New Tricks with Old Tools

Peter Spark
Survey Research Center, University of Michigan

Blaise has a wealth of utilities, tools and features packaged within its system, and multiple ways of extracting and using information from datamodels, databases, modelib files; e.g., datamodel properties and so forth. Such tools as Manipula/Maniplus, Cameleon, Registry Editor, Delta, Data Centre, BCP, "watch" window, and more allow authors and managers to get the in-depth information they need to effectively perform their job.

However, the Blaise environment allows even richer capabilities beyond typical methods to produce a quality survey. Such features as alien routers and procedures (using BCP and/or Manipula), datamodel and data XML, Modelib, native and BOI databases, menu files, etcetera, give even more control. With such information, methods can be used to analyze current programming and provide needed details that help make decisions in the hectic survey programming environment.

This paper examines the capabilities of existing tools in the Blaise suite and explores practical new ways to make use of them, as well as ways of extracting and using information in ways that are relevant to both authors and managers in the Blaise 4.8 environment. Crosswalks, counts, finding unused code and questions, complexity calculations, and more are the sort of tricks that can be added to the survey tool bag that reduces errors and decreases turn-around time.

1. Blaise Datamodel Information

Before analysis of Blaise datamodels can begin, it is useful to review the list of current tools and their practical applications to extract datamodel information. The Blaise datamodel, for the most part, is the most useful source of information. It has the advantage of being syntactically correct according to the Blaise parser. However, certain information is lost, such as comments and source code formatting, and other information is more is difficult to retrieve, such as the code within procedures, and rules text.

The source code more easily provides all this other details, but requires a custom parser that mimics the behavior of the Blaise parser. In many cases the source code is not available, or the proper version does not prepare to a datamodel, or some other vital file could be missing. This paper concentrates mainly on existing utilities that read the datamodel, although it is possible to read the source code with custom parsers written in a high level language or Manipula.

The goal is not to reinvent the existing utilities but to take advantage of them. Otherwise it would be just a matter of writing a new utility using the BCP and a high level language to extract all the needed information. The disadvantage of such an approach is the need for in-depth programming skills. Intermediate methods, such as Manipula and Cameleon, may also require programming skills, while other utilities provided by Blaise just need a skilled user.

The table below shows a few of methods that can be used to retrieve data and metadata from the Blaise datamodel.

Method	Data	Meta	Notes
BCP	Full case data, comments	Full meta information	Original comments and formatting lost, requires advanced programming skills, essentially creating a new utility.
Blaise Data Centre	Filtered/full case data	Can export data selected on fields/row filters to text, BDB,	Can also get Blaise-generated source code for the dictionary.[1]

Method	Data	Meta	Notes
		or XML	
Blaise Registry Editor	Related to settings for various files	Settings information and some metadata (i.e., BXI: All defined types, INI format, version and copyright info)	Supports reading BOI, BTR, BIC, BIS, BFS, DBI, BIN, IDM, BMF, MSX, BXML, BCI file types. Use File – Export Registry to get a BRF (INI format).
Cameleon	No data	Defined fields, blocks	May require additional routines to create a report. TechDesc.cif - Gives technical information about the datamodel. Such as number and length of fields, number of parameters for each block.
Datamodel properties	No data	All defined types, INI format	Use – Types – Export. Can be used as part of an input for other utilities. Does not include codes, ranges
Delta	No data	Partial metadata (fields, rules, used types, calls) Use special stylesheet to export.	Need to rename html files to xml, and change utf-16 to utf-8. May be a Delta bug to always produce separate files. XSLT could merge all these files together with document().
Manipula	Full case data, comments	Partial meta (defined and used fields, blocks, used types)	Not a "utility" but provides an easier-to-use method of retrieving both data and meta outside of the BCP. This can also be used to read Blaise source code and parse for information.
XSD Schema Wizard	No data	Partial metadata (defined fields, used types, blocks)	Matches XML from Blaise Data Centre.

[1]Note: escaped quotes (two double quotes to make one double quote upon export) within question text are not handled correctly – they are not escaped in the generated source code (4.8.4.1737). It can be manually fixed, and once the datamodel is prepared it can be used for the XML export.

1.1 BCP

The Blaise Component Pack is the API to the Blaise datamodel and database. It gives full access to the Meta information with (Field) and without (FieldDef) a connection to the data. It is most useful in building new utilities and is the core of the Blaise system, but by itself cannot be considered a utility. Its flexibility and inherent richness also mean that it is technically much more challenging to program.

1.2 Blaise Data Centre

The Date Centre is excellent for selecting data records and columns from Blaise data, filtering the results, and exporting the information in a variety of formats. It can create a new datamodel based upon the selected fields, export the data in XML, and create BOI files to the data. It can be thought of as a Blaise to Data extraction tool.

The XML file can be used as an input when looking at the data and Meta information, but it is not very rich in details, and so would have limited use for certain applications. The data is only stored per element if there's something to store. That is, skipped questions don't appear. The data is also in the order of the fields defined in the datamodel. Essentially, this is the Blaise to XML export run by the Data Centre interface.

1.3 Blaise Registry Editor

The Editor is a very handy utility that examines a variety of formats:

BXI	Blaise Extended Meta (datamodel properties)
BOI	Blaise OLEDB Interface
BTR	Blaise Cati Specification
BIC, BIS, BFS	Blaise IS Specifications
DBI	Blaise Database Info
BIN	Binary files
IDM	Blaise IS ID Mapping
BMF	Blaise Menu Files

MSX	Manipula Extended Meta
BXML	Blaise XML Properties
BCI	Blaise Cari Specifications

For any of these formats, the information can be extracted to a BRF file that can be opened with a text editor (or read through Manipula). The format of the file is INI using key=value pairs.

For the purposes of this paper, the BXI is the most useful because it contains all the type information, as well as text and actions for parallel blocks, the status bar, version info, and others. Note that this information is available in the Datamodel Properties, but the BRF contains more, such as the parser version, sizes of the status bar displays, use of aliens, and so forth.

1.4 Cameleon
The old standby has proven itself well over the years, although its functions have been duplicated by the BCP and Manipula. However, it is easy to run and the scripts can be modified without too much difficulty, but requires a more advanced user to decipher them.

Cameleon has access to much of the datamodel Meta information, but not all. Its value lays in the ease of running the script and exporting the metadata in just the right way.

1.5 Datamodel Properties
Similar to using the Blaise Registry Editor to extract information from the BXI, the Blaise Datamodel Properties utility from within the Control Centre also gives the same information but in a different format. The type information is useful because it can be compared against all the types in the datamodel and see which types are not referenced.

1.6 Delta
This utility is probably the closest to getting the most complete information out of a Blaise datamodel besides BCP. Behind the scenes Delta generates XML, and then renders the results into html which are then displayed in the interface. It has an option to select a different stylesheet (View – Select Stylesheet), and the default “dump” of the XML code is through iexml.xsl. However, this places additional html commands in the output that make using the XML more difficult.

A simple stylesheet that just copies all the XML to the output is given below.

DumpDelta.xsl
```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes" encoding="UTF-8"/>
  <xsl:template match="/">
    <xsl:copy-of select="*"/>
  </xsl:template>
</xsl:stylesheet>
```

When this stylesheet is applied within Delta, it will show node information without additional formatting, and is generally unreadable. For example,

```
Do you do any appreciable amount of walking?
Yes1YesNo5NoDoWalking.ASK22.2Treetruetruetruetruetruefalsetruetruetruetruetruefalsetru
etruetruetruetruetruetrue
```

However, the actual XML behind this has much of the information needed to work with. It contains a mixture of Meta information and Delta information. A refinement of the simple stylesheet above would remove the Delta-related elements, such as <RichText> and <htmlproperties> elements, and remove the html formatting of the <cat_rules> element.

The information can be stored from Delta by selecting File – Save – Save all details. This would store an html header file, and then a separate html-named file for each node of the tree. These files are actually xml files, and need a small adjustment. First, the filename extension needs to be changed from .htm to .xml. Secondly, the statement

```
<?xml version="1.0" encoding="utf-16"?>
```

should become

```
<?xml version="1.0" encoding="utf-8"?>
```

Another stylesheet, or Manipula script, or other utility should be sufficient to combine all the separate files into one XML file and change the encoding to utf-8.

Delta has the advantage of also exporting the rules text for each block in the datamodel (but not for the datamodel itself). This information is stored in the <cat_rules> element for blocks, but is html-based.

For example, here is a question from a small questionnaire:

```
DoWalking (B1)
   "Do you do any appreciable amount of walking?"

   TYesNo
```

And here is an excerpt of Delta XML for the same question:

```
<qnode caption="DoWalking" position="2.1" positionf="2_1" captiontype="QUESTION"
   imgsrc="quest.gif" imagepath=".">
   <cat_fielddef fieldnamestripped="SubQs.DoWalking" fieldname="SubQs.DoWalking"
      fieldkind="DataField" fieldtype="Enumeration" typename="TYesNo" fieldsize="1">
      <field_descriptives dummy="neverhide"/>
      <field_specifications dummy="neverhide"/>
      <question_field tag="B1">
         <fieldattributes empty="false" dontknow="true" refusal="true"/>
         <field_description/>
         <field_text>
            <langtext langid="ENG">
               <origrichtext>Do you do any appreciable amount of walking?</origrichtext>
               <RichText FontFamily="Arial" FontSize="12" FontColor="rgb(0,0,0)"
                  FontBold="false" FontItalic="false" FontUnderline="false">
                  <RichTextLine Align="Left">
                     <RichTextElement Text="Do you do any appreciable amount of walking?"
                     />
                  </RichTextLine>
               </RichText>
            </langtext>
         </field_text>
```

```
<field_answers>
            <answers_closed>
               <answer>
                  <name>Yes</name>
                  <code>1</code>
                  <langtext langid="ENG">
                     <origrichtext>Yes</origrichtext>
                     <RichText FontFamily="Arial" FontSize="12" FontColor="rgb(0,0,0)"
                        FontBold="false" FontItalic="false" FontUnderline="false">
                        <RichTextLine Align="Left">
                           <RichTextElement Text="Yes"/>
                        </RichTextLine>
                     </RichText>
                  </langtext>
               </answer>
               <answer>
                  <name>No</name>
                  <code>5</code>
                  <langtext langid="ENG">
                     <origrichtext>No</origrichtext>
                     <RichText FontFamily="Arial" FontSize="12" FontColor="rgb(0,0,0)"
                        FontBold="false" FontItalic="false" FontUnderline="false">
                        <RichTextLine Align="Left">
                           <RichTextElement Text="No"/>
                        </RichTextLine>
                     </RichText>
                  </langtext>
               </answer>
            </answers_closed>
         </field_answers>
      </question_field>
  </cat_fielddef>
  <cat_statement_text>DoWalking.ASK</cat_statement_text>
```

1.7 Manipula

Manipula is capable of pulling an extensive list of data and Meta items from the datamodel, but not the rules text and some information on external files. It can, however, retrieve fields in the order of the rules as well as fields in order of their definitions.

Writing manipula scripts to accomplish the work is not as complex as writing a program for BCP, but still involves good working knowledge of how to navigate through the structure of the datamodel. It also has the advantage of being flexible, easily updated, and current with new builds of Blaise by simply preparing the script again.

The capabilities of Manipula have been expanded for some time, and Getfieldinfo(), Getmetainfo(), and Gettypeinfo() are invaluable for retrieving the datamodel meta information.

1.8 XSD Schema Wizard

The XSD Schema Wizard exports an XML schema for a Blaise datamodel. The field information is based upon the list of defined fields rather than the fields actually used within the datamodel, and are missing some pieces that may be useful: field and block sizes, description texts, and parameters. However, it appears the schema generated is compatible with the Manipula Blaise to XML file format.

2. Crosswalk

Crosswalks are a listing of all variables within a program and how they're used: within functions, assignments, questions asked, shown, kept, referenced, fills, and so on. They are essential tools for authors by identifying variables that are no longer being used, or ones that potentially could cause confusion, such as the same named variable used within different blocks. They help pinpoint old code that should be removed, or new code that has not been fully implemented.

Crosswalks can be used in different environments, but for this discussion the DEP datamodel will be used.

Let's assume that we have the Meta information in a format that is usable from one or more of methods described earlier, and the information is complete. Then the type of crosswalk that could be created from them could be classified according to the type of metric, such as fields, assignments, or parameters. Below is a table summarizing the sort of crosswalk items that should be possible to extract and use for documentation.

Item	Crosswalk metric (references)	Counts Metric (#)	Notes	Metric can be found from
Alien Procedure	Reference in RULES	# references	Independent of fields/auxfields	BCP Delta
Alien Router	Reference in RULES	# references	Always associated with a block	BCP Delta
Arrays	ASK KEEP SHOW Assigned Calculations	Total # # ASK # KEEP # SHOW # Assigned # calculations	Usually requires loops. A structure used on blocks, fields, auxfields, and locals.	BCP Cameleon Manipula Delta XSD Data Centre
Auxfields	ASK KEEP SHOW Assigned Fill reference Calculations Languages Question text Description Tag DK RF EMPTY	Total # defined Total # used # ASK # KEEP # SHOW # Assigned # used in fills # calculations Length of all question texts # descriptions Length of all description texts # codes # tags # DK # RF # EMPTY	Temporary field, does not store data	BCP Cameleon Manipula Delta XSD Data Centre
Blocks	KEEP ASK SHOW Assignments	Total # # ASK # KEEP # SHOW # assigned # parameters Deepest level # embedded	Complex type structure. Can have block assignments	BCP Cameleon Data Centre Delta Manipula XSD
Checks		# CHECK keyword # SIGNAL keyword # checks # signals	Compare # keyword occurrences vs. check/signal to look for inadvertent type	BCP Cameleon (checks/signals) Delta

Item	Crosswalk metric (references)	Counts Metric (#)	Notes	Metric can be found from
Datamodel		Languages Parallel Blocks Key fields Embedded blocks		BCP Cameleon Delta Manipula
Externals	Declaration to external database	# references	Needed before using SEARCH, LOOKUP statements in the rules	BCP Cameleon (external parameters) Manipula Delta (lookup and external field kind).
Fields	ASK KEEP SHOW Assigned Fill reference Calculations Languages Question text Description Tag DK RF EMPTY	Total # defined Total # used # ASK # KEEP # SHOW # Assigned # used in fills # calculations Length of all question texts # descriptions Length of all description texts # codes Length of all code texts # tags # DK # RF # EMPTY	Stores data	BCP Cameleon Manipula Delta XSD Data Centre
Layouts		# layout sets # page breaks # infopane refs # fieldpane refs # grid refs		BCP Source code
Locals	Assigned Used in fills, calculations	Total # # assigned # calculations # used in fills	Loop counters, fills, calculations	BCP Manipula Delta (used)
Parameters	Assigned Used in fills, calculations	Total # # IMPORT # EXPORT # TRANSIT # generated # assigned # calculations	Treat like locals. Can be explicit or generated	BCP Cameleon Manipula Delta
Procedures	Declarations	# definitions # parameters	Can call other procedures, be used as alien routers/procedures	BCP Source code Delta (used)
Rules	N/A	# lines of codes File references # conditions # functions # methods # procedure calls # loops # assignments		BCP Delta (not for datamodel level)

Item	Crosswalk metric (references)	Counts Metric (#)	Notes	Metric can be found from
Types	Defined Fields Auxfields	# defined # in Fields # in Auxfields Per kind (Named, Block, Date, Classification, Enumerated, Open, String, Set, Integer, Real, Time, Dummy)	Can be declared but never used	BCP Datamodel properties Cameleon Manipula Delta XSD
Uses	Declaration to external datamodel	# references	Needed before using EXTERNALS statement	BCP Source code

Item refers to the crosswalk metric, the references column to how the metric is used, the counts column to the number of times the metric is encountered (discussed more below), a few notes, and finally the places the metric could be obtained.

For example, if we were to look at the Locals item and its crosswalk metrics we could get some output from a utility that might look like this:

Locals	Assigned (line)	Referenced (line)	Code

Counter	119
	121
		
		129
			Counter := 0
			Counter := Counter + 1
			IF Counter > 5 THEN

I	120	
		122
		130
		132
			FOR I := 1 TO 10 DO
			IF HHL[I].Name = EMPTY THEN
			xFill := HHL[I].Name
			xFill2 := HHL[I].Name + ‘,’

J	 140	135	IF J > 2 THEN FOR J := 1 TO 10 DO
K	--	--	--

The line number would ideally refer back to the source code, but potentially could also reference a generated document that contains “new” source code.

Then information in the results table could be interpreted, with potential problems flagged:

Variable Counter is assigned twice, used once
Variable I is assigned once and referenced three times
Variable J is referenced once, and assigned only once. **reference before assignment **
Variable K. ** Never assigned, never referenced **

Common loop variables potentially could be used in many places and be used before they are initialized, such as for variable J. An automated report could indicate suspect situations that may be hard to catch otherwise.

Another view of the features of each method and the Meta information they contain are shown below.

Meta	BCP	Data Centre[1]	Datamodel Properties	Delta	Cameleon	Manipula	XSD
Arrays	X	X		X	X	X	X
Block	X	X		X	X	X	X
Block size	X			X	X	X	
CHECK, checks	X			X		X	
Codes (text, value)	X			X	X	X	X

Meta	BCP	Data Centre[1]	Datamodel Properties	Delta	Cameleon	Manipula	XSD
Conditions	X			X			
Datamodel attributes	X			X		X	
Datamodel languages	X		X	X	X	X	X
Datamodel name	X			X	X	X	X
Datamodel title	X			X	X	X	X
Dictionaries used	X			X			
Externals	X			X			
Field attributes	X	(data)		X	X	X	
Field (name, text, ...)	X	(name)		X	X	X	X
Field kind	X			X	X	X	
Field long name	X			X	X	X	X
Field size	X			X	X	X	
Field/Block type	X			X	X	X	X
Field tag	X			X	X	X	X
Parameters	X			X	X	X	
Parallels	X		X				
Primary, secondary keys	X			X	X	X	X
Rules	X			(no root) [2]			
SIGNAL, signals	X			X		X	
Status bar	X		X				
Types	X		X	X	X	X	X

[1]Data Centre only exports data or creates a datamodel based upon the data to be exported. The meta information is sufficient for this need. Hence, only the basic field name within the block structure can be retrieved. Only when DK/RF values entered on a field will it show in the data.

[2]Delta does provide a version of the rules, but the export using a simple dump of the information has already been formatted into html. A slightly more complex stylesheet to convert this to xml would make it easier to use. Delta currently does not export the top-level rules of a datamodel (build 4.8.4.1737).

3. Metric Counts

It's easy enough to find the width of the data record via the structure browser, but what's the answer to the total number of variables and number of each type? How many assignments are made within the code, how many questions are asked, how many arrays are there, the number of blocks, etc.?

Metric counts refer to the number of times a metric is encountered within the datamodel. Blaise comes packaged with an existing Cameleon script, TechDesc.cif that reports a number of these metrics within the data structure of a datamodel. However, there is more information that can be retrieved in a variety of ways. The table for the crosswalk also has a column for the counts metric.

3.1 How to use the counts

Counts by themselves don't mean anything, but can be used as a complexity measure, or could be used to compare two datamodels to see how much effort (as a percent) it would take to bring one datamodel up to another, or how a datamodel has changed over time. Certain measures, such as generated parameters, could help indicate areas to improve to decrease hidden overhead.

Counts could be used to locate metrics that are overused as well as those underused. The former would be useful to point out areas that may need a reduction of complexity, while the latter may be areas that could be removed.

4. Finding unused items

During the course of development, and when working with succeeding years of the same survey, changed over time, there is a potential for having old fills, auxfields, locals, and types remain when they should be removed. And for new development, fields often can be declared but can be missed in testing. The question is how can we find these items?

4.1 Unused items: fills, auxfields, locals, types

An unused fill occurs when a variables (such as an auxfield or local) has been declared, and perhaps even is assigned but is never used within question text or the construction of another fill, or used as a parameter to a block or procedure.

For example, it's not too hard within the Blaise Control Centre to locate unused fills manually (enhanced search on each fill declaration and look to see if there are any references in the rules or displays in texts), but an automatic routine could locate these unused variables more quickly. The unused variables take up space in both the source code and in the datamodel when running, and removing unneeded variables would improve the load time (probably just marginally) for a survey. More practically, it makes the code tighter and easier to maintain.

Once the metrics and counts have been created for a datamodel, it would be a simple thing to look for variables that are assigned but never used as a fill in question text, assigned and never referenced, or never assigned or referenced.

Likewise, an overabundance of unused types makes the code more complex than necessary. (Types are linked to the datamodel properties, and removing a type from the program also removes the entry from the BXI file.) However, Blaise does allow libraries with the express purpose of providing standard types across a number of surveys, and so removing unused types may not fit the organization's methods.

To find the unused types is a two-step process. First, all the defined types can be examined, and then all fields, auxfields, and parameters are examined for their types. The two lists are then compared. If a type has been declared but isn't used it should appear in the first list but not the second. A report could then be created showing these discrepancies, with either the line number or at the block where the unused types are defined.

4.2 Unused fields

Questions that are defined but never used could be a problem in the making. Either they were purposely removed from the logic and never removed from the FIELDS declaration, or they were accidently removed. And what could be worse than to field a study without collecting data on critical questions? Finding unused questions can be done manually by the same process as finding unused fills by searching for references on each question. This is a time consuming process.

Unused questions can be found by running in a similar manner as other unused code: run a metrics and counts report and then look for questions that have a zero count for being ASKed, SHOWed, KEEPed, or used in a reference.

Blaise has generated statements at the end of each block for fields/auxfields that have been defined but have not explicitly been ASKed/SHOWed in the rules (mostly for assigned fields/auxfields). This is another area to examine in order to look for potential problems.

5. Code complexity

Another use of the metric counts is to help calculate the complexity of a Blaise program. By using a series of measures, the datamodel can be analyzed and comparisons made between authors, instances of a survey, and similar or different surveys.

For example, it would be safe to say that if a survey has few IF..THEN statements it will be relatively simple, and one that has 50% IF..THEN statements versus asked questions it is more complex, and one that has 200% IF..THEN statements would be even more complex, and one that has in addition many levels of conditional embedding would be very complex.

There are a variety of measures that can be used to measure Blaise datamodel complexity. The practical application of knowing the complexity of a datamodel is to be able to relate that to the amount of effort to modify, add, or remove code, or to develop similar surveys.

The complexity of a datamodel is determined by a number of metrics. They can include, but are not limited to:

- Number of questions
- Number of fills
- Lines of code (rules)
- Number of preload variables
- Number of decision points
- Number of procedure calls
- Number of languages
- Number of blocks
- Depth of blocks
- Number of operands in an expression

Furthermore, some metrics are linear, such as a series of ASKed questions, and other metrics should be exponential, such as the depth of decision points in the rules. Therefore, a program with a series of linear decision points is simple, but a series of decision points that are embedded become complex very quickly.

A weight should be applied to each metric. In general, linear items, such as the number of questions, should have a weight of one. Decision points should be exponential, and such things as languages should be factored as a multiplier of complexity, as shown in the table below.

Metric	**Weight**	**Symbol**
Number of fields asked/showed	1	Q
Fills displayed	1	F
Number of assignments	1	A
Lines of code (rules)	1	R
Number of preload variables (used/displayed)	1	P
Depth of each decision point	2	2^D
Number of procedure calls	1	C
Number of fields in blocks	1	B
Depth of blocks	1.5	1.5^K
Number of languages	1	L

Therefore, one possible measure of complexity for a survey would become

$$\boxed{\text{Complexity score} = (Q + F + A + R + P + (2^D)\ldots + C - B + (1.5^K)\ldots) * L / Q}$$

This reads as the summation of (the number of questions, number of fills, lines of code, preload variables, each decision point's depth, procedure calls, each block's depth, and operands) multiplied by the number of languages defined in the datamodel.

Blocks at the top level of the datamodel would have a score of 1 (1.5^0), a block within a block then has a score of 1.5 (1.5^1), and another block within that would have a score of 2.25 (1.5^2).

A condition, such as IF A = 1 THEN, would have a score of $2^1 = 2$, and another condition within that would have a score of $2^2 = 4$, and so forth. The ELSE portion of an IF would count as a condition.

For example, suppose we had a very simple datamodel with no fills, blocks, preload…

```
A
B
C

Score is 2:  (Q=3 + F=0 + A=0 + R=3 + P=0 + D=0 + C=0 – B=0 + K=0) * 1 / 3
```

A slightly more complex datamodel:

```
A
IF A THEN
  B
ELSE
  C
ENDIF

Score is 4.3:   (Q=3 + F=0 + A=0 + R=6 + P=0 + (D=2^1 + 2^1) + C=0 – B=0 + K=0) * 1 / 3
```

And even more complex:

```
A
IF A THEN
  B
  IF B THEN
    D
    SomeVal := 1
  ELSE
    E
  ENDIF
ELSE
  C
ENDIF

Score 6:  (Q=5 + F=0 + A=1 + R=12 + P=0 + (D=2^1 + 2^2 + 2^2 + 2^1) + C=0  – B=0 + K=0) * 1 / 5
```

Using these examples implies that the third datamodel is three times as complex as the first, and about 50% more complex than the second.

The number of fields in blocks has been expressed as a positive element by subtracting these values from the score. That's because breaking down the datamodel into parts is beneficial, and although embedding blocks within each other is useful it does raise complexity.

Of course, this oversimplifies a number of things about producing a survey and the environment it is intended to be in. It leaves out other measures that are worthy of consideration: number of lines used in layouts, web or CATI, external lookups, alien routers, procedures via DLLs and Manipula, and so forth. This measure is intended as a starting point for further exploration and discussion.

Although the complexity score as discussed is at the datamodel level, this can be applied against a single block or procedure. Then scores could be computed in parts and those areas of programming that get a high complexity score are areas that should be reviewed.

6. Reducing errors

Even the best authors and the most careful checking by an expert team of testers will miss mistakes and errors. Not only will these have an impact for data collection, but when it comes time for making documentation, creating codebooks, running analysis, and so forth there will be problems found that would have been corrected if only they had been caught. Being able to detect and correct these sorts of errors is beneficial all around. Below are a few examples that can be created once a metrics and counts analysis has been generated.

6.1 Inadvertent checks and signals

These are situations where an assignment was intended, but instead a check was created by a typo. For example, A = 1 is a check while A:= 1 is an assignment. These can be located by the counts and metrics by looking for all the checks with defined error text.

6.2 Ill-defined lengths

Blaise checks all simple string assignments and numeric assignments against the width of the variable being assigned when the datamodel is prepared. However, the parser does not catch situations where the operand is made of a series of constants, such as xFill:= ‘this ’ + xSomeOtherFill + ‘ that’ for a fill, or xNumber := 1 + 5 for a numeric expression. It is possible to calculate the length of each operand of an assignment and therefore the overall length needed for a fill, or the possible ranges needed for a numeric assignment. Catching these during the development period would reduce the number of errors that could occur from truncated fills or “imputation errors” that happen during data collection.

6.3 Missing Parts

Different parts of the source code can be missing by accident or on purpose, and while not critical to making a working survey, these are things that could have impact to authors, interviewers, researchers, analysts, and others before, during, and after data collection. Making sure everything is completed is time consuming up front but could reap huge savings afterwards.

Some areas that can be reported by the metrics and counts report that would help pinpoint areas that need work include:

- Texts for each language (Questions, Descriptions, Blocks, Checks/signals, Codes)
- Datamodel text (header)
- Explicit numbering in code frames
- Language identifier for each text
- Layouts for each language
- User-defined types for each field
- Detection of generated parameters (implied parameter)

Needless to say, programming time available for projects is rarely in excess. However, if these areas and other like them are addressed then time spent in debugging should decrease. And during analysis trying to figure out what happened should also decrease because the datamodel will be well-documented.

7. Conclusion

The large variety of existing tools and utilities within the Blaise system makes it possible to retrieve a number of metrics about the Blaise datamodel. Using these metrics then allows authors to produce a higher quality datamodel that fulfills immediate and future needs, reduces errors, and should decrease development time. They would also facilitate estimating programming efforts and costs for future Blaise surveys of similar types.

References

Rhonda Ash and Danilo Gutierrez, Longitudinal Survey Data – “Move It Forward”, IBUC 2010 Proceedings pp.195-200.

Barbara S. Bibb, Lillie Barber, Ansu Koshy, Coding Tricks To Save Resources, IBUC 2010 Proceedings pp.188-194.

Rebecca Gatward, An evaluation of Delta, a documentation tool, IBUC2004 Proceedings pp.389-410

Jean-Pierre Kent, Performance and Design, IBUC1995 Proceedings pp.73-102.

Peter Sparks, A Systematic Approach to Debugging in the Blaise Environment: An Author's Perspective, IBUC2009 Proceedings pp.185-195.

Margaret Tang and Daniel Collison, Blaise Testing , IBUC 2007 Proceedings pp. 109-116.