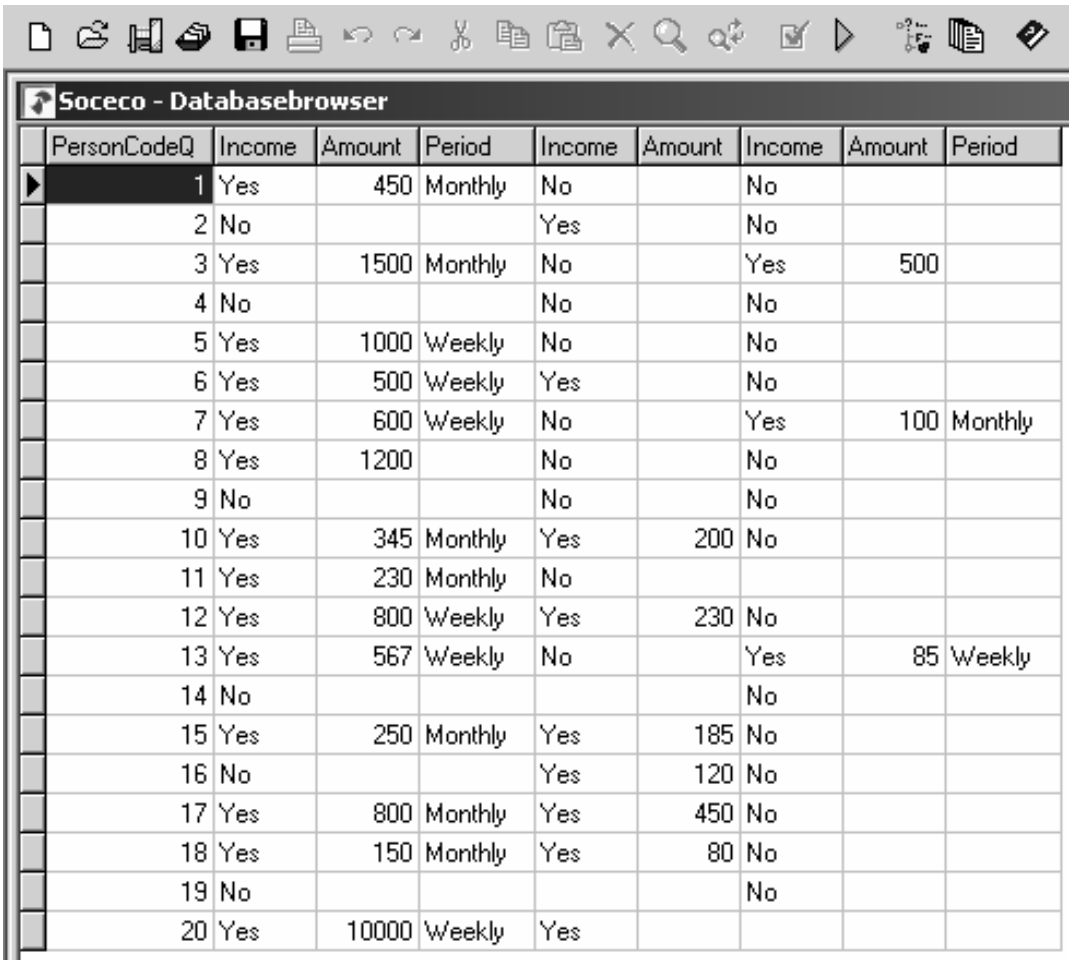
Miscellaneous Examples Of Programming In Blaise And Manipula

Rob Groeneveld, Statistics Netherlands

1. Anonymizing a Blaise database

Starting with a Blaise database with identifiable records, we would like to get rid of the primary key which may be some identifying variable like the name of a company or a registration code for a person or company. It is desirable not only to strip the identifier, but also to sort the records in the database in random order so as to nullify the association between the order of the records and the individuals. As an example, we take the records in the example database Soceco.bdb, part of the examples of the Blaise Component Pack (in StatNeth\Blaise 4.7 Enterprise\Samples\BCP). Here is the complete database:



Soceco - Databasebrowser

	PersonCodeQ	Income	Amount	Period	Income	Amount	Income	Amount	Period
▶	1	Yes	450	Monthly	No		No		
	2	No			Yes		No		
	3	Yes	1500	Monthly	No		Yes	500	
	4	No			No		No		
	5	Yes	1000	Weekly	No		No		
	6	Yes	500	Weekly	Yes		No		
	7	Yes	600	Weekly	No		Yes	100	Monthly
	8	Yes	1200		No		No		
	9	No			No		No		
	10	Yes	345	Monthly	Yes	200	No		
	11	Yes	230	Monthly	No				
	12	Yes	800	Weekly	Yes	230	No		
	13	Yes	567	Weekly	No		Yes	85	Weekly
	14	No					No		
	15	Yes	250	Monthly	Yes	185	No		
	16	No			Yes	120	No		
	17	Yes	800	Monthly	Yes	450	No		
	18	Yes	150	Monthly	Yes	80	No		
	19	No					No		
	20	Yes	10000	Weekly	Yes				

The identifying field is PersonCodeQ, which is in the primary key block:

```
DATAMODEL SocEco
PRIMARY PersonCode
BLOCK PersonCodeBl
  FIELDS
    PersonCodeQ "What is your personal code?" : 1..50
  RULES
    PersonCodeQ
ENDBLOCK
FIELDS
  PersonCode: PersonCodeBl
TABLE TIncome "Income specification"
(etc.)
```

As a first step, we replace the primary key field by a secondary key:

```
DATAMODEL AnonSocEco
SECONDARY
  RK = PersonCode
BLOCK PersonCodeBl
  FIELDS
    RandomKey : 0..200
  RULES
    RandomKey
ENDBLOCK
FIELDS
  PersonCode: PersonCodeBl
TABLE TIncome "Income specification".
.
.
```

This datamodel does not have a primary key. Note that the range of the secondary key is somewhat larger than the range of the original primary key. We will fill this field with random numbers and it is advantageous to have some extra space.

```
PersonCode.RandomKey := RANDOM(200)
```

A first Manipula setup transforms the data from the model SocEco to the model AnonSocEco, leaving out the primary key but adding the random secondary key field RandomKey. There is no association between the value of the secondary key and the previous primary key because the secondary key is assigned randomly.

Why can't the new key be the primary key? The method we have chosen does not preclude the possibility of duplicate values for the new key; hence it cannot be a primary key. If we would use a procedure to generate random unique keys the new key could be the primary key. However, generating new unique keys is more difficult and takes more time per record.

Why is the range for the secondary key bigger than the original range? Because it lessens the chance of getting duplicate keys.

Now we want the put the records in the order of the secondary key to cancel the association between the order of the records in the original Blaise file and the order in the resulting Blaise file. You cannot simply say

```
{ wrong! }
SORT
  PersonCode.RandomKey
```

because Blaise files cannot be sorted. We will process the records in the order of the secondary key and write them to a new Blaise database in this order. An additional advantage is that the datamodel for the new Blaise database can have features such as having the secondary key as the primary key or omitting the secondary key altogether.

The essential part of the second Manipula setup is:

```
INPUTFILE
  In1: InputMeta ('SocecoRandomSort', BLAISE)
  SETTINGS
    KEY = SECONDARY(RK)
OUTPUTFILE
  Out1: OutputMeta('AnonSoceco', BLAISE)
```

We tie everything together in a Maniplus program:

```
{ File: RandomSortRecs.man }
PROCESS RandomSortRecs
  { This setup starts with the Blaise file with the field
PersonCode as the primary key, adds a random number as a
secondary key while omitting the primary key, then processes
these records in the order of the secondary key writing the
records to a new Blaise file }
  SETTINGS
    INPUTPATH = ''
  USES
    OutputMeta 'AnonSoceco'
  OUTPUTFILE
    Out1: OutputMeta('SocecoRandomSort', BLAISE)
  AUXFIELDS
    Reslt: INTEGER
  MANIPULATE
    Reslt := CALL('RandomSort1 /OSocecoRandomSort')
    Reslt := CALL('RandomSort2')
    Out1.ERASE
```

Note that the intermediary database SocecoRandomSort.bdb is deleted at the end of this program.

2. Printing Open Fields In Field Order In A Blaise Database

In a Blaise database obtained as a result of a survey, the task is to print the Open Fields in field order, i.e., firstly the answers from all records in the first open field, then the answers from all records in the next open field, and so on. That is, if we have three records with three open fields, like this:

OpenField1	OpenField2	OpenField3
aaa	bbb	ccc
ddd	eee	fff
ggg	hhh	iii

we want output in the form

```
OpenField1
==================
aaa
ddd
ggg

OpenField2
==================
bbb
eee
hhh

OpenField3
==================
ccc
fff
iii
```

For the output file, we use this datamodel:

```
DATAMODEL OpenQuestions
FIELDS
  ALine: STRING[1200]
ENDMODEL
```

And use the output file type PRINT.

The start of the Manipula program goes like this:

```
SETTINGS
  CONNECT = NO
  AUTOREAD = NO

USES
  InputMeta 'AllQuestions'
  OutputMeta 'OpenQuestions'

INPUTFILE
  InputFile1: InputMeta ('AllQuestions', BLAISE)
OUTPUTFILE
  OutputFile1: OutputMeta ('OpenQuestions.txt', PRINT)
```

The setting CONNECT = NO is used because there are no fields in input and output which have the same names. The setting AUTOREAD = NO is used because we will be traversing the input file from start to finish once for each open field, under programmatic control. The datamodel AllQuestions contains both the normal fields and the open fields.

To write all the answers to an open field OpenField1 to output, we can use the following fragment:

```
    Inputfile1.RESET
    InputFile1.READNEXT
    OutputFile1.LINE(2)
    OutputFile1.PRINTSTRING('OpenField1 ')
    OutputFile1.PRINTSTRING(FILL('=', 80))
```

```
REPEAT
  OutputFile1.ALine := OpenField1
  IF LEN(ALine) > 0 THEN
    OutputFile1.WRITE
  ENDIF
InputFile1.READNEXT
UNTIL InputFile1.EOF
```

This produces the output text in the form

```
OpenField1
============
aaa
ddd
ggg
```

Note that the field is mentioned twice, first its name as the string ‘OpenField1’ and the second time as itself in the assignment:

```
  ALine := OpenField1
```

When we want to turn this sequence of statements into a procedure, we face the problem that parameters in a procedure cannot be of OPEN type. The only types allowed are: user-defined, INTEGER, REAL, STRING, DATETYPE or TIMETYPE. A user-defined type may not be a based on a BLOCK. So this is not possible:

```
{ wrong! }
PROCEDURE OutputAField
  IMPORT strFieldName: STRING
  IMPORT FieldName: OPEN
INSTRUCTIONS
  Inputfile1.RESET
  InputFile1.READNEXT
  OutputFile1.LINE(2)
  OutputFile1.PRINTSTRING(strFieldName)
  OutputFile1.PRINTSTRING(FILL('=', 80))
  REPEAT
    OutputFile1.ALine := FieldName
    IF LEN(ALine) > 0 THEN
      OutputFile1.WRITE
    ENDIF
    InputFile1.READNEXT
  UNTIL InputFile1.EOF
ENDPROCEDURE
```

Hence, we must remove the assignment with the OPEN type from the procedure:

```
OutputFile1.ALine := FieldName
```

And create two procedures. Since the loop REPEAT…UNTIL cannot be split across two procedures, we make a procedure for the heading:

```
PROCEDURE Heading
  PARAMETERS
    IMPORT Fieldname: STRING
  INSTRUCTIONS
    Inputfile1.RESET
    InputFile1.READNEXT
    OutputFile1.LINE(2)
    OutputFile1.PRINTSTRING(Fieldname)
    OutputFile1.PRINTSTRING(FILL('=', 80))
ENDPROCEDURE
```

And a procedure for writing to the output file:

```
PROCEDURE WriteToOutput
  INSTRUCTIONS
    IF LEN(ALine) > 0 THEN
      OutputFile1.WRITE
    ENDIF
    InputFile1.READNEXT
ENDPROCEDURE
```

For every OPEN field we use the two procedures plus the assignment and the REPEAT loop:

```
Heading('OpenField1')
REPEAT
  OutputFile1.ALine := OpenField1
  WriteToOutput
UNTIL InputFile1.EOF
Heading('OpenField2')
REPEAT
  OutputFile1.ALine := OpenField2
  WriteToOutput
UNTIL InputFile1.EOF
.
.
```

This produces the desired output of the OPEN fields. Note that only fields which have some content are written because we test if LEN(Aline) is greater than zero.

3. Partially Filled, Nested Arrays In A Survey

The usual way of programming an array with a varying number of filled elements in a Blaise survey is to ask in the first place for the number of repetitions, e.g., the number of people in the household, followed by a FOR loop to put a set of questions to each member of the household, for example,

```
  ..
  FIELDS
    HHSize "How many people are there in this
household?": 1..10
    Person: ARRAY[1..10] OF BPerson
  RULES
    HHSize
    FOR i := 1 TO HHSize DO
      Person[i]
    ENDDO
```

Another way is asking, after each set of questions, if there is a further set of questions to be asked. An example is this datamodel:

```
DATAMODEL Sandwiches
BLOCK Blk_Sandwich
  FIELDS
    Name "What sandwich did you eat?": STRING[15]
```

```
    MoreSandwiches "Did you eat more sandwiches?": (Yes,
No), EMPTY
  RULES
    Name
    MoreSandwiches
ENDBLOCK { Blk_Sandwich }
LOCALS
  I: INTEGER
FIELDS
  Sandwich: ARRAY[1..10] OF Blk_Sandwich
RULES
  Sandwich[1]
  FOR I := 2 TO 10 DO
    IF Sandwich[I - 1].MoreSandwiches = Yes THEN
      Sandwich[I]
    ENDIF
  ENDDO
ENDMODEL
```

After each question about the sandwich you ate the question is asked if you ate more sandwiches. If the answer is Yes, the question about the next sandwich is asked. Note the way the FOR loop works: the first element must be asked before the FOR loop is started, because inside the FOR loop reference is made to the question MoreSandwiches in the previous element of the array.

We can expand this model by asking about the meals the person ate today. Each meal can include sandwiches or not:

```
DATAMODEL Meals
BLOCK Blk_Sandwich
  FIELDS
    Name "What sandwich did you eat?": STRING[15]
    MoreSandwiches "Did you eat more sandwiches?": (Yes,
No), EMPTY
  RULES
    Name
    MoreSandwiches
ENDBLOCK { Blk_Sandwich }
BLOCK Blk_Meal
  FIELDS
    Name "What meal did you eat with sandwiches?":
STRING[15]
    Sandwich: ARRAY[1..10] OF Blk_Sandwich
    MoreMeals "Did you eat more meals today with
sandwiches?": (Yes, No)
  LOCALS
    I: INTEGER
RULES
  Name
  Sandwich[1]
  FOR I := 2 TO 10 DO
    IF Sandwich[I - 1].MoreSandwiches = Yes THEN
      Sandwich[I]
    ENDIF
  ENDDO
```

```
  MoreMeals
ENDBLOCK { BMeal }
FIELDS
  Name "What is your name?": STRING[15]
  Meal: ARRAY[1..10] OF Blk_MEal
LOCALS
  I: INTEGER
RULES
  Name
  Meal[1]
  FOR I := 2 TO 10 DO
    IF Meal[I - 1].MoreMeals = Yes THEN
      Meal[I]
    ENDIF
  ENDDO
ENDMODEL
```

The field MoreMeals asks if any more meals with sandwiches were eaten.

A datamodel like this can be applied to an inventory of the knowledge of people in a department, both in terms of the areas of expertise and the length of the time spent in specific jobs. A few example questions and answers:

First level:

Question: In which divisions did you work?
Answer: Social Statistics

Second level:

Question: In which areas within the Division of Social Statistics did you work?
First answer: Household Surveys
Second answer: Population Statistics

Third level, elaborating first answer:

Question: In which areas within Household Surveys do you have experience?
First answer: Blaise Datamodelling – Question: How many years of experience do you have with Blaise Datamodelling? – Answer: 1 to 4 years
Second answer: Analysis – Q: How many years? – A: More than 10 years

Third level, elaborating second answer:

Question: In which areas within Population Statistics do you have experience?
Answer: Sample design – Q: How many years? – A: 5 to 9 years.

A varying number of answers can be given at any level.