Blaise Programming Techniques for Better Documentation

Peter Sparks & James Hagerman (University of Michigan, USA)

1. Introduction

There has always been a divide between the finished software product and what really is going on internally. In the survey business, it is sometimes necessary for non-programmers to examine the actual source code. This is not always desired, either by the non-programmer or the programmer! However, a number of programming methodologies can be implemented to ease this pain and make the program more robust, easier to maintain, easier to transfer, and so on.

This paper examines different methods that can be used by the programmers (also known as “authors” or “developers”) to aid in providing clear documentation for a variety of users: other programmers (Blaise and non-Blaise), non-technical Survey Managers, and automated documentation systems (i.e., MQDS [Michigan Questionnaire Documentation System], Delta). Overly complex code typically yields poor or no documentation, reduced performance, and is not robust. Programming for good documentation improves portability and maintainability of the program code. It also provides better output from automated documentation systems and the ability to compare cross-wave instruments.

2. The Basics

First rule of thumb: Make the program usable by humans.

People are the single best method to actually understand the internals of a programmed survey. Even so, there are limits that are reached by anyone. Hence the following suggestions are given to make sure the program code is usable without the aid of automated documentation systems.

2.1. Comments

It is the bane of all programmers and desired by everyone. The best practice is "comment as you go." Write the comment as the code is being programmed and plan on being like a tourist - get the snapshot now and do not wait until another time. You will find that your own comments will help as the code is being developed. It is even more important when working with a team of programmers.

It is difficult to over-comment program code. You will know when you have reached that point because you cannot find the actual program code. This is the point in time that a separate document needs to be written.

2.1.1. Header Comments

Place a comment header at the start of each file. Include things such as the study, date, modification history, authors, copyright and use notice, and what the section is about. Keep this up-to-date! *Note:* some version control software (see 0) can automatically insert the date and time of the last modification.

```
{******************************************************** }
{                        STUDY NAME                        }
{                        Module A                          }
{                  Main Application Control File           }
{PROGRAMMER: Jim Hagerman                                  }
{CREATED:    04/21/2004                                    }
{MODIFIED:   04/05/2005                                    }
{COPYRIGHT 2005 The Regents of The University of Michigan  }
{******************************************************** }

{Production Release Information:

      Rel 1 03/04/2005 (JRH)
      Rel 2 04/05/2005 (JRH)

      Changes Implemented in 04/05/2005 version:
       03/18/2005 A21 - Deleted "(contribuciones)" and replaced "las
                        contribuciones" with "los impuestos"
}
```

2.1.2. Block/Procedure Comments

Before each block and procedure put a description of the block (input, output, purpose ...)

2.1.3. End of Structure Comments

If the end of a structure (ENDIF, ENDDO, ENDBLOCK ...) is far away from the start of the structure (IF, FOR, BLOCK ...) it is helpful to create a comment following the end structure statement duplicating the condition for the start of the structure. This is more important where a large number of structures are all ending together.

2.1.4. Rules Comments

There always will be instances of complex program code. Document these as with a plain description of what is being accomplished before using technical details. Your goal is to write a comment so that you could pick up the same program code years later and know what it's doing.

Rule of thumb: could a new programmer understand the code by reading it without comments? If "No," then comment and simplify!

2.1.5. Revision Comments

After the survey has been released and amendments have to be made, revision comments are needed. There should be a revision history maintained (usually at the start of each file) and a revision comment before each change. In the revision comment put the following:

Date, Programmer's initials, Description of the change.

This will make it useful for re-examining the history of changes when it's time for data analysis.

2.1.6. Well-named variables

Because the Blaise fieldname is used for routing in the Rules section and it is the default identifier for the data, you may not have the flexibility of using descriptive field names. This is a situation where good programming methods conflict with the intended use of a datamodel: collection and export of the data. If you are in an organization that uses the same datamodel for both purposes then flexibility in choosing field names will be done according to their data use. For example, the actual field name could be SpBirthDate, the field tag labeled A15, and the field description would be “Spouse Birthdate.” This way, the field name would be used in the rules, while the field tag displayed on the status bar, and the field description is displayed instead of the field name in the field pane and could also be used in the final dataset as a meaningful variable label.

2.2. Well-named Response Codes

When possible use meaningful descriptive response code names. For example:

```
 YesNo =    (C01        "Yes",
                         C05 (5) "No")
```

Instead of "C01" the word "Yes" could instead be used:

```
tYesNo =    (Yes (1) "Yes",
                         No  (5) "No")
```

Hence reading the program code will be made much easier.

2.3. Multiple languages

When programming multiple languages be sure to explicitly define each language at the top of the program. This is not only the language code (i.e., Eng for English) but it is the text of the language as well.

```
LANGUAGES =
      Eng "English",
      Spa "Spanish"
```

Throughout the program, use these language tags to explicitly mark the text even though the default will be the next language. For example:

```
TdDate (D154_) ENG "(Please tell me today's date.)

                   @/@/THE DATE ^AFillMonth/^AFillDay/^AFillYear
                   @/THE DAY IS ^AFillDayName

                   @/@/DAY OF WEEK:"

               SPA "(Por favor, dígame la fecha de hoy.)

                   @/@/LA FECHA ES ^AFillMonth/^AFillDay/^AfillYear
                   @/EL DIA ES ^AFillDayName

                   @/@/DIA DE LA SEMANA:" /
                   ENG "TODAY'S DAY OF WEEK"
                   SPA "LA FECHA DE HOY" :

                   (DAYOK      (1)   ENG "Day ok"
                                     SPA "Día ok",
                    DAYNOTOK   (5)   ENG "Day not ok"
                                     SPA "Día no ok")
```

2.4. Standard case conventions

As suggested by the Blaise Reference Manual all keywords should appear in caps, i.e. DATAMODEL, IF, THEN, DO, TABLE, BLOCK, FIELDS, RULES, … Do not use mixed/lower case for keywords whether they are bolded or not in the program editor.

If you can use descriptive names, use "Camel" case, avoid confusing letters and numbers look-alikes (the lowercase letter “l” and the number “1,” letter “O” and number zero “0,” the number “2” and the letter “Z”). Hence a name will look like SpouseBirthDate where each word starts with a capital letter followed by lowercase letters and no underscores.

Tip: The F12 key in the Blaise editor will change the text under the cursor into uppercase and a second press will change it to lowercase.

Tip: The list of keywords is in the C:\Program Files\StatNeth\Blaise 4.7 Enterprise\Bin\Blaise.SHT file that determines which keywords are highlighted in the editor. If there's a new keyword that you want highlighted add it to this list.

2.5. Consistent indenting

Spacing is very important for readability and maintainability of the program. It is relatively easy to spot different programmers in code that does not follow rules for indents, variable naming, layout, and so forth.

2.5.1. Matching Control Structures

Make sure that <u>all</u> your structures that have starting & ending parts are matched-aligned. These include:

IF/ELSEIF/ELSE/ENDIF
BLOCK/ENDBLOCK
TABLE/ENDTABLE
DATAMODEL/ENDMODEL
LIBRARY/ENDLIBRARY
FOR/ENDDO
PROCEDURE/ENDPROCEDURE

Some structures are division markers in the program and do not have matching end members:

FIELDS
RULES
PARAMETERS
TYPE
CHECK
SIGNAL
LAYOUT

For all structures that are inside other structures (i.e., embedded IF statements), indent the next level so that each structure is easy to match the beginning and ending points.

```
PD16_HadEpisode
IF PD16_HadEpisode = Yes THEN

     PD16a_HowManyEpisodes
     SIGNAL
            PD16a_HowManyEpisodes < 90
            Eng "Iwer: please verify number of episodes"

     PD16b_TypesEpisodes
     IF PD16b_TypesEpisodes IN [Mild..Severe] THEN
            PD16b_ExampleEpisode
     ENDIF

ELSE

     PD17_OtherExperience

ENDIF
```

2.6. Well-defined Storage Structure

Place your source code in the same directory structure from one project to another. Place your datamodel release and all associated files, likewise, in another structure. Name all your files consistently so they can be easily found. For example, the main source file with the uppermost Rules section could be named with an extension .BLA, while all additional sections which are included could be named .INC. A quick search (*.bla) will then locate the main file no matter what it is named.

Use the project manager to your advantage (and everyone else's). Absolutely use a project file and maintain the list of all files that belong with the project. Be sure to include your Modelib (.BML), configuration file(s), menu file, and anything else that is required for deploying your project.

Both of these standards will be aids for archiving and recovery.

2.7. Program Defined Types

Use precompiled type libraries by placing all common types into one that will be used throughout your project. Make use of the type file via the LIBRARY command. Since the .lib is compiled it then helps make sure common types are used within your project and across projects in your organization. When archiving a project it is important to put a copy of the source type with the code.

2.7.1. Type Use

Types should be used when feasible to ensure consistency throughout the program. Although many things can be assumed for working with questions it is better to document more than less. For example,

```
MyYesNo    "Answer Yes or No": (Yes, No)
```

Would be better programmed as:

```
TYPE
  tYesNo = (Yes        (1) "Yes",
            No         (5) "No")

FIELDS
  MyYesNo  (A1)  "Answer Yes or No" /
                 "Sample Yes/No Question": tYesNo
```

The field tag and field description have been added, a reusable type has been added, and the answer categories are defined with explicit values instead of i mplied. The defined type also allows the use of various input masks as well as making all tYesNo questions comparable across the survey.

2.8.2. Blocks and Tables as Types

Both Blocks and Tables can be considered complex types consisting of a small datamodel within their structure. Therefore, they should also be preceded by a type designation. For Blocks, the standard has been to use a preceding "B" and for Tables it is with a "T."

```
BLOCK BPerson
           ...
ENDBLOCK {BPerson}

TABLE TPersonGrid
      FIELDS
            Person : ARRAY[1..10] OF BPerson
           ...
ENDTABLE {TPersonGrid}

FIELDS
      PersonGrid : TPersonGrid
```

2.8.3. Input Masks & Types

There is an added benefit to using types: input masks. Since each type can have a custom mask, there is a benefit of having a few types used consistently throughout the program. For those questions that have unique response codes or ranges it is actually better to put that actual type on the field itself rather than creating a unique type. There is less overhead in maintenance, compiling, and so forth if there are only types that are needed stored in the datamodel properties file (.bxi).

2.9. Create Formatting Standards

These are highly attuned to an organization and its own standards, so all fonts and styles, placement, and so forth are not part of this paper. Make use of the customization possible but endeavor to be consistent.

Custom fonts

Blaise allows multiple attributes to be assigned to each custom font letter. In general it is easier to "read" the question text if the formatting tags are not embedded (e.g., @A@BSome interviewing instruction here.@B@A) but are instead defined with a defined composite meaning per tag; for example, interviewing instructions in Arial 10 blue and bold could be set with one attribute (e.g., @I; *I* = *Interviewer*). Close each section of formatted text with its matching formatting tag (e.g., @ISome interviewing instruction here.@I).

A testing feature useful for checking fills is to define a font tag (e.g., @T; *T* = *Testing*) that turns on the red color. These tags are used to surround fills in the question text and make the fills easier to spot during testing/debugging. When the testing is done the @T is set back to default to hide the red highlight.

If possible put the formatting tags in the question text rather than embedding them within fills. This will make it easier to spot and also reduce the risk of having unbalanced enhancements because a fill was active one time and not another.

2.9.1. Info & Field Panes

Create standard named layouts in the Modelib, and use standard font styles as well. Thus, if programmers or project managers switch between projects they understand what is meant by Size18Col1Height0 Infopane and what characteristics the @B formatting tag will have. Make use of these named layouts in the LAYOUT section of the program code.

Avoid scrolling in the info pane because it is known that text not shown beyond the current pane is likely to be missed when being read.

2.9.2. LAYOUT instructions

Layout instructions embedded at the end of blocks and datamodels tend <u>not</u> to be very descriptive as to what the final presentation will look like to the interviewer (or field representative). All that information is buried in the Modelib file.

In order to see what the interviewer sees, the Modelib and DEP configuration files must be read and interpreted – a daunting task since it would essentially duplicate the work of the DEP using the layout, font, grid, and other formatting instructions.

If the purpose of the documentation is simply to retrieve the question text and some basic formatting (bold, underline, italics, colors, font and size) then the layout instructions can be ignored. If the effort is to truly duplicate the DEP screens, then much more work is necessary.

2.10. Program Structure

2.10.1. Blocks

In general, each section should be programmed as a block by itself, and, according to Blaise reference manual, the more a program is partitioned into blocks, the better. The smaller units have several advantages; faster execution speed, easier to maintain, better data manipulation, and so forth. The trade-off is that the more a program is subdivided then the more difficult it is to read the source code.

Remember that blocks (and tables) are essentially complex types. In terms of the Meta information the reference to a block is the same whether one block definition is embedded within another block or that they are all at the same level. If the goal of the documentation is to reconstruct the exact block structure of a datamodel then more information than just the name will be needed.

For example, both block structures below yield exactly the same field name with block notation: A.B.C.SomeField.

```
BLOCK BA
      BLOCK BB
            BLOCK BC
                  FIELDS
                        SomeField : TYesNo
            ENDBLOCK

            FIELDS
                  C : BC
      ENDBLOCK

FIELDS
      B : BB
ENDBLOCK

FIELDS
      A : BA
```

or

```
BLOCK BC
      FIELDS
            SomeField : TYesNo
ENDBLOCK

BLOCK BB
      FIELDS
            C : BC
ENDBLOCK

BLOCK BA
      FIELDS
            B : BB
ENDBLOCK

FIELDS
      A : BA
```

Block Parameters

For program efficiency and lower overhead pass parameters into and out of blocks. Do not reference the block itself when using variables. Remember that generated parameters have the highest overhead, and large numbers of generated parameters will easily bog the execution speed of an interview.

Explicitly state each parameter. By default Blaise treats them as IMPORT and it will be confusing to have the following:

```
BLOCK BA
      PARAMETERS
             IMPORT    piA : INTEGER
                      piB : INTEGER
             EXPORT    peA : INTEGER
                       peB : INTEGER
```

In this example the peB is really an IMPORT parameter because the EXPORT command only is for the field (list) that immediately follows. That is, IMPORT, EXPORT, and TRANSIT are not status change flags like CHECK or SIGNAL. So attempting to assign a value to peB will result in an error by the Blaise compiler.

The form that will work for EXPORT on multiple items is:

```
EXPORT      peA,
            peB : INTEGER
```

Pass parameters whenever possible. Otherwise the variables references that require a parameter-like treatment will be "generated parameters" which are the same as TRANSIT at a high overhead.

2.11. Parallel Blocks

Parallel blocks present an interesting problem for documentation. They can be a view of the same fields and rules that will be eventually collected but presented on demand, or they can be a review of information, or even collecting information that's outside the normal survey (i.e., language switching and how often it occurs).

The names of the parallel blocks are defined at the top of the datamodel. For automated documentation systems these blocks could be presented as unique views.

3. Related assignments and checks closely grouped

When programming edit checks in the rules of the instrument it is good practice to place the checks as close to the source questions as possible. This has the benefit of triggering potential checks quickly rather than searching through all the rules and the person reading the code will also be able to find the relevant questions quickly.

```
A1_Day
IF A1_Month = September OR A1_Month = April OR
   A1_Month = June OR A1_Month = November THEN

      CHECK
            A1_Day < 31 OR A1_Day = NONRESPONSE
            "@D@Rw@R@D @B@RApril, June, September, and
            Novemberonly have 30 days@/@/@|···Please verify
            the day of birth and re-enter@R@B"

ELSEIF A1_Month = February THEN

      CHECK
            A1_Day < 30 OR A1_Day = NONRESPONSE
           "@D@Rw@R@D @B@RFebruary only has 28 or 29 days
           @/@/@|···Please verify the date of birth and
           re-enter@R@B"
ENDIF

A1_Year
IF A1_Year <> NONRESPONSE THEN
      IF A1_Month = February AND (A1_Year MOD 4 <> 0) THEN

            CHECK
                  A1_Day < 29 OR A1_Day = NONRESPONSE
                  INVOLVING (A1_Day)
                  "@D@Rw@R@D @B@RFebruary has only 28 days
                  for the given year@/@/···Please verify the
                  date of birth and re-enter@R@B"
      ENDIF
ENDIF
```

4. Tips

4.1. String vs. Open

Use string types whenever you want the data stored in your dataset and there is a fixed upper size to the information. Use open types whenever the data could be large and probably should be recoded at a later date, and that the data itself is not going to be exported to a data set. You can freely copy information back & forth between these forms but realize the default string size is currently 255 characters (up to 32767 if specified) and the default open size is 32767 characters variable capacity.

4.2. Passing Parameters

If you have an array of information that needs to be passed to a sub block then you should arrange your programming to pass one member of that array to the sub block at a time (similar to working with tables and grids).

4.3. When to use Procedures

This item is certainly open to debate, but a good rule of thumb is to use a procedure when a repeated result (twice or more) is desired that does not involve the interaction of the interviewer. For example, string comparisons, conversion of dates into a formatted date string, complex sorts and searches, complex counts, and the like. It is wise to avoid collecting data within procedures for the following reason:

- The audit trails will contain the keystrokes for the values gathered in a procedure, but the data in a procedure will be lost the same way auxfields are lost, and any remarks gathered in a procedure will be lost in the same way.
- Upon recovery of a suspended case, the DEP will stop at the procedures to re-gather the data and in order to avoid re-asking the procedure, questions flags have to be set, a decision made to ask procedural questions only once or to store the data elsewhere (in which case, why not collect it in a normal block – very messy).

Procedures in Blaise do make logical sense and should be used for routine tasks that will otherwise lengthen code and make maintainability difficult.

In terms of documentation, procedures should be commented as for blocks (inputs, outputs, expected results). They are like blocks in an instrument and so should be placed at the nearest level where they are used or in a separate procedure file (*.prc). Please remember that like blocks, the more levels that are embedded, the more difficult it becomes to decipher the instrument. That is, procedure A calls procedure B calls procedure C calls ... Done correctly, it can be a concise method of performing very complex operations. Done incorrectly it can become a nightmare to read and troubleshoot, decrease performance, and make it difficult to document code.

In short, keep procedures short and to the point and avoid making calls from one to the next (go "wide" instead of "deep").

4.4. Paging in Tables

Break Table pages so no scrolling occurs during entry

```
       LAYOUT
              BEFORE C2a_Names[1]   NEWPAGE
              BEFORE C2a_Names[6]   NEWPAGE
              BEFORE C2a_Names[11]  NEWPAGE
              BEFORE C2a_Names[16]  NEWPAGE
ENDTABLE {TC2a}
```

5. Simplified code approaches

Below, are three examples of writing an IF statement and all reach the same outcome. The point here is that two of these examples are probably better suited to more efficient programming style whereas the remaining example is best suited for documentation purposes. Thus, the question becomes, "Where is the middle ground on this issue?"

Example 1: This could be typical code that a programmer would write. All it tells you is to select people who are not dead and no DKs or RFs. People who read this code are generally more interested who is actually passing this IF statement as opposed to those who are not.

Example 1

```
IF Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu <> DIED AND
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = RESPONSE THEN
```

Example 2: Another example of code a programmer would write but with the inclusion of who passes this IF statement instead of who doesn't. Anyone who reads this code would need to understand the use of the "IN" operator in the IF statement.

Example 2

```
IF Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu IN [FU, HUCS, HU,
   FUMI, M_O, Jail, Mil, Educ, Health, Other] THEN
```

Example 3: This is the type of code would be most easily read by a non-programmer. It is not the most efficient method programming (it works though), but it does provide valuable information regarding the IF statement and is very useful in automatic documentation systems.

Example 3

```
IF Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = FU     OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = HUCS   OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = HU     OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = FUMI   OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = M_O    OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = Jail   OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = Mil    OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = Educ   OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = Health OR
   Coverscreen.CYPersInfo.CYPInfo[I].CYFuHu = Other THEN
```

6. Datamodel Properties

The datamodel properties are ways to enhance the basic functioning of the Blaise program while running in the DEP. These are: masks to the types defined in the datamodel, both general and custom; and additional information set aside for status bars, names and methods for parallel blocks, languages, alien routers, and so forth.

In complex documentation systems, it is possible to extract the datamodel properties such as masks to give an even more accurate representation of the datamodel as it would appear in the DEP. These are important since there is no other place in the source code that gives this indication. However, the masks tend to be interactive (inserting the thousands marker and the currency symbol at the correct places) and not really "seen" until data is entered.

Storing this information from automated systems makes sense because different users of the information may or may not want to see this supplemental formatting.

6.1. Global DK/RF

The program should always contain a global DK/RF for every question. This is accomplished by putting the following code in:

```
DATAMODEL FirstNew "First questionaire"

      ATTRIBUTES = DK, RF
```

6.2. Discontinuous Ranges

If a question requires multiple ranges (i.e., ages 1..120, 995..997), then the range should be put in full (1..997) and a hard check be put in to exclude invalid values.

6.3. Calculations/Constants

All global constants should be declared at the very start of the program to make changes easy to do. Modular (“include” files) constants should be at the very start of module.

For example, get rid of the constant (1980, 1, 1)

```
IF (BirthDate > (1980,1,1)) THEN
      CHECK
           Married = No "^Name is too young to be married"
ENDIF
```

And instead declare it early in the program by using a calculation:

```
cAge18YearsAgo : 0..120
           ...
cAge18YearsAgo := STARTDATE - (YEAR(STARTDATE)-18,
                  MONTH(STARTDATE), DAY(STARTDATE))
```

6.4. Locals

Variables such as reused loop indexes and calculations should be stored as locals. A rule of thumb to consider is whether this temporary information is of value to the project staff. If so, then ask the project staff to verify.

```
LOCALS
      LLoopIndex : INTEGER
```

Calculated ages could be stored as either AUXFIELD or stored in the data (using the KEEP method). Generally, any calculations based upon a current date are evaluated one time at that date, and then are not re-evaluated. If the calculation was always redone every time the form was entered, then there is a potential for wiping out data, or causing questions with missing data to appear. So locals may be used for calculations based on dates, but the source of those calculations should not vary.

7. External Databases

Blaise provides powerful methods of retrieving extra information that is easily retrieved from within the interview. This external data can be used for fills, look-up lists, classifications, searches, and the like. As such, the data within an external database can change through the life of a survey without any change to the datamodel.

There is little that a programmer can do to help in this documentation other than provide reasonable comments at the USES statement (describe what the database is about, primary keys, fields accessed, etc...). More than one EXTERNAL statement can be used to retrieve from the same database. There doesn't appear to be a performance issue from using multiple EXTERNAL statements within a program to the same database though in theory it might have an effect.

8. Other Useful Tidbits

8.1. Project Files

Project files are needed to not only provide a workspace for the development of a datamodel, but they keep the Modelib, run parameters, datamodel properties and other important details together. Without a project file the programmers run the risk of forgetting something when the datamodel is prepared.

8.2. Source Control

It should be an absolute necessity that version control of the source code as well as releases of a datamodel be maintained (i.e., SourceSafe, QWin...). This is not only a safeguard for the normal development of the source code (ability to recover from mistakes, compare prior versions) but it makes it much easier to develop code as part of a team. When it comes time for releases, the datamodel can be recreated and/or archived with the matching code. Although not frequently required, having past versions available to answer questions is important.

9. Electronic documentation methods for better documentation:

9.1. How do you improve your documentation?

Add more to the code rather than less. It's impossible for a program to add more detail that isn't there to begin with. This means adding:

- description element for blocks
- question tags
- language markers per text
- well-defined code values (don't assume a value) and code texts

```
BLOCK BSection_A  "Social Relations II Core Network - Section A"
```

9.2. How do you improve the management and efficiency of your code?

Try to keep logic and coding in small units that are simple and build the program from those units. In other words “block” your code.

9.3. How do you simplify the description of universes and the generation of logic for automatic documentation systems?

Use the IF/ELSEIF construction when possible instead of repeated IF constructions

The IF/ELSEIF structure will check only as many conditions as necessary to find a true statement before skipping the following statements. This generally is greater program efficiency since on average most logic checks will never reach the final condition.

However, when the statement is broken into separated IF parts then each of the IF conditions have to be checked thereby resulting in a slower execution because each has to be checked each time. But, sometimes this is necessary if, as in the example below, the asking of fields (B1, B2, and B3) is not mutually exclusive.

```
A1
IF A1 = Yes THEN
      B1
ELSEIF A1 = No THEN
      B2
ELSE
      B3
ENDIF
```

versus:

```
A1
IF A1 = Yes THEN
      B1
ENDIF

IF A1 = No THEN
      B2
ENDIF

IF A1 = NONRESPONSE OR A1 = EMPTY THEN
      B3
ENDIF
```

9.4. How do you improve the use of text enhancements and avoid redundancies?

Avoid embedding text enhancements that do the same job; for example:

let @B be defined as toggling Bold on

let @S be defined as Arial 11 Bold on Blue color

Then some question text such as:

```
"@BA11. @SSome interviewer instruction@S and @B no more bold here."
```

Results in the DEP as:

A11. Some interviewer instruction and no more bold here.

The inner text formatting of @S overrides the prior @B text formatting. The confusion may arise from the bold being turned on a second time and yet not turned off from the second @S since the @B is still in effect. With slightly more text the same result is given but it is clear that the first and third parts of the sentence are bold while the middle part is an interviewer instruction.

9.5. How do you program readable question text while using text enhancements and fills?

Avoid embedding text enhancements inside fills. Let @B be defined as toggling Bold on.

If your fill assignment is:

```
IF ChildGender = Male THEN
      xFillHeSheUndertake := 'he @Bundertake@B a job'
ELSE
      xFillHeSheUndertake := 'she @Bundertake@B a different job'
ENDIF
```

Then the question text might read "When did ^xFillHeSheUndertake?"

Whereas if the assignments were:

```
IF ChildGender = Male THEN
      xFillHeShe       := 'he'
      xFillDifferentJob:= 'job'
ELSE
      xFillHeShe       := 'she'
      xFillDifferentJob:= 'different job'
ENDIF
```

Then the question text might read "When did ^xFillHeShe @Bundertake@B a ^xFillDifferentJob?"

Ultimately it is easier to read text that has fewer small piece changes than it is to derive the text from one fill.

9.6. How do you document explicitly in your logic without formal comments?

Adding text to logic conditions:

```
IF A1 = C05 "Live at home = No" THEN
```

This is another method for documenting program logic without explicitly commenting in the code. Also, this extra information can be pulled out, via the Blaise Component Pack, and processed by documentation systems in addition to any "normal" information. This feature is undocumented but is still supported in Blaise 4.7.

9.7. How do you define descriptive fills for multiple languages that make it easier for automatic documentation systems to interpret?

If memory is not an issue then create a unique fill per situation instead of reusing the same fill with different contents (e.g., two fills for [he/she] and [is/was]). Create unique fills per language (i.e., FillHeShe_Eng and FillHeShe_Sp).

Use descriptive names that describe the fill itself, such as:

```
AUXFIELDS

      xyou_youanyfamliv: STRING[50]
      {you/you or anyone else in your family living there/
       they/they or anyone else in their family living there}

      xyou_youanyfamliv_sp : STRING[50]
      {Es Ud. dueñ(o/a)/Son Uds. dueños}

      xyou_yourfamliv   : STRING[40]
      {you/you and your family living there/they/
       they and their family living there}

      xyou_yourfamliv_sp : STRING[40]
      {Ud./su familia}
```

9.8. What type of difficulties do arrayed fills cause for automatic documentation systems?

Arrays of fills make documentation difficult, but sometimes they are very useful in the code. These types of fills are good for creating restricted enumerated lists, displaying household members, and creating lists of selected items.

This means any array element could hold a fill for any assignment to the array. For example, John Doe could appear in any of one of ten places for Sibling Name in the example below. Fills that are arrayed are always difficult to document and ideally each element of the array should be documented separately.

```
Sibling_Name           : ARRAY[1..10] OF STRING[50]
```

```
BC17  (BC17)
      "You mentioned ^BCMostRecCount jobs that ended at the same
      time.··Which one of these jobs ^xdoyou_doeshe_doesshe
      consider to have been ^xyour_his_her main job?"

      "Mencionó empleos que terminaron al mismo tiempo.··¿Cuál de
      estos empleos considera ^xUd_el_ella haber sido su trabajo
      principal?" /
      "Most Recent Main Job" :

      TMRJobList, NODK, NORF


TYPE
      TMRJobList =      (Job1        "^xMRJob[1]",
                         Job2        "^xMRJob[2]",
                         Job3        "^xMRJob[3]",
                         Job4        "^xMRJob[4]",
                         Job5        "^xMRJob[5]",
                         Job6        "^xMRJob[6]",
                         Job7        "^xMRJob[7]",
                         Job8        "^xMRJob[8]",
                         Job9        "^xMRJob[9]",
                         Job10       "^xMRJob[10]")
```

9.9. How do Parameters affect automatic documentation systems?

Although the use of parameters in blocks and procedures may be confusing and more difficult to work with than just a straight reference, the programmer should use clearly defined parameters whenever possible. This avoids generated parameters which carry a much higher overhead than passed parameters; passed parameters clearly document all the information a block/procedure needs.

Although they improve execution efficiency greatly, parameters being passed back and forth in an application can cause a great deal of pain for automatic documentation systems due to a significant increase in the complexity of the algorithms. This is primarily due to the automatic documentation systems' need to match up the internal parameter names with the original call to the block or procedure.

This affects not only program logic but the use of fills too; due to data being passed in and out to construct various fills.

9.10. How does implicit definition of field widths affect the programming?

Avoid using straight basic types (i.e., INTEGER, REAL, STRING) for fields and auxfields. These are stored by default values and generally give grief to the data analysts because the ranges are not defined. They also contribute to "loose" programming by allowing values out of an implied range. For example LCount below should have values 0 through a maximum of 5.

However the actual value for LCount is -99,999,999,999,999,999 to 999,999,999,999,999,999 which gives plenty of leeway to make mistakes. This is a much greater concern if LCount is later used as an index (which means Blaise will stop preparation due to an invalid range) or was stored as a field for later data analysis.

```
LOCAL
     LCount : INTEGER
RULES
     LCount := 0
     IF A1 = Yes THEN
           LCount := LCount + 1
     ENDIF
     ...
     IF A5 = Yes THEN
           LCount := LCount + 1
     ENDIF
```

9.11. How do you handle using one instrument for either multiple languages or modes?

Use a flag indicating the mode or language of the interview throughout the instrument. This mode variable can be established at the start of the interview so that appropriate logic and fills can be assigned (e.g., between CAWI and CATI interviews). Such a variable could be filled from the CAPI/CATI/CADI/CAWI functions available in the RULES section along with other variables.

The ACTIVELANGUAGE command provides a handy way to retrieve the current interviewing language that can be used in conjunction with a mode variable. If you keep all fills separate by language by naming them uniquely then you will not need extra programming to make assignments in the RULES section.

In this example the same fill is used for both English and Spanish, and so the ACTIVELANGUAGE command needs to be used in the construction of the fill.

```
IF ACTIVELANGUAGE = SPAN THEN
      IF ChildGender = Male THEN
            xFillHeShe := 'el'
      ELSE
            xFillHeShe := 'ella'
      ENDIF
ELSE
      IF ChildGender = Male THEN
            xFillHeShe := 'he'
      ELSE
           xFillHeShe := 'she'
      ENDIF
ENDIF
```

However, if separate fills were created per language then this could be programmed more simply as

```
IF ChildGender = Male THEN
            xFillElElla := 'él'
            xFillHeShe := 'he'
ELSE
            xFillElElla := 'ella'
            xFillHeShe := 'she'
ENDIF
```

This has the advantage of requiring less work during the rules scan after each field is entered. It is also easier to maintain, reduces the numbers of lines of code, and keeps fills confined to their appropriate language.

Use the ACTIVELANGUAGE command if there are appropriate language-specific questions:

```
IF ACTIVELANGUAGE = SPA THEN
      YearEnterCountry
      EntryConcerns
ENDIF
```

10. Conclusion

Blaise is a powerful programming language that gives the programmer a great deal of versatility. It also then becomes the programmer and the organization's responsibility to establish and follow programming standards. These allow for better resource management between programmers and projects for maintainable code; as well as open the door for automated documentation systems to use the consistency of the code. Standards in programming also lead to other individuals making use of resources that may otherwise be cryptic.

11. References

Altvater, D., Stanford, V., Ziesing, C. (2001). Programming Techniques for Complex Surveys in Blaise. Paper presented at the 7th International Blaise Users Conference, Washington, D.C., September.

Blaise OnLine Assistant (release 4.7.1000). (2005). "Good Programming Practices," and "Examples, Tips, Tricks." Retrieved February 20, 2006 from World Wide Web: http://www.blaise.com/onlinehelp/

Hansen, S.E. (2004). Programming Guidelines for Good Data Documentation. Paper presented at the 9th International Blaise Users Conference, Gatineau, Québec, September.

Kelly, M. (2000). What users want from a tool for analysing and documenting electronic questionnaires: the user requirements for the TADEQ project. Paper presented at the 6th International Blaise Users Conference, Kinsale, Ireland, May.

Pierzchala, M. & Farrant, G. (2000). Helping non-programmers to specify a Blaise Questionnaire. Paper presented at the 6th International Blaise Users Conference, Kinsale, Ireland, May.