpp, 71% of putout.cpp, all of munge.cpp, all of func.c, and all of orai.c, and all of the header files. Of the 12,822 lines of code in this project, 7648 or **60% is reusable**, and has been demonstrated as such in a subsequent project.

NEW PROCESS & DEFECT CORRECTION

Defect identification within all modules was improved because:

a) One could now specify any starting record number, and break into the debugger at any line.

b) SQL Create table script errors induced in step 3 of the old process have been eliminated since these scripts are no longer used.

c) Syntax errors induced in step 4 of the old process (.CTL SQL* Load files) have been totally eliminated since load files are no longer used.

d) The C++ environment is far better suited than SQL* Load at defect identification in general. SQL* Load has no in line messaging or debug capability.

e) The munge object constructed has the capability of detecting Oracle load errors on a field by field basis. The guesswork associated with row level load errors has been eliminated.

Add or Change a Field using Old Process	Add or Change a Field using New Process
code field change	code field change
Run	Observe Output on the Screen during Run
Examine comma delimited file	Examine field in Database
Add field definition to Create Table SQL	
Drop/Create table in SQL	
Add field to .CTL Load file for SQL* Load	
Load using SQL* Load or custom application	
Examine SQL* Load logs for errors	
Speculate as to field causing load error	
Examine field in Database	

Because the defect identification capability was pushed much earlier in the process, hours and even days of time needed in the past for defect correction have been eliminated. The ten step process formerly required is now comprised of 3 steps. The principles of integrity degradation observed in all engineered systems as related to complexity also holds true here. A 300% improvement is a conservative estimate of the cost or time savings obtained here.

DATATYPES MIGRATED

Input Type	Input HEX Samples	Output of Samples	Output DataType
Unsigned Packed	34 65	+3465	Number(4,0)

Unsigned Packed with precision	34 65	+34.65	Number(4,2)
Packed Sign in Front	0D 76 54	-7654	Number(4,0)
Packed Sign in Front with precision	0D 76 54	-76.54	Number(4,2)
Packed Sign at Rear	76 23 1C	+76231	Number(5,0)
Packed Sign at Rear with precision	76 23 1C	+762.31	Number(5,2)
Unpacked ASCII	34 31 32 33	4123	Number(4,0)
Unpacked ASCII with precision	34 31 32 33	41.23	Number(4,2)
Unpacked EBCDIC	F3 F2 F1	321	Number(3,0)
Unpacked EBCDIC with precision	F3 F2 F1	32.1	Number(3,1)
IBM DOS/VSE unpacked with packed sign	F2 F3 C0	+230	Number(3,0)
	F6 F3 D5	-635	
Binary	81	10000001	VARCHAR2
Block Length Indicator	02 C0	704	Number(8,0)
ASCII	41 42 43	ABC	VARCHAR2

DATE DATATYPES MIGRATED

<u>Input Type</u>	<u>Input HEX Samples</u>	<u>Output of Samples</u>	<u>Output DataType</u>
Packed YYMM packed	34 12	Dec 1, 1934	Date
YYYYMMDD packed	19 97 05 15	May 15, 1997	Date
MMDDYY packed	01 31 55	Jan 31, 1955	Date
MMDDYY unpacked	30 32 31 35 38 36	Feb, 15, 1986	Date
MMYY packed	12 34	Dec 1, 1934	Date
Unpacked EBCDIC	F2 F0 F0 F1 F0 F9	Sept. 21, 2001	Date

YYYYMMDD	F2 F1		
YYDDD (Julian) packed	85 02 90	Jan 29, 1985	Date
YY0DDD (Julian) packed	85 00 29	Jan 29, 1985	Date
YYMMDD unpacked	38 39 30 35 30 36	May 6, 1989	Date
YYYYDDD packed	19 97 00 32	Feb 1, 1997	Date

YEAR 2000 ISSUES

Dates migrated without Century indicators are handled with a combination of business rules regarding the information, and technically with an intelligent date format mask. For example, mortgage instrument maturity dates with YY=01 will always indicate the year 2001, as will a birth date. Oracle has a handy date format mask 'RR' whose behavior is determined by the current year, and the specified year.

	Year 0 - Year 49 Specified	**Year 50 - Year 99 Specified**
Present year is 00-49	Current Century date returned	Century before current century is returned
Present year is 50 - 99	Century after current Century returned	Current Century returned

A Year 2000 date format example from the Oracle7 Server SQL Language Reference Manual, p. 3-55

```
SELECT TO_CHAR ( TO_DATE ( '27-OCT-95' , 'DD-MON-RR' ), 'YYYY') "4-digit- year"
FROM DUAL

4-digit year

        1995
```

```
SELECT TO_CHAR ( TO_DATE ( '27-OCT-17' , 'DD-MON-RR' ), 'YYYY') "4-digit- year"
FROM DUAL

4-digit year

        2017
```

CONCLUSIONS

The application of some basic software engineering principles to the data migration process is not a complicated task. All you really need are three things;

1. Sponsorship: A high level manager with a commitment to quality.

2. A Champion: You or 1 or more of your team members.

3. Doing It!: Getting the work done.

Guy Simonian lives and works in West Hartford, Ct, with his wife Darlene and their dog, Smudge. You can reach him by writing to cotal@delphi.com.

[1] Deming, W. E. *Out of the Crisis*, Cambridge, MA: MIT Center for Advanced Engineering Study, 1982.

[2] Martin, James & McClure, Carma *STRUCTURED TECHNIQUES: The Basis for CASE*, Prentice Hall 1988, p. 75-77

[3] Grady, Robert B. *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall 1992, p. 232

[4] Miller, G. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review*, 1956

[5] Yourdon, Ed & Constantine, Larry *Structured Design*, Yourdon Press 1975, p. 95-126

[6] Martin, James & Odell, James *Object-Oriented Analysis and Design,* Prentice Hall, 1992, p. 37

[7] Grady, p. 202