Abstract:
The construction of a program to solve a simple problem, written using a top-down structural approach, is described. An independent analysis of this program is provided commenting on the possible problems that arise from the use of such a technique.

An Experiment in Structured Programming

by

P. Henderson and R. Snowdon

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

© 1971 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
HENDERSON, Peter

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1971.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 18)

12"

Added entries

SNOWDON, Robert Archer
UNIVERSITY OF NEWCASTLE UPON TYNE. Computing Laboratory. Technical Report Series. 18

Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th):
U.D.C.: 510.7834
681.322.06

Suggested keywords

STRUCTURED PROGRAMMING
PROGRAM CORRECTNESS

Abstract

The construction of a program to solve a simple problem, written using a top-down structural approach, is described. An independent analysis of this program is provided commenting on the possible problems that arise from the use of such a technique.

About the Author

Mr. Henderson is a Demonstrator in the Computing Laboratory of the University of Newcastle upon Tyne. He has recently been awarded the degree of Doctor of Philosophy for work in Formal Programming Language Semantics.

Mr. Snowdon is a postgraduate research student in the Computing Laboratory of the University of Newcastle upon Tyne.
1. Introduction

In the first section of this report a program is developed using a 'top-down' structural approach with the hope that the program can be seen to be correct by its very structure. The method of construction is to write a program using concepts that are relevant to the problem, even though they would not be understood directly by the machine on which the program will be run. These concepts are then successively elaborated until they are expressed in a machine understandable language.

This technique has been reported by Dijkstra (1970) and the notation used here, although self-explanatory, is based on this work. The second section is devoted to an analysis of the program with the idea of learning a few lessons about programming using such a technique.

The programmer (Henderson) had developed the program in order to demonstrate to students an approach to problem solving. The program was thus developed before this experiment was contemplated. Further it was the programmer's first attempt at this problem, although he is its originator, and his first experience with structured programming.

It was not until after the program had been written that the co-author (Snowdon) became interested in analysing it. His main interest was in studying the way other people program and this offered an ideal opportunity for him. The experiment reported here is probably not unique, but is interesting in that an independent and detailed assessment of a structured program is provided.

2.1. The Problem

A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer area where it is to be processed. The words in the telegrams are separated by sequences of blanks and each telegram is delimited by the word 'ZZZZ'. The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words 'ZZZZ' and 'STOP' are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word.
2.2. The Solution

With a little thought, the basic structure of the required program is quite obvious.

PROGRAM: (INITIALISE FOR WHOLE PROGRAM;

\textbf{repeat} \text{INITIALISE FOR NEW TELEGRAM;}

\text{PROCESS TELEGRAM}

\textbf{until} \text{EMPTY TELEGRAM})

Each telegram up to and including the last is to be processed by PROCESS TELEGRAM. The repetition is concluded only when the empty telegram has been processed. The separated process INITIALISE FOR NEW TELEGRAM, which can be considered a logical part of the processing required for each telegram is intended to cater for the, as yet unspecified, initial conditions required by PROCESS TELEGRAM. A similar situation arises in respect of INITIALISE FOR WHOLE PROGRAM.

It is the objective of PROCESS TELEGRAM to accumulate the required information about the words in each telegram and to print out this information. This process is conveniently elaborated to separate the activities associated with each word and the results associated with the telegram as a whole:

PROCESS TELEGRAM: (COUNT CHECK AND PRINT WORDS;
PRINT WORD COUNT AND CHECK MESSAGE)

The decision to print each word as it is processed stems from a recognition of the sequential nature of the required processing. Once a word has been counted, checked for length and printed it is no longer required by the processor. Had we decided to separate the counting and checking from the printing, it would have been necessary to store the telegram as a whole (note that the telegram may be in considerable excess of the buffer size so can in no way be considered to be in the processor's store).

The process COUNT CHECK AND PRINT WORDS must inspect each word, by removing it from the buffer. The word can then be subjected to the necessary checking and printed. The process must obviously iterate upon this activity until it has inspected the word 'ZZZZ', and takes the form: COUNT CHECK AND PRINT WORDS:

\textbf{repeat} \text{EXTRACT WORD;}

\textbf{if} \text{WORD IS CHARGEABLE \textbf{then} \text{COUNT WORD;}}

\textbf{if} \text{WORD IS TOO LONG \textbf{then} \text{SET CHECK FLAG;}}

\text{PRINT WORD}

\textbf{until} \text{WORD IS ZZZZ}
The subprocess WORD IS CHARGEABLE tests for occurrences of 'STOP' and 'ZZZZ', while WORD IS TOO LONG simply tests the length of the word constructed by EXTRACT WORD. At this stage the requirement arises for the introduction of variables to hold the word count and the flag indicating the occurrence of overlength words. The processes COUNT WORD and SET CHECK FLAG update the variables in an appropriate way. Here we note that the refinement of INITIALISE FOR NEW TELEGRAM necessarily gives these two variables initial values, since they are quantities local to each telegram.

Considering how the characters are constructed to form words we encounter a process similar to the way words are constructed to form telegrams. We must repeatedly extract letters from the buffer until we encounter a blank (space). Redundant blanks preceding a word must first be removed from the buffer. A suitable elaboration of EXTRACT WORD is then:

```
EXTRACT WORD: (SET WORD EMPTY INITIALLY;
    ADJUST INPUT;
     repeat EXTRACT LETTER
     until LETTER IS SPACE)
```

The process COUNT CHECK AND PRINT WORDS introduces another variable, required to hold each word as it is constructed. The subprocess SET WORD EMPTY INITIALLY simply gives this variable an initial value. Redundant blanks occurring between words are removed by the subprocess ADJUST INPUT. This guarantees that the first character is removed from the buffer and stored in the variable by EXTRACT LETTER. The process continues until a blank is transferred. Note that this means the value of the variable includes the delimiting blank.

The remaining subprocess of COUNT CHECK AND PRINT WORDS has still to be elaborated. The objective of PRINT WORD is to transfer the words with which it is presented to some output medium in lines of predetermined length. For this purpose it requires a variable in which to construct a line of printing. As it receives each word it then appends it to the current value of this variable until this value has a printable length. It then prints the line. Another reason for printing the line would be because the transferred word concludes a telegram, that is to say it is 'ZZZZ'. The elaboration of PRINT WORD is therefore:

```
PRINT WORD: (APPEND WORD TO LINE;
    if LINE IS FULL OR WORD IS ZZZZ then PRINT LINE)
```

The variable used to hold each line of printing is local to each telegram and can therefore be initialised by INITIALISE FOR NEW TELEGRAM. When the line has been printed by PRINT LINE the variable must be set to empty again.

5.
Returning to the process EXTRACT WORD we must elaborate EXTRACT LETTER and ADJUST INPUT. The buffer area to which the characters are transferred from the device will be referred to in the elaboration as the input string. The objective of ADJUST INPUT is simply to remove leading blanks. It must therefore inspect the buffer for emptiness and if it is not empty, inspect the leading character. This character is removed if it is blank and in this event the process is repeated. The elaboration is:

ADJUST INPUT:

while INPUTSTRING NOT EMPTY AND LEADING SPACES ON INPUT do
    REMOVE FIRST LETTER FROM INPUT STRING

The process EXTRACT LETTER must inspect the buffer to select the next available character. If the buffer is empty it must demand more characters from the input device. Finally EXTRACT LETTER must transfer the character it obtains to the variable in which it is constructing each word, using an additional variable to effect this transfer:

EXTRACT LETTER:

(if INPUT STRING IS EMPTY then DEMAND MORE INPUT;

    SELECT FIRST LETTER OF INPUT STRING;
    REMOVE FIRST LETTER FROM INPUT STRING;
    PUT LETTER ON END OF WORD)

When the subprocess DEMAND MORE INPUT is invoked it transfers characters from the device to the buffer. Expecting the unit of input to be a complete sequence of words it then appends a trailing blank to delimit the last of these words:

DEMAND MORE INPUT:

(TRANSFER CHARACTERS FROM DEVICE TO INPUT STRING;

    APPEND A TRAILING BLANK)

The subprocess INITIALISE FOR WHOLE TELEGRAM would then initially establish an empty buffer.

It is worth noting at this point that the structure of the program we have obtained, reflects directly the structure of the data it processes (there is a sense in which it defines the structure of the data). A word is a sequence of one or more letters and digits terminated by a single blank. A telegram is a sequence of zero or more words, separated by zero or
more blanks and terminated by the word 'ZZZZ'. A stream is a sequence of zero or more telegrams terminated by an occurrence of the empty telegram.

The following program is the final refinement of the structure described above in a language which is hoped to be self-explanatory:

```plaintext
integer wordcount; logical checkflag; string input, word, letter, line;
PROGRAM: ( input: = emptystring;
  repeat
    ( wordcount: = 0; checkflag: = false;
      line: = emptystring);
  PROCESS TELEGRAM:( COUNT CHECK AND PRINT WORDS;

  repeat
    EXTRACT WORD: ( word: = emptystring;

    ADJUST INPUT: while input ≠ emptystring and first (input)= ' ' do input: = rest (input);

    repeat
      EXTRACT LETTER: ( if input = emptystring
        then input: = read + ' ';
        letter: = first (input); input:= rest (input);
        word: = word + letter)
        until letter = ' ');

      if ¬ (word = 'STOP' or word = 'ZZZZ')
        then wordcount: = wordcount + 1;

      if length (word) >12 then checkflag: = true;

    PRINTWORD: ( line: = line + word;

      if length (line) >100 or word = 'ZZZZ'
      then (print (line); line: = emptystring))
      until word = 'ZZZZ ');

  PRINT WORDCOUNT AND CHECK MESSAGE:

    ( print (wordcount); if checkflag then print ('CHECK'));
    until wordcount = 0)
```

At this stage the author's hope that the reader has been convinced by the structural approach, that a working program has been constructed.
Table 1.

1. PROGRAM
   1.1 INITIALISE FOR WHOLE PROGRAM
   1.2 INITIALISE FOR NEW TELEGRAM
   1.3 PROCESS TELEGRAM
   1.4 EMPTY TELEGRAM

2. PROCESS TELEGRAMS
   2.1 COUNT CHECK AND PRINT WORDS
   2.2 PRINT WORD COUNT AND CHECK MESSAGE

3. COUNT CHECK AND PRINT WORDS
   3.1 EXTRACT WORD
   3.2 WORD IS CHARGEABLE
   3.3 COUNT WORD
   3.4 WORD IS TOO LONG
   3.5 SET CHECK FLAG
   3.6 PRINT WORD

4. EXTRACT WORD
   4.1 SET WORD EMPTY INITIALLY
   4.2 ADJUST INPUT
   4.3 EXTRACT LETTER
   4.4 LETTER IS SPACE

5. PRINT WORD
   5.1 APPEND WORD TO LINE
   5.2 LINE IS FULL OR WORD IS ZZZZ
   5.3 PRINT LINE

6. ADJUST INPUT
   6.1 INPUT STRING NOT EMPTY AND LEADING SPACES ON INPUT
   6.2 REMOVE FIRST LETTER FROM INPUT STRING

7. EXTRACT LETTER
   7.1 INPUT STRING IS EMPTY
   7.2 DEMAND MORE INPUT
   7.3 SELECT FIRST LETTER OF INPUT STRING
   7.4 REMOVE FIRST LETTER FROM INPUT STRING
   7.5 PUT LETTER ON END OF WORD

8. DEMAND MORE INPUT
   8.1 TRANSFER CHARACTERS FROM DEVICE TO INPUT STRING
   8.2 APPEND A TRAILING BLANK
As the program is constructed, so the tree will grow. The form of the tree indicates how particular elaborations were made and how concepts fit together within the overall structure. This is true at any time, and so even an incomplete tree provides information on the state of the program as it is being written. We may draw a horizontal line through the tree at any point and then the tree above this level will represent the hierarchy of the concepts used to give a solution to the problem. If we also add the relevant sequencing primitives to this tree we may obtain a solution to the problem, not necessarily in a form understandable by a computer. The main gain from looking at the tree is that it summarises the actual construction in a concise manner.

3.1 Analysis of the program

We have discovered, by running it on a computer, that the program does not work properly. Yet the entire motivation for writing the program in the above manner was to ensure its correctness from the start. What can we learn, therefore, from the investigation of the error and the decisions taken during the program construction.

3.1.1 Node by node investigation

As noted above, if we draw a line through the tree at any level, then, with knowledge of the sequencing we can interpret the structure to provide a program to solve the problem. Thus we may draw a line beneath the first node and obtain the solution:

INITIALISE FOR WHOLE PROGRAM:
repeat INITIALISE FOR NEW TELEGRAM;
PROCESS TELEGRAM
until EMPTY TELEGRAM

Now assuming that the meaning of each process is what is meant by the sequence of words used to denote it, taken as an English sentence, then this program is 'correct'. We will see how important it is to know what is meant by, "knowing what the English means".

A first point to notice at this level is that there has been no explicit introduction of any data structures. The idea of a telegram has been left as an idea, and again it will appear that such a recourse will not encourage the clarity of understanding of some particular concept or its subsequent elaboration.

10.
At this level in the tree we have what we can see as a 'correct' program. We must thus continue the investigation by looking further down the tree. From this present position there are two subtrees worthy of discussion. If we look at INITIALISE FOR NEW TELEGRAM we find that a telegram is to have two attributes; a word count and a check flag and that after the operation INITIALISE FOR NEW TELEGRAM, they will have the values '0' and 'false' respectively. Thus subsequent elaboration of PROCESS TELEGRAM will be able to depend on these values. This subsequent elaboration is the most interesting subtree to look at and so we will follow it further. The elaboration is:

COUNT CHECK AND PRINT WORDS;
PRINT WORD COUNT AND CHECK MESSAGE

This would seem to be a 'correct' elaboration of PROCESS TELEGRAM, without knowing how a telegram is structured. Our program is still 'correct' and the error not located, so we proceed to the subtree indicated by COUNT CHECK AND PRINT WORDS. This is still the main processing part of the program and it is here that most of the interest lies. Immediately the problem of implicit data structures returns with the introduction of a 'word'. The elaboration is

repeat EXTRACT WORD:
   if WORD IS CHARGEABLE then COUNTWORD;
   if WORD IS TOO LONG then SET CHECKFLAG;
   PRINT WORD;
until WORD IS ZZZZ;

It is worth noting that before PROCESS TELEGRAM is performed the condition

wordcount = 0 and ¬ checkflag

is true. By tracing down the tree (with the listing of each elaboration), it can be seen that this is still true on entry to COUNT CHECK AND PRINT WORDS. It will be seen that knowledge of such conditions is important to maintaining the correctness of the elaboration of a concept - more particularly the conditions that hold after an operation has been performed.

It was also noted earlier that it was important that the meaning of the English was well understood, so that the correctness can be established both at that particular elaboration, and also within it when concepts inside it are elaborated. As elaborations become more complex this becomes more important, as does the realisation of the interaction between concepts within the particular elaboration.
We return now to elaboration of COUNT CHECK AND PRINT WORDS. This elaboration is correct with the usual implications of the meanings, noting that a word has been introduced as a data concept, but nothing said about its structure.

We continue by investigating EXTRACT WORD, as the remainder is reasonably straightforward. EXTRACT WORD has the purpose of providing a word which can be checked. Thus there are no conditions on its action but the result is specified. We have, for EXTRACT WORD:

```
SET WORD EMPTY INITIALLY;
ADJUST INPUT;
repeat EXTRACT LETTER
until LETTER IS SPACE;
```

There are a number of points to be made here. The idea of an input string has been introduced prematurely by INITIALISE FOR NEW TELEGRAM. Thus it has to be brought into consideration at places where it becomes a nuisance to understanding. At this point we are elaborating EXTRACT WORD and it has been decided (implicitly) that a word shall consist of a sequence of one or more letters terminating in a space. The need to worry, at this level, about the input buffer clouds the issue. However, with the program written as it is, it has to be considered. So the elaboration of EXTRACT WORD is 'correct'.

### 3.1.2 Location of the error

Details now become important. A variable 'word' is set initially to empty and so after SET WORD EMPTY INITIALLY the condition

\[ \text{word = emptystring} \]

is true. Next the input string is adjusted to remove any spaces from the beginning so that the first letter is the first letter of a word. Thus we have after ADJUST INPUT that the condition

\[ \text{first letter of input = space} \quad (1) \]

is true.

Our acceptance of these conditions as being what is meant by the pieces of program leads to the conclusion that the elaboration of EXTRACT WORD is 'correct'. However, if we now look at the elaboration of ADJUST INPUT then we see that the condition (1) does not always hold but the condition

\[ \text{first letter of input} \neq \text{space or input = emptystring} \quad (2) \]

holds. Thus the building of a word by

```
repeat EXTRACT LETTER
until LETTER IS SPACE
```

## 12.
will have to take account of this weaker condition and the meanings implied by the elaboration of EXTRACT WORD become a little misleading.

This is the crucial point. At the level of EXTRACT WORD the programmer conceived the process of building a word as taking place in the context of the stronger condition (1). Presumably at some point in the elaboration of the subprocess of EXTRACT WORD, the weaker condition (2) was implicitly realised. The natural reaction of the programmer was to adjust to this condition and continue (the familiar, all embracing "patch").

Such a technique is entirely against the idea underlying the method of construction. At the level of EXTRACT WORD the program can be read and be seen to the 'correct'. Below this level things will become less obvious and, because of the clarity of the structure above, misleading.

Having accepted the less-than-complete elaboration of ADJUST INPUT, the elaboration of EXTRACT LETTER has to take into account something that logically has no place there. When the elaboration of EXTRACT WORD is read the natural meaning inferred is that first the input is adjusted so that a word starts in the first character position, and then the word is built up letter by letter until the terminating space is recognised. The important lesson to be learnt is that of knowing exactly what is meant by a particular concept, so that subsequent elaboration fulfills the meaning exactly.

The error is found now as the elaboration of EXTRACT LETTER (and subsequently of DEMAND MORE INPUT) shows:

```
if INPUT STRING IS EMPTY then DEMAND MORE INPUT;
SELECT FIRST LETTER OF INPUT STRING;
REMOVE FIRST LETTER FROM INPUT STRING;
PUT LETTER ON END OF WORD;
```

The conditional statement at the beginning is designed to take care of the weakness of the elaboration of ADJUST INPUT. The remainder adds another letter to the partially found word and removes the letter from the input string. If after DEMAND MORE INPUT the condition

```first letter of input ≠ space```

was true, then this condition would hold after the conditional statement. The error occurs because the elaboration of DEMAND MORE INPUT does not fulfill this condition.
3.1.3 Origins of the error

Although the actual error occurs in DEMAND MORE INPUT its origins can be traced back through the structure. If ADJUST INPUT had fulfilled its obligations in establishing condition (1) then everything would have been acceptable. The error can be traced back even further however. Elaboration of the process EXTRACT WORD should not be concerned with the details of where the characters in the word originate. Rather this concern should be deferred until the characters are handled directly. Because of the premature introduction of the input string more detail had to be carried through the tree than was necessary and so eventually the error was made.

That the error was made when a machine understandable form had almost been reached brings in an interesting point. If the gap between the machine-understandable program and the program expressed at the level of primitive concepts is too large, then errors will occur in bridging it. The technique is aimed at taking small steps toward a solution; to have to finally make a large jump is defeating the object. The programming language, and the programmer's experience with it, will tempt him to try too complex final elaborations. This suggests that the programming language should be a subset of the language used to express the elaborations during construction. In this case we may regard the basic operations and data structures of the programming language as concepts that require no further elaboration and hence maintain the uniformity of the complete structure.

3.2 Summary of results

Such a top down technique is a logical way of constructing a program. It does however have its pitfalls as the work reported here has shown. We describe these under the following headings.

(i) Meanings

Because so much is written down in descriptive, rather than formal notation, there must be some way of attaching meanings to the descriptions. To achieve this obviously requires something similar to the 'conditions' we have used here (cf Floyd 1967), but the elements involved in the conditions will of necessity also be of a conceptual nature. Additionally it is important that the elaboration of a concept is correct in that it fulfills precisely the meaning as specified by the defining condition (i.e. the elaboration should have no side effects). In particular we should require that if some invariance condition (Hoare 1970) is applied over some operation at a particular level, then
this invariance must also hold over any subsequent elaboration of that operation.

(ii) Data Structures

Data structures should be explicitly stated and a data concept should be elaborated in the same manner as any other concept. In other words data structures should appear in some form analogous to a declaration. The naming of a data type is an explicit enough introduction for that type and makes the programmer appreciate its importance a little more. Elaboration of a data concept will have effects on other concepts referring to that data concept. (e.g. the concept of TELEGRAM and the idea of PROCESS TELEGRAM). Thus the further one can proceed before elaborating a data concept the better the understanding of what that data structure should look like. This also means that there should be fewer concepts to be considered in any particular elaboration (e.g. in EXTRACT WORD hopefully we would not need to worry about input), and hence a clearer understanding of what is meant. Thus the order of what to refine next is important (what Dijkstra calls 'judicious postponement' (Dijkstra 1970)).

(iii) Simplicity

Although no specific mention has been made of this so far, it can be argued that maybe the technique appears too simple. Because of its simplicity the programmer is lulled into a false sense of security. Other work in this field (cf. WIRTH 1970) has tended to emphasize this feeling by the apparent ease with which it achieves solutions to examples. The work in our report suggests that, whatever the technique used, programming remains a difficult task, and the programmer must not be led to believe otherwise.

4. CONCLUSIONS

Consideration of all the points raised draws one to the conclusion that in such a technique we must apply formal methods. The importance of data structures, the meaning of concepts and correct elaboration should be brought out and indicated in some well chosen manner.

If the programmer had been more formal about the assignment of meanings to the concepts he introduced it could be expected that the inconsistency between these formal meanings would become obvious at the level at which they occur. Thus an error is obvious as it is made; correction rather than patching becomes the simpler task.
It is the patching of partially correct programs that makes them obscure. The nature of formal aids will be dependent upon such things as the problem area, the programming methodology and the programming environment.

The elaboration of requisite data structures should be as explicit as the elaboration of the sequence of operations in a program. A programming language should therefore provide facilities for the conception of abstract data structures and their subsequent elaboration.

Finally in order to ensure correct elaboration, the techniques used to elaborate concepts should be well understood. A limited but well-chosen set of such techniques is desirable. Teitelman (1970) recommends automatic aids to programming. His work describes a step in the direction of a programming laboratory using a system called PILOT. This system acts as an interface between the user and his programs, monitoring the requests of both. For example it can be told to take some specified action if an error is detected. However PILOT does not help in the static construction of programs, but only when they are running. This report indicates that such facilities should reflect programming behaviour, although it is not suggested that these facilities be directly based upon the techniques applied here.

References

DIJKSTRA (1970): EWD249 'Notes on Structured Programming'

FLOYD (1967): 'Assigning meanings to Programs'

HOARE (1970): 'Proof of a program: Find'
Privately circulated.


WIRTH (1970): 'Programming and Programming Languages'