

## Writing programs to update your programs

Following on from the last months “IBM i the present and the future” we are going to look at how application mapping can be used to actively change an entire system programmatically. Using the application map as a primary input, some simple reengineering concepts, and a fair amount of time to perfect, programs can be written to update application programs. This approach has saved many companies literally thousands of man-hours and millions of dollars – rare commodities in current times.

Writing programs to update programs is typically used to make structural changes to the application source, not functional changes. When a system enhancement produces a huge number of fairly simple system wide changes, then automating these changes programmatically begins to make sense. There is of course a critical mass at which such an approach becomes viable. The most brutal and obvious example of this was the Y2K “bug”. Some companies spent as much as 5 million dollars to change their systems. Some companies used programmes to carry out the same amount of work on similar sized systems for 5% of the cost. How did they do that and why is this relevant nearly ten years on?

After 20 or 30 years of an applications life it is fairly safe to assume that at some stage, there might be a business demand to change important and well used fields in the database. This might be driven by Industry Standardisation, system integration or upgrades, internationalisation, or just plain old commercial growth: you run out of invoice numbers or even customer numbers.

Y2K affected almost every RPG application in existence. It also affected just about the entire application in each case. Since 2000 most systems have grown at a rate of 10% per year. It’s a widely acknowledged fact that RPG resources have not kept pace with this growth, in reality they have probably reduced by the same amount each year. So while database changes are now generally industry or company specific, the problems and their related solutions remain the same, but with more code affected and less people to fix it.

There are other applications for automated reengineering of a system which I will briefly mention later in this article. Solving a field expansion problem is however quite relevant for many companies, so I will use it to flesh out the subject of this article in more detail.

## An Engineered Approach

A more conventional approach to solving this type of problem is to get a “feel” for the scope and size of the problem, understand clearly the requirements for the change, and then send between 1 and an army of developers off to fix the problem one program at a time see figure 1.

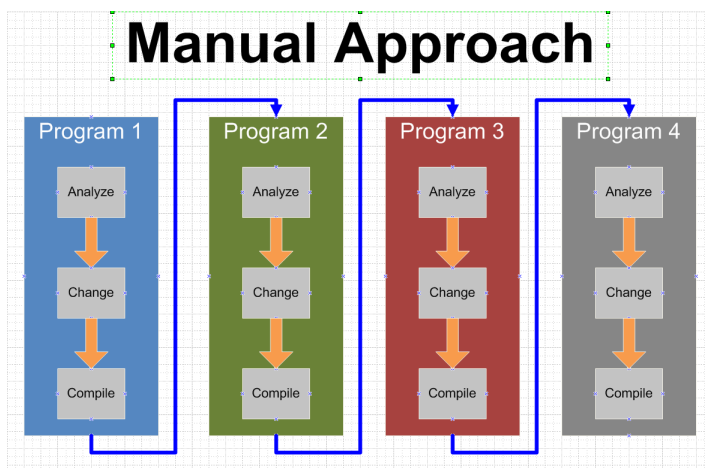


Figure 1

There are many problems associated with this approach. Here are a few:

- labor-intensive
- scope and time-lines are vague at best
- not repeatable
- much more prone to human error & inconsistencies
- much riskier

The upside is of course that such an approach requires very little preparation, and very little initial investment in time or money. It is also generally very flexible and so useful for very small projects.

The basis of an engineered approach is to break down the process into a set of discrete, repeatable, and automated steps. Each step is then applied across the entire system or project and repeated until an optimum result is achieved. Figure 2 shows diagrammatically how this compares to a conventional manual approach.

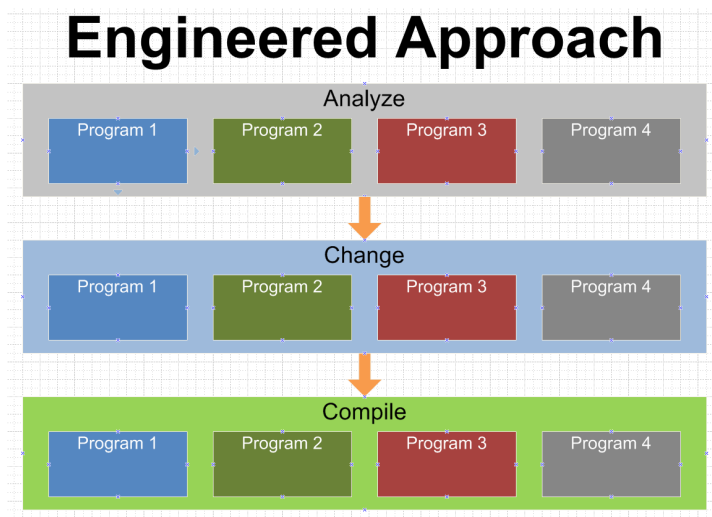


Figure 2

There are many benefits associated with a structured and engineered approach. Some of these include:

- each step is repeatable and so can be perfected
- the outcome is more predictable
- scope and approach can be changed without a loss of expended effort
- latest code version can be introduced at the last minute
- far fewer resources required
- much quicker
- potentially less testing as changes are consistent

Without an explicit, detailed, and very precise measurement of the impact of a database change across the system, automating the required changes would not be possible, so let's start by looking at this in more detail.

### Establishing the precise scope of a task

Even in well designed and documented systems, the impact of changing the database of an integrated and complex application on IBM i can be huge. Just the recompilation and data copying tasks can create logistical nightmares. The most significant and difficult task is of course measuring this impact on the source code across the entire system. If analysis is done right, subsequent work will be highly predictable and measurable. Do this wrong, and the results could be catastrophic. Aside from causing overruns in project timelines, I don't think I need elucidate the potential outcome of having "missed" something in a production system.

### *Specifying Fields to be changed*

The first task in the analysis stage is to specify which fields need changing in the database. This should be straightforward but may be complicated by virtue of integrated systems, poor documentation and often a combination of both. The next step described below may actually produce results that warrant additional fields be added and be included in the process.

### *Finding where fields are used*

The next step is to establish precisely where these fields are used throughout the system. This is where things start to get tricky. Establishing the explicit use of a given field by its name can be achieved with a simple FNSTRPDM command. One then needs to start at these specific points and establish where these fields are associated with any other variable or data construct, by virtue of a compute or definition statement. There is only one way to do this, and that's to read the source code of every single instance where the field being changed is used. RPG applications have many technical constructs that make this type of analysis complex and time consuming. Let's have a look at some of these that are relevant:

- Variable names not matching or resembling field names,
- the use of Prefix or Rename keys words in the programs
- tracing input and calling parameters
- CL's having no file definitions
- Data Structures & Arrays
- Input & return parameters not being defined in procedure prototypes.

Legacy cross-reference tools can help with this analysis up to a point. That point ends at each level or instance of a variable, so many – sometimes thousands of individual queries need to be run and amalgamated using these older technologies. Figure 3 shows a simple example of where conventional approaches to where used analysis of tracing the CUSNO field stop.

```
C* If customer number not provided then prompt
C   CSTMER      IFEQ      *ZEROS
C               CALL      'CUSFSEL'          99
C               PARM      CSTMER
C               ENDIF
C* Set customer number
C               MOVE      CSTMER      CUSNO
```

**Figure 3**

The obvious answer to this problem is to pre-build an application map of the entire system being analyzed, whereby variable-field-variable associations are instantly available. Using this map one can write a program that will trace a field throughout all its iterations and variants across the entire system in a single query. Some of the trace work is subcontracted to a previous stage in the form of pre-building the application map. Let's look at an example of this at work.

```

0001.00      PGM          PARM(&CUSNN)
0002.00      DCL          VAR(&CUSNN) TYPE(*DEC) LEN(5 0)
0003.00      DCL          VAR(&CUSNC) TYPE(*CHAR) LEN(5)
0004.00      DCL          VAR(&LETSQ) TYPE(*DEC) LEN(3 0)
0005.00      DCL          VAR(&LETNR) TYPE(*CHAR) LEN(3)
0006.00      DCL          VAR(&prefix) TYPE(*CHAR) LEN(5)
0007.00      CHGVAR      &CUSNC &CUSNN
0008.00      CALL          CUSLET1  (&CUSNC)
0009.00      ENDPGM

```

Figure 4

Figure 4 shows the source of a CLP named CUSLET. If I were carry out a traditional analysis on a system with this program in it, looking for the impact of a change to the field CUSNO, this program would not show in the results.

```

0032.00      * Retrieve last order number
0033.00      C            *HIVAL      SETGTCUSTSL3
0034.00      C            READPCUSTSR      4040
0035.00      C            *IN40      IFEQ *ON
0036.00      C            Z-ADD1      DSORDN
0037.00      C            ELSE
0038.00      C            CUSNO      ADD 1      DSORDN
0039.00      C            END
0040.00      *
0041.00      C            MOVE'0'      *IN34
0042.00      C            MOVE'1'      *IN33
0043.00      C            CALL 'CUSLET'
0044.00      C            PARM          CUSNO

```

Figure 5

Figure 5 however shows a snippet of the source of an RPG program that calls CLP CUSLET passing the parameter CUSNO. Now looking at the spreadsheet of results of our extraction program written over the application map in Figure 6, we can see the CUSLET has been included in the analysis results. This is because the parameter CUSNO was passed to CUSLET from the RPG program displayed in Figure 5.

E129		QCLSRC	
	A	B	C
3	Name	Seq No	*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
115	CUSFTRAND	0003.20	A 2CUSNO SY OB 5 36DSPATR(HI)
116	CUSFTRAND	0008.00	A 2CUSNO SY OB 5 36DSPATR(HI)
117	CUSLET	0001.00	PGM PARM(&CUSNN)
118	CUSLET	0002.00	DCL VAR(&CUSNN) TYPE(*DEC) LEN(5 0)
119	CUSLET	0003.00	DCL VAR(&CUSNC) TYPE(*CHAR) LEN(5)
120	CUSLET	0007.00	CHGVAR &CUSNC &CUSNN
121	CUSLET	0008.00	CALL CUSLET1 (&CUSNC)
122	CUSLETSQ	0003.00	C PARM CUSNC 5
123	CUSLETSQ	0005.00	C MOVE CUSNC CUSNO
124	CUSLETSQ	0006.00	C CUSNO CHAINCUSFL3 81
125	CUSLET1	0001.00	PGM PARM(&CUSNC)

Figure 6

The output of this analysis is a specific list of all source members and lines therein that are affected by the proposed field changes.

***Making the required changes programmatically***

Changes that can be made without causing any conflicts can be done so programmatically. The percentage of these against the total changes required may vary from project to project, but essentially this can be fully automated with a carefully written program. The tedious and time consuming part of writing a program to do this is catering for all instances or specific types of change. Nevertheless this can provide a very significant productivity gain in any project. There are different standards that can be used to notate and make the change such as making comments in margins, commenting out

replaced code, or just overwriting existing code. This can be done one way during iterative trial conversions, and changed for a production conversion with little effort.

These programmatic changes can be categorized into two types:

**Direct Definition Changes** – where database fields or variables that can be traced back to database fields are defined.

This includes files, displays, reports, and programs (RPG or CL) and refers to D specs, arrays, in-line calc specs amongst others. This type of change is straightforward and is the most obvious candidate for programmatic change. Figure 7 shows the source of a physical file that has been programmatically updated and the original code commented out. Columns 1-5 have had the programmers name added for audit purposes.

```
0001.00      * SIMON      24/02/06  A0000001      Default Audit, delete if not
0002.00      * Active Contacts File:
0003.00      A          R RCNTAC
0004.00 KAMALA*      CUSNO          5P 0          TEXT('Cus. No.')
0005.00 KAMALA      CUSNO          7P 0          TEXT('Cus. No.')
0006.00      A          COLHDG('Cus.' 'No.')
```

Figure 7

**Indirect Definition Changes** - In some cases the direct definition changes will have a “knock-on” effect. For example if a field is expanded by two digits, and this field is used before the end of an internal data structure in an RPG program, the other elements in the data structure must be adjusted to accommodate this change. Similarly in a print file format, a column size increase may require columns to the right to be shifted to make space. In some cases this “knock-on” effect may actually cause conflicts of vary types. These conflicts might be resolved by using clever algorithms in the programs that make the changes, but usually conflicts require human intervention. Figure 8 shows an example how the data structure definition is adjusted, the second element is expanded, and subsequent elements are moved to accommodate this change. This type of change is fairly straightforward to program into the automated process. The time consuming part is finding and allowing for all different types patterns of instances in a system. As such the repeated use and fine tuning of programs that make changes to programs, makes them naturally more useful with each successive project.

```
0003.40 D*SRP314      ds          32
0003.50 D SRP314      ds          34
0003.60 D  p_cono          1          2  0
0003.70 D  p_dvno          3          5  0
0003.80 D*  p_cust          6         10  0
0003.90 D  p_cust          6         12  0
0004.00 D*  p_cslc         11         20
0004.10 D  p_cslc         13         22
0004.20 D*  p_cnno         21         25  0
0004.30 D  p_cnno         23         27  0
0004.40 D*  p_svit         26         32  0
0004.50 D  p_svit         28         34  0
0004.60
```

Figure 8

### ***Managing Design Conflicts & Manual Intervention***

In virtually every field expansion project there will be design problems that arise from the proposed changes. These might vary from a simple overlay or overflow on a report, to embedded business logic based upon a field substring. Whilst it may not be possible to automatically make changes to these constructs, it is possible to programmatically identify where they occur. Again the role of the pre-built application map is critical to this process as a primary input to the search

algorithms. These conflicts can be clearly identified by subtracting the changes made programmatically from the total required changes. These conflicts can generally be categorized as follows:

**Device Problems** – where any direct change or shuffling of affected columns runs out of space.

**Program Problems** – an example of such a problem is lines where there may be a conversion problem because a resized field (or a field dependent upon it) is a sub-field in a structure that may not be resized. Another example is where a work field is used in a program by two fields at various stages. One field is being resized, and the other is not. Again this requires design logic to resolve.

**Database Problems** – the whole process above starts by specifying which fields will be changed. The Where Used analysis when run on resized database fields might trace to fields not included in the resize exercise. This may or may not be a problem but generally must be assessed manually.

Some of these problems might be resolved by making some manual changes prior to rerunning the analysis and programmatic changes. In certain cases this might have an exponential effect of removing problems with a conversion project. In other cases it will be necessary to make these design decisions and changes after the completion of the programmatic changes are complete. The objective output of this stage is an optimum result combining programmatic changes with whatever manual intervention deemed necessary.

### ***Final Conversion & Production Integration***

The automated nature of this process, allows for the latest version of the source code to be brought in and run through the first three stages. It is also only at this stage that formal SCM policies and procedures need be implemented.

In many cases no conversion or change will take place but a recompile will be needed. Again the application map can be sued to good effect here. Simply building recompile lists based upon the converted source code and all related objects from the where sued information, will help ensure nothing is missed. It also means that simple CL's can be written to bulk recompile and incorporate any compilation strings to the compile commands.

## **Application Modernization**

Structural changes to an application can be a key part of a company's modernization strategy. Some of these structural changes are motivated by more strategic objectives such as agile development, reusable architecture and functional redesign. Other modernization projects are driven by more commercial demands such as internationalization.

### ***UNICODE Conversions***

An increasingly popular modernization requirement on IBM i is UNICODE conversions. The principal of a UNICODE conversion is largely the same as a field expansion project: changing the attributes of database and display fields and updating all affected logic in the programs. There are some differences in the process and requirements, but the same approach can generally be followed. Indeed the same programs used for field expansion can be enhanced to accommodate for UNICODE conversions without too much work involved.

Let's have a look at some simple examples of what could be changed programmatically with regards a UNICOE conversion. The first aspect is updating the fields in the files and displays. This sort of change is consistent wit the field expansion algorithms mentioned earlier in this article. Figure 9 shows how the field definition for the COMPANY field has been updated to a type "G" and the desired UNICODE CCSID specified in the function column for this field.

Line 1	Column 1	Replace
	.....A.....	T.Name+++++.....Functions+++++.....
000100	A	R CUSRCD
000101		
000200	XRSZEA*	COMPANY 34A TEXT('Company')
000300	XRSZEA	COMPANY 34G CCSID(13488)
000400	XRSZEA	TEXT('Company')

Figure 9

Figure 10 shows how the H-Spec of an RPGLE program has been automatically updated with the requisite CCSID code. In this instance the CCSID H-specification keyword is used to set the default UCS-2 CCSID for RPGLE modules and programs. These defaults are used for literals, compile-time data, program-described input and output fields, and data definitions that do not have the CCSID keyword coded.

Line 18	Column 1	Replace
	.....1.....	+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....
000160	h	CCSID(*UCS2 : 1200 )

Figure 10

Figure 11 shows how by using a fairly straight-forward algorithm your automated program can intervene in your C-specifications and automatically convert statements to include the %UCS built-in function where required. In this example, as with the field expansion samples, old lines have been commented out to show how the programmatically created new line has been changed.

Line 251	Column 1	Replace
	.....1.....	+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8.....
002480	XRSZEC*	Eval PrintTex1 = %Trim(PrintTex1) + ' ' + Cents1 +
002490	XRSZEC	Eval Printtex1_c = %Trim(%ucs2 (PRINTTEX1)) +
002500	XRSZEC	%ucs2(' ') + %ucs2 (CENT1A) +
002510	XRSZEC*	Eval '/100 ' + %Trim(Z5PRT1)
002520	XRSZEC	%ucs2 ('/100 ') + %Trim(Z5PRT1)
002530	XRSZEC	Eval Printtex1 = Printtex1_c
002531	*	=====
002532		
002540	C	Else
002550	XRSZEC*	Eval PrintText = %Trim(PrintText) + ' ' + %Trim(
002560	XRSZEC	Eval Printtext_c = %Trim(%ucs2 (PRINTTEXT)) +
002570	XRSZEC	%ucs2(' ') + %Trim(
002580	XRSZEC*	Z5PRT1) + ' ' + CentsA + '/100 ' + %Trim(
002590	XRSZEC	Z5PRT1) + %ucs2(' ') +
002600	XRSZEC	%ucs2 (CENTSA) + %ucs2 ('/100 ') + %Trim(
002610	C	Z5PRT2)
002620	XRSZEC	Eval Printtext = Printtext_c
002630	C	End

Figure 11

There are two important points to make with regards UNICODE conversions:

- Unicode data is not supported in non-ILE versions of RPG. If you wish to implement Unicode support in non-ILE RPG programs, you must convert them to RPGIV (ILE RPG) source code and recompile beforehand.
- IBM are actively enhancing Unicode support on the IBM i via the release of Program Temporary Fixes (PTFs) both for the DB2 database and the ILE RPG compiler.

### Externalizing Database I/O

Another increasing trend in the IBM i Application space is the need to separate I/O logic out from legacy programs. One primary motivation for this is the need for making significant changes to database architecture without interrupting proven process and business logic.

Another business driver for this is from companies replacing legacy custom software with off the shelf applications, but wanting to keep certain core functions ruing as is, at least for a period. In this scenario mapping to the replacements database architecture can be carried out without interruption to critical legacy functions, provided of course that the database I/O has been externalized from the legacy programs first.

The algorithms used by programs that would automatically make such a change would be different from a field expansion process, but once again the core asset here would be the application map for the initial analysis. These reengineering programs can then be designed to identify and convert all source code instructions needed to transfer file I-O into external modules, giving identical functionality.

Thus the following code in figure 12 shows how an I/O statement is replaced with a procedure:

```
c      xwbccd      chain(e)  custsr      31

BECOMES

c      eval      wCUSTSR = $CUSTS_chai('E':$inx:XWBCCD)
```

Figure 12

Another requirement of the reengineering programs is to automatically build fully functional I-O modules, which can then be adapted to a radically changed database, with no impact on the reengineered RPG code – the module returns a buffer identical to the original file. So if one wanted to switch to a completely new customer file, one could simply change the I-O module code (as sown in Figure 13) and the hundreds of RPG programs using the CUSTS file require no source changes whatsoever!

```
chain(e) %kds(kdCUSTS:1) CUSTS ;

BECOMES

chain(e) %kds(kdCUSTS:1) NewCusFile ;
cusname = newCusName ;
cusadr1 = newAdress1 ;
etc.
```

Figure 13

### ***Refactoring Monolithic Code into Services***

Another important way of using programs to update programs is in the area of building services from legacy application code.

There are many articles and guidelines from leading thinkers such as Jon Paris, Susan Gantner and various others on the subject of using sub-procedures over sub-routines. This fine for new applications, but most interactive legacy programs are written in a monolithic style. This can severely limit long-term modernization opportunities, not to mention adding significant stress and complexity to ongoing maintenance and development tasks in general.

By advancing the algorithms of replacement and code regeneration described in all three areas above, it is possible to refactor monolithic programs by externalizing the subroutines into procedures automatically. Breaking up the program into two components like this makes the rewrite of the user interface layer easier, but simultaneously makes available to externalized sub-procedures as callable services. This is a great way to start on a staged application reengineering while realizing immediate benefit.

Figure 14 below shows two subroutines “VALID1” and “VALID2” being invoked in a monolithic legacy program “WWCSUSTS”.

```

Line 381      Column 36      Replace
.....CLONO1Factor1++++++Opcode (E) +Factor2++++++Result++++++Len++D+HiLoEq....
037200
037300      C      *IN04      IFEQ      '1'
037400      C      *IN08      OREQ      '1'
037500      C      *IN09      OREQ      '1'
037600      C      ITER
037700      C      ENDIF
037800
037900      C      select
038000      C      when      pageno = 1
038100      C      EXSR      VALID1
038200      C      when      pageno = 2
038300      C      EXSR      VALID2
038400      C      ends1

```

Figure 14

A “rework” program was written using similar logic to the field expansion and I/O externalization programs above, to create a new “Business Logic” module that would contain procedures created from all of the legacy subroutines in the original programs. Figure 15 shows the definition for the “wwcustsvalid1” in this new ILE module “WWCUSTSB”

```

Line 94      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords++++++
007300      I*OR e c o r d   S t r u c t u r e s
007400      I*****
007500      IRCUSF
007600      I      DSDCDE      XSDCDE
007700
007800      P*****
007900      P wwcustsvalid1...
008000      P      b      export
008100
008200      D wwcustsvalid1...
008300      D      pi
008400      D  plactdsp      like (actdsp)
008500      D  pidname      like (dname)
008600      D  pidsdcde      like (dsdcde)

```

Figure 15

The “rework” program also updated the original program to use the service program “WWCUSTSB” invoking the appropriate procedure as opposed to subroutine, and passing the correct parameters. The “rework” program also created the necessary prototypes in the updated “WWCUSTS” program as is shown in Figure 16 below.

Line	Column	36	Replace
	.....	CLON01Factor1++++++	Opcode (E) +Extended-factor2++++++
031100			
031200	C	*IN04	IFEQ '1'
031300	C	*IN08	OREQ '1'
031400	C	*IN09	OREQ '1'
031500	C		ITER
031600	C		ENDIF
031700			
031800	C		select
031900	C		when pageno = 1
032000	C		eval zin = *in
032100	C		callp(e) wwcustsvalid1(actdsp:dname:dscdce:errmsg:
032200	C		msgid:person:pname:xwbccd:xwbncd:xwg4tx:
032300	C		xwkhtx:zin)
032400	C		eval *in = zin
032500	C		when pageno = 2
032600	C		eval zin = *in
032700	C		callp(e) wwcustsvalid2(actdsp:cusno:errmsg:msgid:
032800	C		xwgiva:xwidv0:zin)
032900	C		eval *in = zin
033000	C		endsl

Figure 16

## In Conclusion

Using programs to update programs is not a new or even unusual technique. Combined with a very detailed application map of an entire system, this approach to system engineering can help solve the problem of modernizing and enhancing large and complex legacy applications using limited resources in shorter time frames. For many companies it has saved millions of dollars in development costs, and has also provided a means to bring legacy application code into the world of modern architectures and techniques.

In the next article in the series I will be looking at how to extract design model assts from legacy systems. I will be covering areas such relational data models, business rules and how these make legacy applications relevant in a modern context.

## Further Reading

Here are two very useful IBM Redbook publications that cover almost all of the subjects discussed in this article:

- Modernizing IBM eServer iSeries Application Data Access - A Roadmap Cornerstone**  
<http://www.redbooks.ibm.com/redbooks/SG246393/wwhelp/wwhimpl/java/html/wwhelp.htm>
- Modernizing and Improving the Maintainability of RPG Applications Using X-Analysis Version 5.6**  
<http://www.redbooks.ibm.com/redpieces/abstracts/redp4046.html>