

IBM i the present and the future.

There is a great deal of uncertainty about the future of IBM i including mixed messages from IBM following the consolidation of Systems i and p into the new Power Systems brand. While IBM may have weakened the public perception of the brand, the hardware and software still delivers what it always has – rock solid reliability and dependable applications.

So while the world has forgotten about the ‘AS/400’ and green screens, there are still huge code bases written over the past 10-40 years (RPG celebrated its 40th birthday in 2009) powering corporations of all sizes. The investment this technology represents cannot simply be replaced with packaged ERP software or quickly re-written in a new language or framework. The reason that these systems are still running is a testament to the success of the platform and its development ecosystem in general. This is a point that seems lost on the wider development and business community. There is simply no other system that supports applications written over 40 years ago in its original form without source code modification.

The challenge for today’s IBM i sites is how to retain sufficient development resources to maintain and develop the applications as the number of active RPG people diminish through promotions, retirement and natural attrition. There has to be a way of enabling new people to understand quickly and accurately the complexities and subtleties of these sometimes vast systems and give them the confidence to make changes and extend these systems, even though they will never have developed anything like them themselves. This article is the first in a series that describes this literally growing challenge in some detail and, how new technologies and concepts are evolving to provide solutions and bolster IBM i development.

A typical application on IBM i could be anything from a few thousand to many millions of lines of code, with all of the complexity, design inconsistencies, languages, syntaxes and semantics that go with years of ongoing development. Mission critical applications consist of a great many physical files or tables, and programs. The inter-dependencies of program-to-file and file-to-program alone can easily reach hundreds of thousands. We are not talking about the abstracted or esoteric nature of individual pieces of technology here, but entire business systems.

As with any successful management system, information about your systems is the key. The level of detail and availability of this information is another critical factor, which has already been proven in business by the success of ERP and business systems in general. The requirement is not a new one but is becoming more universal as systems continue to grow and mature. A key issue is how to manage the cost and risk of maintaining and modernizing these systems.

This first article discusses how application mapping has become a core solution to the problem. Application mapping means analyzing and extracting a database of information about the resources that constitute a business application system.

Making informed decisions

By mapping an entire application, a fundamental base line of information is made available for all sorts of metrics and analysis. Counting objects and source lines is generally the most common practice used for obtaining system-wide metrics. Many companies carry out software project estimations and budgeting using only this type of information. The level of experience and technical knowledge of a manager and his staff might help these numbers to some degree, but more often than not, it’s mostly guesswork.

A slightly more advanced approach used with RPG or COBOL applications, is to dig deeper into the application and count design elements within the programs themselves. These elements include:

- files,
- displays,
- sub-files,
- source lines,
- sub-routines
- called programs
- calling programs.

By using a simple formula to allocate significance to the count of an element, you can then categorise programs by their respective counts into low, medium and high complexities. This type of matrix based assessment is still fairly crude, but adds enough detail to make estimations and budgeting much more accurate without too much additional effort.

1	Application Metrics								
2									
3	Complexity Level	Units	Attribute	Source Lines	Files	Device Files	Called Progra	Calling Progra	Copybooks
4	Grand Total	137		15829	173	38	117	104	0
5	Batch Programs	58		2464	75	0	38	60	0
6	Low Total	58		2464	75	0	38	60	0
7	Average Total	0		0	0	0	0	0	0
8	High Total	0		0	0	0	0	0	0
9	Interactive Programs	38		9543	98	38	79	44	0
10	Low Total	38		9543	98	38	79	44	0
11	Average Total	0		0	0	0	0	0	0
12	High Total	0		0	0	0	0	0	0
13	Others	41		3822	0	0	0	0	0
14	Display Files	39		3554	0	0	0	0	0
15	Printer Files	2		268	0	0	0	0	0
16	Copybooks	0		0	0	0	0	0	0

Figure 1 – Simple Design Complexity Matrix [fig1.png]

Another common practice is to take small representative samples such as those selected for a POC (proof of concept), do project estimations, and then extrapolate this in a simplistic linear way across the entire system or for an entire project. This naturally relies upon the assumption that design, style, and syntax for the entire application are consistent with the samples used for the POC. The reality is that samples are most often selected for POC’s based on functionality rather than complexity. Sometimes the opposite is true whereby the most complex example is selected on the basis “if it works for that it’ll work for anything”.

Calculations that use comprehensive and accurate metrics data for an entire application, versus data from a sample, will exponentially improve the reliability of time and cost estimation.

Risk is not entirely removed, but plans, estimates and budgets can be more accurately quantified, audited, and even reused to measure performance of a project or process.

There are more advanced techniques to measure application complexity that are worth mentioning. If used over an application map, a number of very useful statistics and metrics can be calculated including detailed testing requirements and a “maintainability index” for entire systems or parts thereof.

Building Application Maps

The cost of ownership of these large complex IBM i applications increases and maintenance becomes more risky, as application knowledge is lost and not replaced.

Display Program References (DSPPGMREF) provides information about how a program object relates to other objects in the system.

```

Display Spooled File
File . . . . . : QPDSPPGM          Page/Line  1/1
Control . . . . :                   Columns    1 - 78
Find . . . . . :
*...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
9/13/09          Display Program References
DSPPGMREF Command Input
Program . . . . . :                   WWCUSTS
Library . . . . . :                   CDEMO
Output . . . . . :                   *
Object types . . . . . :              *PGM
Program . . . . . :                   WWCUSTS
Library . . . . . :                   CDEMO
Text 'description' . . . . . :        Work with Customers
Number of objects referenced . . . . . : 21
Object . . . . . :                   CUSFSEL
Library . . . . . :                   *LIBL
Object type . . . . . :              *PGM
Object . . . . . :                   CUSGRSEL
Library . . . . . :                   *LIBL
Object type . . . . . :              *PGM
F3=Exit  F12=Cancel  F19=Left  F20=Right  F24=More keys  More...

```

Figure 2 – DSPPGMREF screen output [fig2.png]

This information is very useful in determining how a program relates to other objects. It is possible to extract this information and store it in a file, and carry out searches on this file during analysis work.

	A	B	C	D	E
1	WHLIB	WHPNAM	WHTEXT	WHF	WHFNAM
494	XAN4CDEM	WWCUSTS	Work with Customers	21	CUSFSEL
495	XAN4CDEM	WWCUSTS	Work with Customers	21	CUSGRSEL
496	XAN4CDEM	WWCUSTS	Work with Customers	21	CUSTOMT1
497	XAN4CDEM	WWCUSTS	Work with Customers	21	DISTSSEL
498	XAN4CDEM	WWCUSTS	Work with Customers	21	RTNMSGTEXT
499	XAN4CDEM	WWCUSTS	Work with Customers	21	SLMENSEL
500	XAN4CDEM	WWCUSTS	Work with Customers	21	WWCONHDR
501	XAN4CDEM	WWCUSTS	Work with Customers	21	WWTRNHST
502	XAN4CDEM	WWCUSTS	Work with Customers	21	XBCCLMSG
503	XAN4CDEM	WWCUSTS	Work with Customers	21	QRNXIE
504	XAN4CDEM	WWCUSTS	Work with Customers	21	QRNXIO
505	XAN4CDEM	WWCUSTS	Work with Customers	21	QRNXUTIL
506	XAN4CDEM	WWCUSTS	Work with Customers	21	QLEAWI
507	XAN4CDEM	WWCUSTS	Work with Customers	21	CONHDL1
508	XAN4CDEM	WWCUSTS	Work with Customers	21	CUSFL3
509	XAN4CDEM	WWCUSTS	Work with Customers	21	CUSGRP
510	XAN4CDEM	WWCUSTS	Work with Customers	21	CUSTS
511	XAN4CDEM	WWCUSTS	Work with Customers	21	DISTS
512	XAN4CDEM	WWCUSTS	Work with Customers	21	SLMEN
513	XAN4CDEM	WWCUSTS	Work with Customers	21	TRNHSTL3

Figure 3 - DSPPGMREF output to spreadsheet [fig3.png]

[Sidebar]

Calculating Complexity

Halstead Volume

Halstead complexity metrics were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module to measure a program module's complexity directly from source code. Among the earliest software metrics, they are strong indicators of code complexity, and because they are applied to code. This metric is most often used as a maintenance metric. They are one of the oldest measures of program complexity. --

http://en.wikipedia.org/wiki/Halstead_complexity_measures

Cyclomatic Complexity

the number of recommended tests for software. Second, it is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable. Cyclomatic complexity is based entirely on the structure of software's control flow graph. --

http://en.wikipedia.org/wiki/Cyclomatic_complexity

A much more efficient way of showing the same information, is to show it graphically. Additional information such as the directional flow of data can be included easily and understood and just as easily added to diagrams. Systems design and architecture is best served using diagrams. Color coding within these constructs is also important as it helps assimilate structure and logically significant information more quickly. A good example of this is showing where updates take place using the color red (see Figure 4 below).

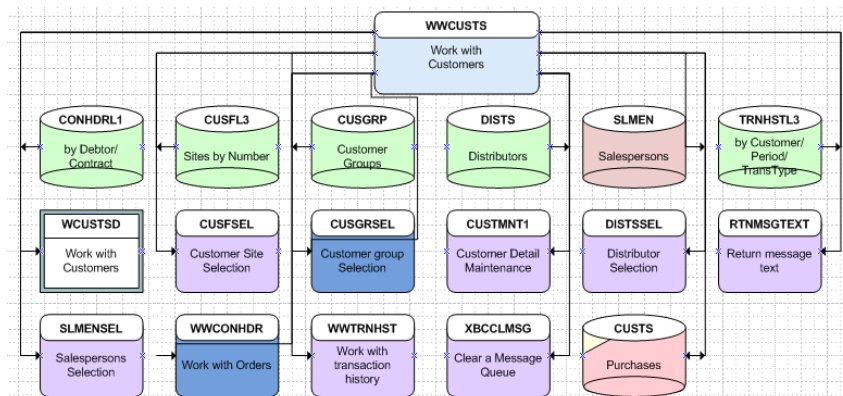


Figure 4 – Visualizing Program References [fig4.png]

Embedding other important textual information such as an object text's into or along with diagrams is another way of presenting information effectively and efficiently. In Figure 4 we see how graphical and textual information are combined to provide rich information about the program references plus arrows being used to show the flow of data between the program and the other objects.

Tom Demarco, the inventor of the data flow diagram concept stated that what is critical is the flow of data through a system. Application mapping information can be extended (Figure 5) to simultaneously include details about individual variables associated with each of the referenced objects. The method used to extract this level of precise variable detail is to scan the source code of the programs, and establish which entry parameters are used in the case of a program-to-program relationship.

In a program-to-file relationship, the job is somewhat more tedious as one has to look for instances where database fields and corresponding variables are used throughout the entire program. It is also useful to see where individual variables are updated as opposed being used just as input. The diagram now presents a very rich set of information to the user in a simple and intuitive way. The amount of work to extract and present this level of detail can quickly become prohibitive, and so better suited to a tools based approach than manual extraction.

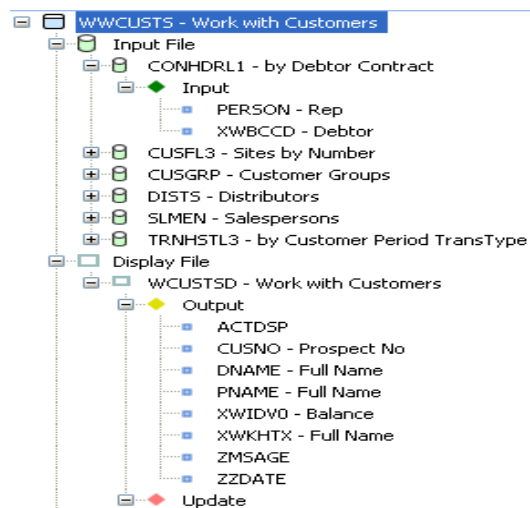


Figure 5 – Variables/Parameter Details [fig5.png]

The diagram in Figure 6 shows program centric diagram. The same diagram where the file is the central object being referenced is also very useful in understanding and analysing complex applications. The same diagrammatic concepts can be used such as color coding for updates, arrows for data flow, and detailed variables simultaneously being displayed. By using the same diagram types for different types of objects in this way, the same skills and methods can be reused to twice the effectiveness. Figure 5 shows how additional information such as related logical files (displayed as database shapes) can be added and easily recognized by using different shapes to depict different object types.

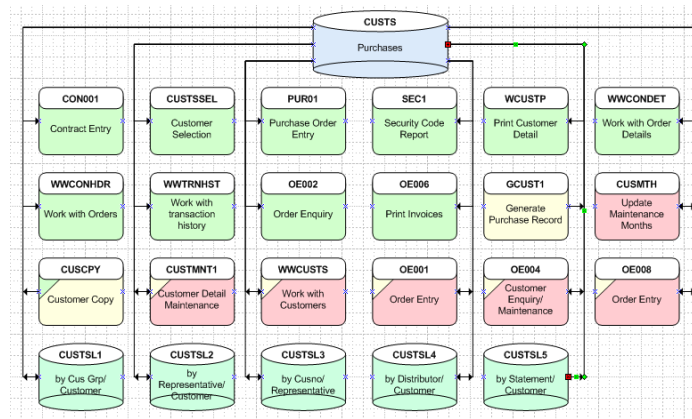


Figure 6 – File Centric Object References [fig6.png]

Functional Organization of an Application

Single level information about an RPG or COBOL program is obviously not enough to understand a business systems design. One needs to be able to follow the logical flow downwards through the application.

The DSPPGMREF data can be used to do this too. If we start at program A and see it calls program B, we can then look at the DSPPGMREF information for program B and so on. Additionally we can deduce precisely in this structure where and how data, print, and display files are being used in the call stack, which is very useful for testing and finding bugs that produce erroneous data.

For large, complicated systems this can be a slow and tedious process if done manually using the display output of the DSPPGMREF. By extracting all programs' DSPPGMREF information out to a single file, this file can be recursively queried to follow the calls down successive levels starting at a given program. This can then show the entire call stack or structure chart for all levels starting at a given program or entry point.

A given programs call stack or call structure can be represented much more effectively diagrammatically than with any textual description alone. Quite often these call stacks may go down as many as 15 levels from a single starting point. It is an important requirement therefore to be able to display or hide details according to the information required at the time, along with search facilities built-in to the diagrams.

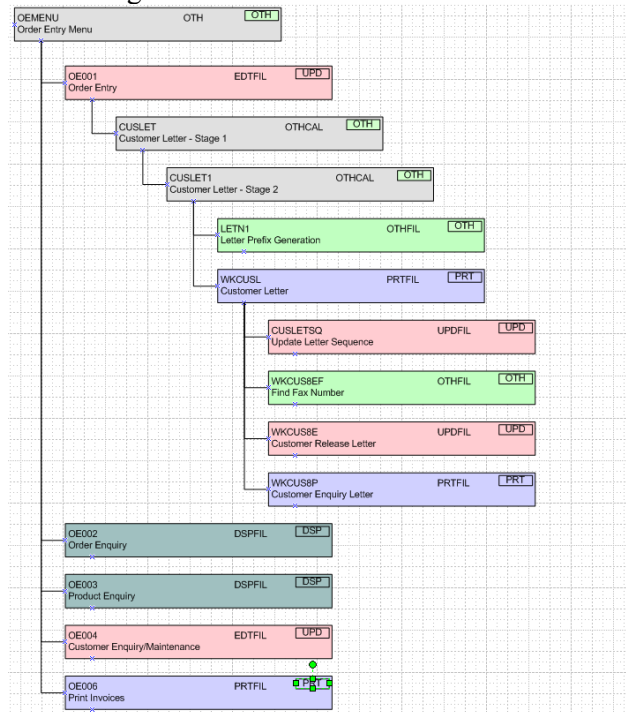


Figure 7 – Diagram showing program call structure [fig7.png]

As with other diagrams, color coding plays an important role in classifying objects in the stack by their general use, such as update, display, input only and so on. Figure 7 shows the structure of a program as seen graphically. Additional information such as what data files, displays and data areas are used by each object can be added to enrich the information provided.

This diagram alone however does not tell us where we are in relation to the overall hierarchal structure of the application. We do not know if the program is an entry point into the system or is buried in the lower levels of the application.

To better understand an entire system therefore, objects need to be organised into functional groups or areas. This can be achieved by using naming conventions, provided that they exists, and are consistent across the application. The entry points into the application need to be established. Sometimes a user menu system is useful for this, but is not necessarily complete or concise enough. One way to establish what programs are potential entry points is to determine each program's call index. If a program is not called anywhere, but does call other programs, it can essentially be classed as entry point into the system. if a program is called and in turn if it calls other programs itself, it is not an entry point.

A functional area can be mapped by selecting an entry point (or a group of them) and then using the underlying application map to include all objects (everything including programs, files, displays) in the call stack. Figure 8 shows a diagram of a series of entry points and their relative call stacks grouped as a functional area.

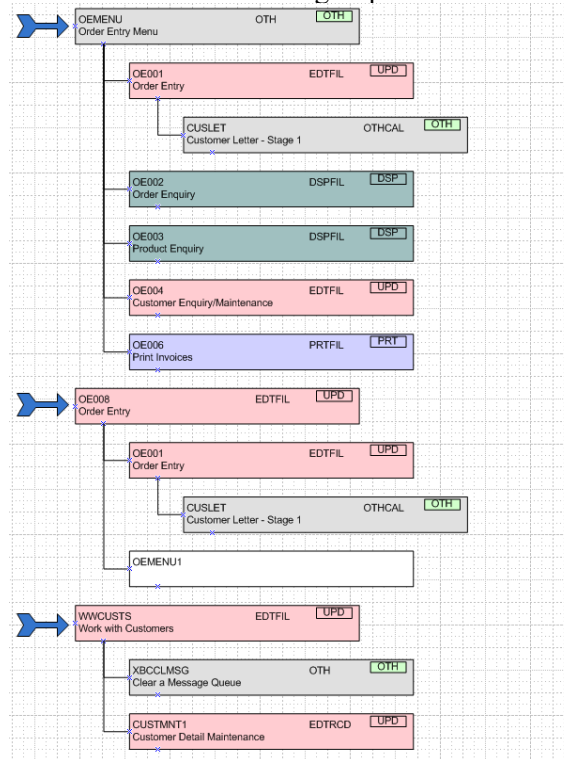


Figure 8 – Functional Application Area [fig8.png]

To more accurately describe an entire system’s architecture, functional application areas might need to be grouped into other functional application areas. These hierarchal application areas can then be diagrammed, showing how they interrelate with each other. This interrelation can be hierarchal but also programmatically, because some objects might be found in more than one application area simultaneously.

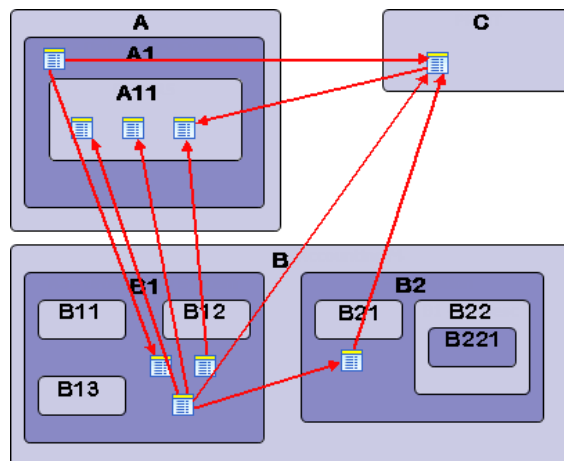


Figure 9 – High Level Functional Relationships [fig9.png]

Figure 9 is a diagram showing how application areas interrelate. For the sake of clarity only those programmatic interrelations from entry level objects have been included in the diagram. The diagrams show how the accounting Main application area has other (B, A2..) application areas embedded in it. The red lines show the programmatic links between objects within the application. In this example this level of interrelation has been limited to programmatic links between entry point programs, and programs they call in other application areas. This is a very good way of mapping business functional areas to application architecture in a simple diagram.

Logical subdivisions of an entire application are also being employed in other areas of application management. Some of these include:

- Clear and concise allocation of responsibility for maintenance/support of a set of objects.
- Integration with Source change management tools for check in and check-out processes during development
- Production of user documentation for support, training and testing staff

Database Mapping

An IBM i business application is primarily an application written over a relational database. Therefore, no map of an enterprise application would be complete without the database architecture explicitly specified. Not just the physical specifications and attributes, but the logical or relational constraints too.

With the possible exception of CA:2E systems virtually all RPG or COBOL applications running on IBM i have no explicit relational data model or schema defined. This means that millions of lines of RPG or COBOL code must be read in order to recover an explicit version of the relational model. What one is searching for is what keys constitute these links or relationships between physical files or tables in the database.

The first task is to produce a key-map of all the primary keys and fields for all physical files, tables, logical files, access paths and views in the database. By using a simple algorithm and looking at the DDS or DDL one can often determine if foreign key relationships exists between files. Figure 10 shows a diagram of this simple algorithm using the database definitions themselves.

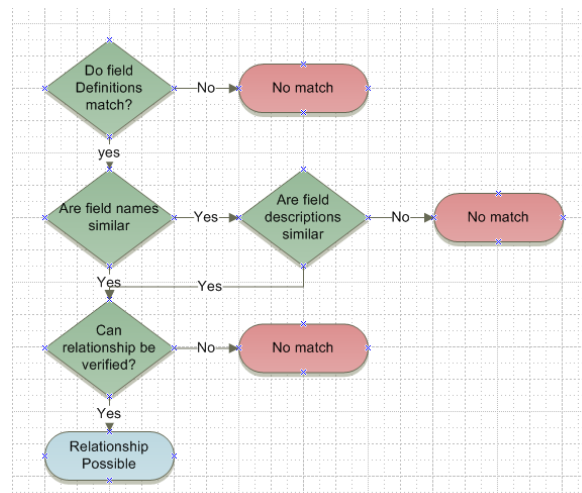


Figure 10 – Establishing Foreign Key Relationships [fig10.png]

A more advanced and comprehensive approach for determining foreign key relationships is to analyse the program source code for the system. If we look at the source code of a program and see that more than one file/table is used, there is a possibility that these files are related by foreign key constraints. By finding instances in the program where one of the files is accessed for any reason, and determining the keys used to do so, we can then trace these variables back through the code to keys in another file in the program. If at least one of the key fields match in attribute and size, with the other file, and is part of the unique identifier of the file then we have a very strong likelihood that there is a relationship between these two files. By then looking at the data using these key matches we can test for the truth of the relationship. By cycling through all the files in the system one by one and testing for these matches with each and every other file, we can establish all the relationships.

This task is complicated generally by the fact that the same field in different files will usually have a different mnemonic name. When analyzing the program source one will have to deal with data structures, renames, prefixes, and multiple variables. By having the program variable mapping information at your fingertips before hand, the analysis process will be a lot quicker. The vast majority of this type of repetitive but structured analysis can be handled programmatically, and thus enable the task to be completed in a few hours rather than several months. Such automation naturally allows for keeping the relational model current at all times without huge overhead on resources.

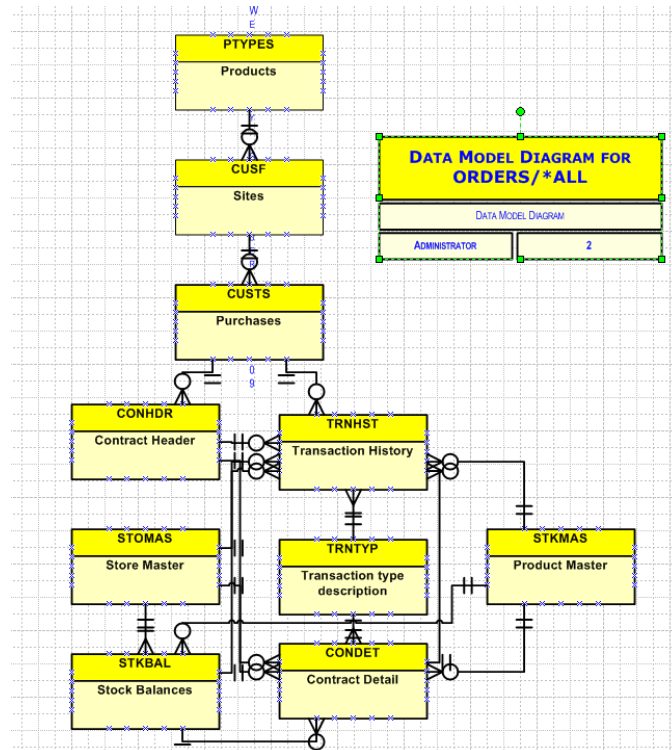


Figure 11 – Mapping Database File Relationships [fig11.png]

Once explicitly defined, the relational model or architecture of the database can be reused in a number of scenarios including:

- Understanding application architecture
- Data quality referential integrity testing
- Test data extraction
- Test data scrambling and aging
- Building BI applications & Data Warehouses
- Data mapping for system migrations
- Build object relational maps for modernization

Database access in all modern languages today is primarily driven by embedded SQL. IBM i legacy databases are typified by transaction based table design with many columns and foreign key joins. This makes the task of writing SQL statements much more difficult and error prone unless the design of the database is clearly understood. It also creates an environment where it is relatively easy for inexperienced developers or users to write I/O routines or reports that have an extremely negative performance impact. One way to combat this is to provide detailed design information about the database being accessed. Figure 11 shows a typical entity relationship diagram, and this can be accompanied with the underlying foreign keys details as displayed in Figure 12.

Rel No.	Dependent File	Relation Type	Parent File	Dependent Fields	Parent Fields
1	Transaction Histo	ACCESSES	Contract Detail	Contract, Product	Contract, Product
2	Transaction Histo	ACCESSES	Contract Header	Contract	Contract
3	Transaction Histo	ACCESSES	Customer Groups	DGrp	CusGrp
4	Transaction Histo	ACCESSES	Purchases	Debtor	Customer
5	Transaction Histo	ACCESSES	Salespersons	Rep	Person
6	Transaction Histo	ACCESSES	Stock Balances	Product, Store	Product, Store
7	Transaction Histo	ACCESSES	Product Master	Product	Product
8	Transaction Histo	ACCESSES	Store Master	Store	Store
9	Contract Detail	OWNED BY	Contract Header	Contract	Contract
10	Contract Detail	ACCESSES	Stock Balances	Product, Store	Product, Store
11	Contract Detail	ACCESSES	Product Master	Product	Product
12	Contract Detail	ACCESSES	Store Master	Store	Store
13	Contract Detail	REFERS TO	Transaction type	Trn Hst Trn Type	Transaction type
14	Purchases	ACCESSES	Sites	Prospect No	Cus. No.
15	Purchases	ACCESSES	Customer Groups	CusGrp	CusGrp
16	Purchases	ACCESSES	Distributors	Distributor	Code

Figure 12 – Foreign key details [fig12.png]

Another more generic approach to ensuring integrity of the database, productivity for modern technology developers, and limiting negative I/O performance impacts, is to build a framework of I/O modules as stored procedures. The explicitly defined data model is a key source of information, which will greatly simplify building of such a framework, and can even be used to automate the generation of the framework itself.

It also worth mentioning that products like IBM’s DB2 Web Query can become exponentially more useful and productive if the meta-data layer is properly implemented. The derived data model can be used to build this data instantly for the entire system

Hard-Coding Application knowledge

The output of DSPPGMREF is a great starting point the type of mapping described so far. In order to produce such details and abstractions, the source code of the application would need to be read and analysed.

From a design perspective, application software is made up of discrete layers or levels of detail. In an IBM i application for example, libraries contain programs, physical files, logical files data areas, commands and many more object types. While programs might contain file specs, variables, sub-routines, procedures, display definitions, arrays and various other language constructs. Data files have fields and text descriptions and keys and other attributes. Having an inventory of all these elements is useful, but only in a very limited way from a management perspective. What’s needed is context. For example mapping what files and displays are specified in a program helps understand at an object level the impact of change. This rudimentary mapping provided by most program comprehension tools, is limited in its usefulness as it still only provides information at single level.

Mapping all levels of detail and how they interrelate with all other elements at all levels is the ultimate objective. The only way to achieve this is to read the source code itself line-by-line, and infer all relationships implicit in each statement or specification. Naturally the mapping process must allow for variants of RPG, COBOL and CL going back 20 years, if it is to be useful for the vast number of companies who have code written 20 years ago in their mix. Relatively few humans have such knowledge or skill and, as mentioned previously, could never keep up with the work load required for even the most modest of IBM i applications. Computer programs can be “taught” such knowledge and retain it permanently. They can also be reused as often as is necessary to keep abreast of any code changes that take place.

Table by Google			
Define scopes and calculations			
(All)	(All)	(All)	(All)
Object	Text		Library
CUSCPY	Customer Copy	File Read By Program	XAN4CEM
CUSFL1	Sites by Name	File Updated By Program	XAN4CEM
CUSFL2	Sites by Status	File Updated/Written To By Program	XAN4CEM
CUSFL3	Sites by Number	Logical File	XAN4CEM
CUSFL5	Sites by Dist.& Status	Usage	XAN4CEM
CUSFL6	Sites By Dist.& Name	Logical File	XAN4CEM
CUSFL7	Sites by Last Cnt Date	Logical File	XAN4CEM
CUSFL8	Sites by Next Cnt Date	Logical File	XAN4CEM
CUSFL9	Sites by Fax No.	Logical File	XAN4CEM
CUSFLA	Sites by Product - renamed from cusfls for testing	Logical File	XAN4CEM
CUSFLB	Sites by Orig List	Logical File	XAN4CEM
CUSFLC	Sites by Salesperson	Logical File	XAN4CEM
CUSFLD	Sites by Validator	Logical File	XAN4CEM
CUSFLE	Sites by Organisation	Logical File	XAN4CEM
CUSFMAINT	Customer Site Maintenance	File Updated By Program	XAN4CEM
CUSFMOLD	Customer Site Maintenance	File Updated By Program	XAN4CEM

Figure 13 – Pre-Building DSPPGMREF output [fig13.png]

Pre-building the application map and storing it in an open and accessible format such as a spreadsheet in Google Docs, is also an important aspect of the overall usefulness of such information. Figure 13 shows the output of a DSPGMREF uploaded into a Google Docs spreadsheet and being filtered. Having the map available provides for any number of complex system-wide, abstractions or inquiries at acceptable speeds. Imagine that every time you zoomed in on Google Earth waiting for satellites to reposition and retake their pictures.

For a complete and accurate application map, one has to follow the trail of inferred references described in the programs themselves. This is obviously a labour-intensive task made all the more difficult by common coding practices such as:

- Overriding the database field name in a CL program,
- Prefixing fields from a file being used in an RPG program,

- Moving values from database fields into program variables before passing them as parameters to called programs,
- Changing key field names between different database files.
- Passing the name of the program to be called as a parameter to a generic calling program rather than making a direct call.

If the pre-built application map includes all of these inferred logical references, then measurement of impact can be complete, and more importantly; instant. It also means that higher-level analysis of rules and model type designs is made easier by virtue of the easy availability of variable and object level mapping.

In Summary

Application mapping provides a new way to manage and modern complex business applications. It is also a way facilitate collaboration between modern and legacy developers. Think about what computerised mapping has done for navigational and guidance systems in our day to day life and travel. It makes many new things possible all the time. Application mapping provides a very strong platform for a number of additional benefits and technologies that will continue to evolve for many years. There will be more about these subjects in later articles in this series.