Short Notes

Structure Clashes

The following paper comments on the changes which are needed to the work of Dromey and Chorvat, this Journal April 1990, to incorporate the concepts of Structured Programming

Received January 1991

1. Introduction

The article in this Journal[1] on Structure Clashes deserved some response.

Dromey and Chorvat have ignored the concept of structured programming. That this should happen 20 years after the theory was proposed, is lamentable. That this should happen to members of a 'Programming Methodology Research Group' is inexcusable.

Structured programming allows only three operations: process; selection; and a DO-loop. The DO-loop has to have any check performed at the start. This ensures that the check is passed before the loop is entered, and in turn gives a clean partitioning of code: you can only get into the loop by passing the check. Actions are easy to understand, and therefore change, because the actions are either inside or outside the loop.

In Dromey and Chorvat's examples, they should change their DO-UNTILs into DO-WHILEs. This would eliminate the checks they have within the loops to see if this is the first time around the loop.

Next, Dromey and Chorvat confuse physical and logical requirements. By 'physical' I refer to the input/output of data, while 'logical' refers to the processing of the input to generate the output. Keeping the two separate makes any program a lot simpler to understand. There is no loss in efficiency either!

Take the example '2.2. A Simple Structure Clash'. There is no problem once you create a routine that outputs the print lines.

The routine keeps track of linecounts and can therefore initiate pagebreaks when necessary. The print line is created at the point in the program where the program logic dictates the need to output the line. The print procedure worries about paging, etc., leaving the program logic unencumbered by printer logic.

Furthermore, the nesting of DO-loops that Jackson's program design gives is the ideal place to initialise headings and sub-headings.

And finally, I am surprised that the article was published as showing something new. Programmers have long been in the habit of using existing code to do something only 'slightly' different from the original purpose of the code. The results have been poor-quality systems that eventually become unmanageable: the process of modification never stops.

Structured programming was developed to ensure quality programs.

Jackson developed a technique that allows a program structure to be designed on the basis of the input/output data. With Jackson, program design became more of an engineering exercise than a 'black art'. Program inversion allowed for the lower processing power available in the 70s. It was an exceptional process then, and should be even more so now. I would expect a new method to offer significant advantages in some areas to warrant publication. Dromey and Chorvat have failed to demonstrate this.

However, to illustrate the above comments, I have reworked the problems used by Dromey and Chorvat. To demonstrate another fact, I have reworked the first problem using more descriptive names. Coding with a mixture of As, Ps and ms is a shortcut to unmanageable code.

For amusement I have enclosed the 'Jackson Rap'. I hope that this will cheer along the people who take the time and effort to review program requirements before making changes, and give pause for thought to those who do not.

There are also sample program structures in the Appendix. These have been designed using Jackson's techniques and serve as examples for those unfamiliar with the method. This allows people to create well-structured programs without having to attend courses and seminars.

2. Alternative Solutions

2.1 Buffer-Copy

In the problem, GETREC reads a record, IN_REC, directly into the buffer IN_REC_BUILD_UP. In some languages this may be a two-step process.

The size of IN_REC_BUILD_UP is always set to be larger than the output record, OUT_REC, and a multiple of the size of the records being read.

In essence, the GETREC command of Dromey and Chorvat changes from

read a record,

to

read records until you have more data than the length of the output record, or end-of-file.

```
IN_REC = nulls
IN_REC_BUILD_UP = nulls
IN_REC_BUILD_UP_LAST_POS = 0
OUT_REC = nulls
OUT_REC_SIZE = size of ''OUT_REC''

GETREC( IN_REC_BUILD_UP,
    IN_REC_BUILD_UP_LAST_POS,
    OUT_REC_SIZE )
DOWHILE IN_REC_LAST_POS > 0
  OUT_REC(1 → OUT_REC_SIZE )
      = IN_REC_BUILD_UP(
      1 → OUT_REC_SIZE)
  PUTREC ( OUT_REC )
  RESETIN    (    IN_REC_BUILD_UP,
      IN_REC_BUILD_UP_LAST_POS,
      OUT_REC_SIZE )
  GETREC(           IN_REC_BUILD_UP,
      IN_REC_BUILD_UP_LAST_POS,
      OUT_REC_SIZE )
END DOWHILE IN_REC_LAST_POS > 0
RESETIN: parm( IN_REC_BUILD_UP,
    IN_REC_BUILD_UP_LAST_POS,
    OUT_REC_SIZE )
/*  reset input to allow for  */
/*  records, written out.     */
/*  This involves moving the  */
/*  unwritten portion of the  */
/*  'in' record to the start of */
/*  the area 'in'. The next read */
/*  will then start adding data */
/*  into 'in' after the data  */
/*  just moved.               */
  J = IN_REC_BUILD_UP_LAST_POS
      - OUT_REC_SIZE
  IN_REC_BUILD_UP (1 to J)
      = IN_REC_BUILD_UP
      ((OUT_REC_SIZE+1 ) to
      IN_REC_BUILD_UP_LAST_POS ))
  IN_REC_BUILD_UP_LAST_POS = J
```

2.2 Simple Structure Clash

N.B. This is a standard program structure.

```
linecount_max = 70
linecount_max_for_course_heading
    = 65
linecount = linecount_max+1
pagecount = 0
eof_data = NO
getrec(data)
DOWHILE (eof_data = No )
  course_old = getrec_data.course
  course_stats = 0
  CALL COURSE_LINE_PRINT(
      course_heading )
DOWHILE    (getrec_data.course =
      course_old
      & eof_data = NO )
    course_stats =
      course_stats+getrec_data_stats
    report_line_data = getrec_data
    CALL REPORT_LINE_PRINT(
        report_line_data )
    getrec(data)
  END    /*DOWHILE    (getrec_data.
      course = course_old*/
  summary_report_line.count =
      course_stats
  CALL REPORT_LINE_PRINT(
      summary_report_line )
END /*DOWHILE (eof_data = NO )*/

COURSE_LINE_PRINT:
    parm( course_heading )
  IF linecount >
    linecount_max_for_course_heading
  THEN
    linecount = linecount_max+1
    CALL REPORT_LINE_PRINT(
        course_heading )
END COURSE_LINE_PRINT

REPORT_LINE_PRINT: parm(line)
  line_asa = char(1) of line
SELECT( line_asa )
   WHEN( new_page )
     linecount = linecount_max+1
   WHEN( doublespace)
     linecount = linecount+2
   WHEN( same_line )
     linecount = linecount
   OTHERWISE
     line_asa = new_line
     linecount = linecount+1
  END /*SELECT( line_asa )*/
  IF linecount > linecount_max
  THEN
     pagecount = pagecount+1
     putrec( heading_line_1)
     putrec( heading_line_2)
     line_asa = double_space
  putrec( line)
END REPORT_LINE_PRINT:
```

2.3 Multiple Input Structure Clash

This is the same solution as in the BUFFER-COPY problem. We ensure that any data read is sufficient to match (or exceed) the amount of data needing to be read out.

```
in_A = nulls
in_B = nulls
m = 0  /* last used position within
    'in_A' */
n = 0  /* last used position within
    'in_B' */
out = nulls
P = size of 'out'
GETREC( in_A, m, P )
GETREC( in_B, n, P )
DOWHILE m > 0|n > 0
  out( 1→P ) = in_A( 1→P)
      +in_B( 1→P )
  /* write out length 'N' from 'out'
      */
  PUTREC( out, P )
  RESETIN( in_A, m, P )
  RESETIN( in_B, n, P )
  GETREC( in_A, m, P )
  GETREC( in_B, n, P )
END DOWHILE
```

N.B. There will be some extra calls to in_A *or* in_B, *depending upon while file is read to end-of-file first. However, the overhead is small, and the program kept simple.*

Even if you wanted to add one record of in_A *to every one record of* in_B, *until you had sufficient data for 'out', this program is simple to modify.*

2.4 Text Formatting Problem

This is another standard program structure, and requires a two-step process.

```
IN_REC = nulls
GETREC( IN_REC )
DOWHILE ( IN_REC found )
/* write each individual word from
    IN_REC */
  last_word_found_pos = 0
  FIND_WORD( IN_REC,
      last_word_found_pos )
  DO_WHILE  last_word_found_pos <
      IN_REC_RECORD_LENGTH
    PUTREC( word_length, word )
    FIND_WORD( IN_REC,
        last_word_found_pos )
  END /*DO_WHILE*/1
  GETREC( IN_REC )
END DOWHILE ( IN_REC found )
```

The downstream program now has a simple format.

```
w = 0  /* 'word' record count */
GETREC( word, w)
DOWHILE w > 0
  out_length = 0
  DOWHILE (out_length <
      out_length_max)
      & w > 0
    LOAD_WORD(word, out)
    GETREC( word, w)
    out_length =
        out_length+word_length
  END
  PUTREC( out )
END DOWHILE w > 0
```

The above format allows a very flexible approach to the problem. The solution allows additional features, such as edit checking, to be added very easily.

Lu LAHODYNSKYJ
103, Martin Grove Road, Islington, Ontario M9B 4K6, Canada.

Reference

1. R. G. Dromey and T. A. Chorvat, Structure clashes – an alternative to program inversion. *The Computer Journal*, **33** (2), 126–132 (1990).

APPENDIX

A.1 The Jackson Rap (or, how to approach program maintenance)

I WAS READING A PROGRAM JUST THE OTHER DAY,
THAT KIND OF BLEW MY MIND AWAY.
IT WAS WRITTEN QUITE SOME TIME AGO
BY SOMEONE ELSE, NOT, A JACKSON BRO.

I READ IT UP AND I READ IT DOWN,
IT WAS GOODBYE SMILE AND HELLO FROWN.
SO I SAID TO MYSELF, 'OH WHAT THE HECK'
AND STARTED TO WRITE A PROGRAM SPEC.

WHEN I'D DONE IT ALL, IT WAS PRETTY GOOD,
IT TOLD YOU ALL THAT A GOOD SPEC SHOULD.
IT TOLD YOU WHAT WENT IN AND OUT,
TOLD YOU WHAT THE PROGRAM WAS ALL ABOUT.

SO I ADDED A PLAN ON HOW TO TEST,
I KNEW I WAS DOING MY VERY BEST.
ONE MORE THING, WHILE I WAS DOING SO WELL,
I HAD TO WRITE SOME PDL.

THEN I SAID WHO000–0000–0000–0000,
I'M FORGETTING I'M A JACKSON BRO.
SOME DIAGRAMS I HAD TO WRITE,
TO MAKE QUITE SURE THE STRUCTURE WAS RIGHT.

SO_WHEN_THE_CODING CAME, IT WAS ALL A BREEZE,
CLEAN, STRUCTURED (AND NO DISEASE).
THE WALKTHROUGHS, THE TESTING, PRODUCTION TOO,
CAME AND WENT LIKE ZIPPIDY-DO.

SO YOU'RE GETTING THE MESSAGE, IF YOU ARE HIP,
THIS PROGRAM CHANGING, CAN BE QUITE A TRIP.
BUT BEFORE YOU GET TO PL/1
MAKE SURE YOUR THINKING HAS ALL BEEN DONE!

DOWHILE, DOWHILE, DOWHILE, SELECT
DOWHILE, DOWHILE, DOWHILE, SELECT
DOWHILE, DOWHILE, DOWHILE, SELECT
DOWHILE, DOWHILE, DOWHILE, etc....

A.2 PDL – Program Design Language

The following Program Design Languages (PDLs) are program structures that were generated by using Jackson's techniques and have stood the test of time. They allow you to set your program structure quickly. You will only need to create a new PDL if the data manipulation does not fall into one of the categories below. In this case you have to return to the basics of Jackson program design.

N.B. Should you have become used to the use of 'IF' statements to control your logic, these will seem somewhat cumbersome, at first. However, there are three things to remembers.

(1) They work – guaranteed.
(2) The code is easy to read.
(3) Changes are easy to implement.

A.2.2 Report from One File

This is a sample PDL for a simple read/print program. It uses an extract file with Buyer#, Expeditor# and Vendor# in the fields: records are sorted by Buyer/Expeditor/Vendor.

The report gives a new page for each change in Buyer# or Expeditor# and produces a summary of vendor information, for each expeditor.

```
OPEN all files
Set all record counts to zero
READ EXTRACT record
DOWHILE (EXTRACT_EOF = NO )
  BUYER#_OLD = EXTRACT_REC.BUYER#
  EXPED#_OLD = EXTRACT_REC.EXPED#
  CALL NEW_PAGE_PRINT
  DOWHILE   (EXTRACT_REC.BUYER# =
      BUYER#_OLD
      & EXTRACT_REC.EXPED#
       = EXPED#_OLD
      & EXTRACT_EOF = NO )
    VENDOR#_OLD =
        EXTRACT_REC.VENDOR#
    DOWHILE (EXTRACT_REC.VENDOR# =
        VENDOR#_OLD
        & EXTRACT_ REC.BUYER#
        = BUYER#_OLD
        & EXTRACT_REC.EXPED# =
        EXPED#_OLD
        & EXTRACT_EOF = NO )
  CALL REPORT_LINE_PRINT
  READ EXTRACT record
    END  /*DOWHILE  (EXTRACT_REC
      .VENDOR# = VENDOR#_OLD*/
    CALL VENDOR_SUMMARY_PRINT
  END  /*DOWHILE   (EXTRACT_REC.
      BUYER# = BUYER#_OLD*/
END /*DOWHILE (EXTRACT_EOF = NO*/
WRITE record counts
CLOSE all files
```

A.2.3 Two-file Match

This is a sample PDL for a simple two-file match program. It uses a master file that needs to be revised by data from an update file. Keys are unique for both the master and update files. The resulting records are written to a new master file.

N.B. The end-of-file situation is used to set the record keys to high values. This ensures that both inputs will be read to the end.

```
OPEN all files
Set all record counts to zero
READ MASTER_OLD record
READ UPDATE record
DOWHILE (MASTER_OLD.KEY .LT. HIGH
    .OR. UPDATE.KEY .LT. HIGH )
  SELECT
/* no update for old master */
    WHEN (UPDATE.KEY .GT.
        MASTER_OLD.KEY)
        WRITE  MASTER_OLD  to  new
            master file
        Increment  record  copied
            count
        READ MASTER_OLD record
/* record not on old master */
    WHEN (UPDATE.KEY .LT.
        MASTER_OLD.KEY)
        Create  new  Master  record
            from UPDATE
        WRITE  MASTER_NEW  to  new
            master file
        Increment  record  created
            count
        READ UPDATE record
/* records match */
```

```
    OTHERWISE
        Update   MASTER_OLD   from
            UPDATE
        WRITE  MASTER_OLD  to  new
            master file
        Increment  record  updated
            count
        READ UPDATE record
        READ MASTER_OLD record
END /*DOWHILE (MASTER_OLD.KEY .LT.
    HIGH*/
WRITE record counts
CLOSE all files
```

A.2.4 Two-file match, one with batches

This is a sample PDL for a two-file match program with one file in batches. It uses a master file that needs to be revised by data from an update file. Unique keys exist in the master file. The update file can have more than one occurrence of a key, all of which have to be applied to the one master file record. The resulting record is written to the new master file.

N.B. The end-of-file situation is used to set the record keys to high values. This ensures that both inputs are processed correctly when one end of file is encountered before another. This logic uses a switch to indicate when a file is empty; this will be more efficient and still give readable code, rather than checking each key.

```
OPEN all files
Set all record counts to zero
READ MASTER_OLD_record
READ UPDATE record
DOWHILE  (MASTER_OLD_EOF = NO  .OR.
    UPDATE_EOF = NO )
  SELECT
/* no updates for old master */
      WHEN (UPDATE.KEY .GT.
          MASTER_OLD.KEY)
        WRITE  MASTER_OLD  to  new
            master file
        Increment  record  copied
            count
        READ MASTER_OLD record
/* record not on old master */
      WHEN (UPDATE.KEY .LT.
          MASTER_OLD_KEY)
      KEY.SAVE = UPDATE.KEY
      STORE_UPDATE = null
      DO WHILE ( KEY_SAVE =
          UPDATE. KEY )
      STORE_UPDATE = STORE_
          UPDATE+UPDATE, by name
      READ UPDATE record
      END /*DO WHILE ( KEY_SAVE =
          UPDATE.KEY )*/
      MASTER_NEW =
          MASTER_NEW+STORE_UPDATE,
          by name
      WRITE MASTER_NEW to new master
          file
      Increment  record  created
          count
/* records match */
OTHERWISE
      key_save = update.key
      STORE_UPDATE = null
      DO WHILE ( KEY_SAVE =
          UPDATE.KEY )
      STORE_UPDATE =
            STORE_UPDATE
            +UPDATE, by name
        READ UPDATE record
    END  /*DO  WHILE  (  KEY_SAVE =
        UPDATE.KEY )*/
    MASTER_NEW = MASTER_OLD+
        STORE_UPDATE, by name
    WRITE MASTER_NEW to new master
        file
    Increment record updated count
    READ MASTER_OLD record
END  /*DOWHILE  (MASTER_OLD_EOF =
    NO*/
WRITE record counts
CLOSE all files
```

A.2.5 Three-file Match

This is a sample PDL for a simple three-file match program. It uses three input files to produce a report. There is a unique KEY value, on each file, though each KEY value may exist on all files.

```
OPEN all files
Set all record counts to zero
READ FILE_01 record
READ FILE_02 record
READ FILE_03 record
DOWHILE  (  End-of-file(FILE_01) =
    NO .
    OR. End-of-file(FILE_02) = NO .
    OR. End-of-file(FILE_03) = NO )
  SELECT on key matches
    WHEN(REC_01.KEY = REC_02.KEY  &
        REC_01.KEY = REC_03.KEY)
      WRITE Message for all  keys
          matching
      READ FILE_01 record
      READ FILE_02 record
      READ FILE_03 record
      Increment CNT_MATCH_ALL
    WHEN( REC_01.KEY = REC_02.KEY &
      REC_01.KEY .LT.REC_03.KEY )
      WRITE   Message   for   not
          matched on #3
      READ FILE_01 record
      READ FILE_02 record
      Increment CNT_MATCH_1_2
    WHEN( REC_01.KEY = REC_03.KEY &
      REC_01.KEY .LT. REC_02.KEY )
      WRITE   Message   for   not
          matched on #2
      READ FILE_01 record
      READ FILE_03 record
      Increment CNT_MATCH_1_3
    WHEN( REC_01.KEY .GT. REC_02.KEY
      & REC_02.KEY = REC_03.KEY )
      WRITE   Message   for   not
          matched on #1
      READ FILE_02 record
      READ FILE_03 record
      Increment CNT_MATCH_2_3
    WHEN( REC.01.KEY .LT. REC_02.KEY
      &REC_01.KEY.LT.REC_03.KEY)
      WRITE Message for REC_01.KEY
          only on #1
      READ FILE_01 record
      Increment CNT_SCALE_123
    WHEN( REC_02.KEY .LT. REC_01.KEY
      & REC_02.KEY .LT. REC_03.KEY
      )
      WRITE Message for REC_01.KEY
          only on #2
      READ FILE_02 record
      Increment CNT_SCALE_213
    OTHERWISE
      WRITE Messages for REC_01.
          KEY only on #3
      READ FILE_03 record
      Increment CNT_SCALE_312
END /*DOWHILE (KEY's .LT.HIGH)*/
WRITE record counts
CLOSE all files
```

N.B. The end-of-file situation sets the record keys to high values. This ensures that both inputs are processed correctly when one end of file is encountered before another. This logic uses a switch to indicate when a file is empty this is more efficient and still gives readable code, rather than checking each key. Record counts are assumed to be kept on each file READ.