stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 78/77

AUGUSTUS

L.G.L.T. MEERTENS

PROGRAM TEXT AND PROGRAM STRUCTURE

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A30

ACM-Computing Review Categories: 4.20

Program text and program structure[*]

by

L.G.L.T. Meertens

ABSTRACT

Even if a program has been developed in a structured way, its structure need not be reflected in the final program text. Modifications, which are an important part of programming, should be applied at the highest appropriate level of abstraction and worked out downward from there. It is hard to enforce this discipline if a low-level program text is available: such a text lacks the proper structure and thus invites patches. This implies that the program text, as kept around, should reflect the structure of program development. A language feature is proposed for making the structure explicit that is introduced by the method of stepwise refinement.

KEY WORDS & PHRASES: structured programming
stepwise refinement
modifiability
documentation

[*] This report is a preprint of a paper that will appear in the Proceedings of the IFIP TC 2 Working Conference on Constructing Quality Software, Novosibirsk, 22 - 27 May 1977.

0. INTRODUCTION

The deplorable quality of many software products is doubtlessly related to the complexity of the task of designing and developing programs. We have witnessed during the last ten years the emergence of a methodology to reduce such complexity by using structure. And yet, in spite of a widespread acceptance of the ideas involved, it appears that the general situation has not substantially improved. It may be that it is too early to notice the effects, but my assessment is different. The method of structured programming is much older than the notion: it is as old as the computer field. Generations of programmers have made use of structuring techniques without being aware that they were doing so. The merit of the prophets of Structured Programming is that they have described the method explicitly, have created the terminology for discussing the design process and have thereby done the groundwork for turning the art into a discipline. But it is one thing to accept a discipline, and another thing to apply it. Most larger software products are developed by teams that are under some kind of pressure to present tangible results, such as an "operational" system (for which it may suffice that it ostensibly does something, though nobody knows what and there is no way to find out).

For a radical improvement we need more than a method or discipline: we need tools with which the willing spirit can enforce the discipline upon his weak flesh. To be sure, already many such tools exist: in fact, each high-level programming language is one to some extent. But it is reasonable to expect that a tool will provide an appropriate support for a given task only if it has explicitly been designed for that purpose. What we need are programming languages that have been designed with an explicit model of the program construction process in mind.

1. STRUCTURED PROGRAMMING

The term "structured programming" refers to the _process_ of program construction, rather than to any intrinsic property of program texts constructed in that fashion. It is theoretically possible - though highly unlikely - that a programmer who constructs his program in a most unsystematic way, eventually produces a program text identical to one developed by a colleague in the most structured way possible.

When I use the word "program", it means a program text _together with_ the structure implied by the way in which the text was constructed.

The task of a programmer is to bridge a gap between a top and a bottom: at the top he has a very abstract algorithm ("solve this problem") and at the bottom he has a collection of concrete types and operations supported by his machinery. His task is to implement the top algorithm in terms of the bottom operations. Two important principles he may use to bring structure in this task are:

(a) _Layers of abstraction_ (Dijkstra [1]): Design a collection of data types with corresponding operations, and possibly some other operations. This collection describes an interface which factors the original task into two new tasks: (i) implement the top algorithm in terms of the operations of the interface, and (ii) implement the operations of the interface in terms of the bottom operations. The design of an interface is in a sense the design of a new, more problem-oriented, programming language.

(b) Stepwise refinement (Wirth [2]): Factor the original algorithm into a number of sub-algorithms. The program is then composed from the implementations of the sub-algorithms. This is clearly a top-down method.

It is tempting to view stepwise refinement as a special case of (a), since one might consider the sub-algorithms as operations of a new language. For example, if "interpret command" is refined to "IF command ok THEN execute command ELSE error message FI", this may be considered as the design of a new language with operations "command ok", "execute command" and "error message". This is, however, hardly helpful. The essential difference between the operations of a layer and such sub-algorithms, is that the operations of a programming language must have their semantics as simple as possible and be of general applicability, whereas the semantics of the sub-algorithms is in general quite complicated, and the applicability is limited to one (or once in a while perhaps two or three) occurrences, and only in conjunction with the other sub-algorithms.

It is better to consider (a) and (b) as different principles which support and supplement each other. Both create hierarchies, but of different natures: in one case an essentially linear hierarchy of layers (or programming languages), in the other case a (hierarchical) tree structure with the resulting program text as terminal nodes, where the intermediate nodes are labelled with the intermediate sub-algorithms. In either case the ordering in the hierarchy corresponds to the level of abstraction, but these hierarchies should not be confused with each other.

To illustrate the process of stepwise refinement I will spend some time on the problem of finding sentences of a context-free language that are palindromes. (The choice of problem is rather immaterial for the purpose of this paper. This problem has been selected since it has no obvious solution, for it is undecidable whether a given context-free language contains palindromes or not. For some languages, of course, the question is settled easy enough, but no general method to solve the matter is conceivable.) At this moment, I have only a vague notion how to attack such a problem, but an approach is suggested by the following consideration. The given problem is a special instance of the general problem to find common elements of two sets (which, in this case, are a context-free language and the set of palindromes). A procedure to solve this problem is to generate members from one set successively, and to test each element for membership of the other set.

The fact that the test for palindromicity is quite simple, whereas even the order n^3 test for membership of a general context-free language is rather complicated (generating the elements of a context-free language is in comparison almost trivial), and the fact that there are probably many more palindromes of a given length than sentences of the language of that length, both point in one direction: to generate sentences and to test for palindromicity. It is silly, of course, to generate a sentence completely if the first production step, say, already displays different terminal symbols at the front and at the end. This suggests a process where the generation process is guided by whatever terminal symbols have already appeared or may still appear at the front and at the end. The process should maintain a list of possible sentential forms (unfinished potential palindromes) and each time select productions conforming to the present findings, more or less like predictive parsing. Rather than taking the reader by the hand and developing the program "together", I immediately give some developmental steps as they occur to me now.

After the very first steps, the algorithm reads as follows:

```
read grammar;
determine possible first and last terminal symbols for productions;
initialize sentential form list;
WHILE neither list nor patience exhausted
DO develop promising form OD.
```

An obvious candidate for refinement is "develop promising form", and a possible text is:

```
select promising {short} form;
determine its kernel {i.e., what is left of the form if matching
    terminal symbols are deleted};
IF kernel is empty THEN solution found
ELSE develop form
FI;
remove form from list.
```

Next, "develop form" may be refined by replacing it by:

```
determine intersection between possible first and last terminal
    symbols of kernel;
FOR each terminal symbol in intersection
DO select corresponding productions;
   add developed forms to list
OD.
```

The following text has now been constructed:

```
read grammar;
determine possible first and last terminal symbols for productions;
initialize sentential form list;
WHILE neither list nor patience exhausted
DO select promising {short} form;
   determine its kernel {i.e., what is left of the form if matching
       terminal symbols are deleted};
   IF kernel is empty THEN solution found
   ELSE determine intersection between possible first and last
            terminal symbols of kernel;
        FOR each terminal symbol in intersection
        DO select corresponding productions;
           add developed forms to list
        OD
   FI;
   remove form from list
OD.
```

Although this is still a very simple piece of program, the structure is already starting to become less obvious, since some of the abstractions involved are no longer explicitly stated. When eventually all sub-algorithms have been worked out and have replaced their indicators in the above text, the tree-like way in which it was constructed will have been obscured completely. The conceptual units of the program will no longer be visible.

(It is too early for commitment as to an intermediate layer, but some outlines are becoming visible. A priority queue is needed, i.e., a list type to which elements may easily be added, and whose shortest element can easily be extracted. Alternatively, a simple queue could be used, in which case "promising" means: derived in few production steps).

2. MODIFIABILITY

As in any model, the ideal situation depicted above is a simplification. The less ideal but more common case is that the programmer is not so lucky that he can plan his future steps in a surprise-free way. Even if he is quite experienced and very careful, many decisions will turn out to have such undesirable consequences that it is preferable to go back and change the decision (and the careful application of the above structuring principles tends to minimize the scope of the effects). If, however, the program is properly modified - a matter of discipline - it will eventually have the structure it would have had if the programmer had made the correct decisions immediately!

It is not only during program development that modifications are necessary. The situation that a large program has to be modified after it has been written is also extremely common. For example, a design requirement for all software products is that they behave "reasonably" even when the user exceeds the limits of the specifications. What "reasonable" means is a matter of user expectations and of human factors and cannot be defined formally. Anyone who has been exposed to software products will be able to supply a list of unreasonable properties that, though not bugs, are still intolerable. One cannot blame (at least, in many cases) the designers for not having foreseen the unfortunate consequences of the combined working of a number of decisions each of which is reasonable enough on its own. One can only require that the product be modified.

It should be clear that the quality of a program depends strongly on its modifiability, if only because it is the result of a long sequence of modifications. If these modifications are done poorly, the quality of the program will be no better, however well designed the original program may have been.

If a program has to be modified, knowledge of the program structure is extremely important. The structure makes it possible to see if a proposed modification would have farther-reaching consequences than was intended. Alas, reconstruction from the program text of the way the program was constructed is in general an impossible task.

The reason for introducing the notion of program structure was to cater for the situation where the program is too complex to see and understand all elements and their interrelationships simultaneously. For the validity of the argument that knowledge of its structure is important, it is therefore completely irrelevant whether the modifier coincides with the original author.

3. STRUCTURING PROGRAM TEXT

I hope the reader is convinced that the program structure must be described explicitly. But this is not enough. Not only must the program structure be made explicit, but this explicit description must also be modified in parallel with the program. It is better not to have any documentation than to have the documentation of a former version. Without documentation it is at least clear that to modify the program reliably one should discard the old text and start from scratch.

The most common reason for obsolescence of documentation is probably that it is just too easy to modify a program without corresponding modification of the documentation, or, to put it differently, that it is not easy enough to modify the documentation at the same time. It is not very realistic to expect that the quality of software will dramatically improve as long as this situation persists. The conclusion is that the structure of the program must be documented in the program text. In that way the documentation can be updated as a routine part of the modification of the program. Also, a description of the structure is always at hand when the program text is looked at. (Remember that the program structure was not an intrinsic part of the program text.)

For example, the "normal" way to implement a layer is to replace the abstract operations textually by the text of their implementations. In that case the structure is lost. What is needed is an explicit description of the interface that can be understood by the receiver of the high-level program text. The attention among language designers is now focussed to such an extent upon this issue (under the name of "abstract data types") that I shall spend no more words on it.

The same problem occurs with stepwise refinements. The normal way to perform a refinement step is to substitute the refinement body _in situ_. The final text is no longer reminiscent of the derivation process.

The solution should be obvious. Simply never perform a refinement step by literal in-situ substitution, but always _retain_ the refinement as a functional part of the program. Of course! Why should such a clerical task as textual substitution be performed by the programmer, especially when it is harmful to him? The start of the palindrome program from Section 1 could then run exactly as it was developed there. The program text becomes a linearization of the program tree, with repetition of the node labels ("develop promising form", "develop form") serving as a pointer. Since now no low level text exists, it becomes even difficult to make a patch.

This requires of course new programming languages (or very simple extensions of existing ones.)

Several questions arise. The first question is whether one could not use procedures for refinements. The general answer is that procedures are a completely different language feature: procedures govern the dynamic flow of control, whereas refinements are intended to control the static program structure. Refinements are possible (and sensible) for languages that do not have procedures at all (e.g., specification languages) or for program parts not involving the flow of control (e.g., declarations). Moreover, for languages that do have procedures we may ask the question: if procedures would do, why, then, are they not used for the purpose? A probable reason is that a procedure call, even without parameters, is rather expensive. This is of course an implementation-dependent property and it is imaginable that an implementation would make the obvious optimization for a parameterless procedure that is called only once. But this makes the implementation more complicated, and we should not encourage programming language features which require an optimization for special cases. It appears that procedures are too general a mechanism. If we look at the way procedures are used in practice, a rough division in three classes can be made: procedures for refinement purposes, which are called from one position without parameters and are not recursive (seldomly used and then only at a high level, near the top), procedures as building blocks for a layer, usually called from many places with parameters but also not recursive (except for the case where they traverse a recursive data structure), and finally procedures to "divide and conquer" a problem that can be expressed recursively, such as quicksort, backtracking or formula manipulation. It is unlikely that one feature is the best way of serving such different uses. Anyway, procedures may have parameters and be called recursively, and both possibilities should be excluded for refinements. Another problem with procedures is that one certainly does not want procedures to inherit access rights from their environment automatically, whereas a refinement should inherit everything. For at the time the refinement is written, we do not even yet know what there is to be inherited. (This is in complete contrast with the operations of a layer, which should inherit nothing whatsoever across the interface, that is, statically.) Finally, many languages require definition of procedures before application, an acceptable restriction, except for refinements.

The next question (having settled that refinements are a language feature on their own) is the syntactic form of their "handle". I have chosen the good old identifier. One reason is that it is an established way of naming programmer-defined entities. Another reason is that a good choice of identifiers is a better support to documentation than any other form I can think of. Identifiers almost compel a natural language choice, which, for not extremely formally inclined people, is a natural way to express the meaning of a sub-program on the proper level of abstraction. Even if a handle such as "$I := \{x \mid \exists u \in V_T\ [k \overset{*}{\Rightarrow} xu]\} \cap \{x \mid \exists v \in V_T\ [k \overset{*}{\Rightarrow} vx]\}$" were allowed, the effect at best would be that the reader would blink a few times and then exclaim: Oh! he means "determine intersection between possible first and last terminal symbols of k". Such formal expressions have a value at the places where they belong - in the assertions of a formal correctness proof. Another, admittedly weak, argument in favour of identifiers is the possibility to write a very simple preprocessor to handle refinements as an extension to an existing programming language.

Programmers are usually well-advised not to use very long identifiers. Handles of refinements, however, occur only twice in the text and the obvious argument against long identifiers does not apply. It makes sense to make them long enough to convey some information as to the meaning of the algorithm they stand for.

A research problem is what syntactic positions may be taken by refinement handles. Obvious cases are statements and expressions. Formal parameters, on the other hand, would be a bad idea. As can be seen from the example in Section 4, it may be desirable to allow refinements for declarations also. A definitive answer can only be developed with respect to specific programming languages, however.

Various programming _systems_ have been developed that aid in stepwise refinement by taking over the clerical tasks of textual replacement (see e.g. [3, 4]). These systems may also have the possibility to run an incomplete program. I do not think this offers a solution. In the first place, the proper and obvious tool for aiding the programming process is the programming language. Programs are what we use to communicate algorithms to ourselves, to other programmers and even to computers. The main reason, however, is rather more down to earth: even with the best conventions possible for indicating modifications, the pencil-and-paper method will remain an order of magnitude simpler. As long as the program is in its development phase, each impediment to easy modification is an impediment to obtaining a reliable program. The value of _describing_ an algorithm incompletely (down to a certain level of abstraction) is obvious, but the value of _running_ an incomplete program escapes me.

4. AN EXAMPLE

The use of abstract data types to simplify program modifications is discussed by Linden [5]. He rewrites Hoare´s sieve-of-Eratosthenes program [6] using abstract data types. But from his program it is apparent that the main gain in readability stems from the choice of identifiers for the operations on his abstract type "sieve set". His program would have been better structured if refinements had been used. The implementation of the operations is rather repetitious and it is as difficult to see that they are correct as in Hoare´s version. A change of representation would involve the major part of the program. Worse yet, there is a rather obscure correspondence between the high-level integer "next" in the program, and the low-level value "xindex". Although Linden states: "this program is a direct translation of the English definition of the algorithm", the coarse structure of the English definition he gives and that of his program are different! The structure of the core of the English definition is:

```
core:
    FOR each integer FROM two through the
        square root of n
    DO remove all multiples from the set OD,
```

whereas that of his program is (slightly simplified):

```
core:
    INT next:= 2;
    WHILE next not exceeding the square root of n
    DO remove multiples of next from the set;
       find next member
    OD.
find next member:
    REPEAT next:= next + 1
    UNTIL next in set.
```

The correctness proof of the second version requires, unlike the first version, an existence proof, given a prime p, of a second prime p′ with $p < p' \leq p^2$. This proof is not completely trivial. The optimization involved is justified by the statement: "it is not necessary to remove the multiples of any number which has already been removed from the set". This corresponds, however, to a refinement

```
    remove multiples if necessary:
        IF multiples already removed
        THEN SKIP
        ELSE remove multiples
        FI.
```

Below I give a version of the algorithm, in accordance with the ideas of this paper. The problem is that the table of primes has to be implemented using a bit map. Primitives for setting and testing one of the bits 0 through wordsize-1 in a "WORD" are assumed available, as is the upper bound n of the primes to be computed. The straightforward solution would involve a division per table access, which is considered unacceptable.

Some notations are employed below that are an immature attempt to express some ideas for the emancipation of abstract data types, taken from EL1 [7], namely the fact that for a new mode the primitive ways of access have to be described explicitly. Operators may also be anonymous (type conversion functions) or 3-adic (used to define a "generator" in the sense of Alphard [8]).

Two new layers are used, a very simple one for the integral square root, and a larger one for indexing. The definitions are not definitions of the class type, but of the module type (Schuman [9]) where details of the implementation are made invisible outside the module.

```
    sieve of eratosthenes USING integral square root, indexing.

    sieve of eratosthenes:
        declare sieve of size n;
        FOR each integer FROM two through the square root of n
        DO remove multiples if necessary OD;
        print table.
    remove multiples if necessary:
        IF multiples already removed
        THEN SKIP
        ELSE remove multiples
        FI.

    declare sieve of size n:
        [n] BOOL prime;
        fill prime.
    fill prime:
        FOR INDEX p FROM 2 BY 1 TO n
        DO prime[p]:= TRUE OD.
    print table:
        FOR INDEX p FROM 2 BY 1 TO n
        DO IF prime[p] THEN print (p) FI OD.
```

```
each integer:
    INDEX p.
two through the square root of n:
    2 BY 1 TO intsqrt (n).
multiples already removed:
    ¬prime[p].
remove multiples:
    FOR INDEX mult FROM p * p BY p TO n
    DO remove multiple OD.
remove multiple:
    prime[mult]:= FALSE.

DEF integral square root:
BEGIN PROC (INT a) INT intsqrt:
      IF a < 0 THEN ERROR
      ELIF a = 0 THEN 0
      ELSE INT rt:= a ÷ 2 + 1;
           {a < (rt + 1)^2}
           WHILE rt > a ÷ rt
           {a < rt^2}
           DO rt:= (rt + a ÷ rt) ÷ 2 OD;
           {rt^2 ≤ a < (rt + 1)^2}
           rt
      FI
END.

DEF indexing:
BEGIN PRIMITIVE MODE INDEX:
          (repr: STRUCT (INT intval, word, bit)
              {invariant: intval = word * wordsize + bit},
           assign (v): repr:= v,
           val: repr);

      {type conversion functions}
      OP (INT a) INDEX:
          (intval: a,
           word: a ÷ wordsize,
           bit: a MOD wordsize);
      OP (INDEX a) INT:
          a.intval;

      OP (INDEX a, b) INDEX +:
          IF a.bit + b.bit < wordsize
          THEN (intval: a.intval + b.intval,
                word: a.word + b.word,
                bit: a.bit + b.bit)
          ELSE (intval: a.intval + b.intval,
                word: a.word + b.word + 1,
                bit: a.bit + b.bit - wordsize)
          FI;
```

```
        OP (INDEX a, b, c) GENERATOR(INDEX) BY TO:
            (init: a,
             cont (i): i.intval ≤ c.intval,
             next (i): i + b);

        PRIMITIVE MODE [INDEX] BOOL:
            (repr (size): [0 : size.word] WORD,
             assign (i, v): setbit (i.bit, repr[i.word], ABS v),
             val (i): getbit (i.bit, repr[i.word]) = 0)
    END.
```

5. CONCLUSION

Let me first point out that I do not think that structured program texts are a panacea for the software crisis. If the structure of a program is abominable, it is not particularly helpful to faithfully mirror its structure in the program text (except that a recipient of the program is in a better position not to place unjustified confidence in the program). Similarly, structuring program texts cannot replace the task of proving correctness (but it may be of help). The main advantage of making the program structure explicit as advocated here is that this provides documentation as an integral part of the program text. All necessary modifications can easily be made in the proper way, and updating this documentation in parallel with the program becomes almost automatic (making patches may become hard indeed). In short, future languages should support explicit structuring of program texts and thereby increase the quality of programs.

Of the two major principles introducing program structure, only one, layers of abstraction, is receiving due attention from language designers. The other one, stepwise refinement, seems more or less neglected. The purpose of this paper has been to show that the introduction of refinements as a language feature is desirable and feasible (in fact, SLAN [10] already has a refinement facility, and the standard control structure of CDL and its offsprings CDL2 [11] and ALEPH [12] is very similar). Since the refinements correspond closely to the way a program is developed, they should prove a natural instrument for the programmer.

ACKNOWLEDGEMENTS

The idea of structuring a program text according to the program structure could not have evolved without the work of Dijkstra, Naur, Wirth, Hoare, and too many others to mention. I am indebted to Kees Koster for several suggestions about the content and organization of this paper.

REFERENCES

[1] Dijkstra, E.W., The structure of the "THE" multiprogramming system, Comm. ACM 11 (1968) 341-346.

[2] Wirth, N., Program development by stepwise refinement, Comm. ACM 14 (1971) 221-227.

[3] Henderson, P. & R.A. Snowdon, A tool for structured programming development, Information Processing 74 (Proceedings of IFIP Congress 1974), 204-207, North-Holland, Amsterdam (1975).

[4] Cunningham, R.J. & C.G. Pugh, A language-independent system to aid the development of structured programs, Software - Practice & Experience 6 (1976) 487-503.

[5] Linden, T.A., The use of abstract data types to simplify program modifications, Proceedings of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11, special issue, 12-23 (1976).

[6] Hoare, C.A.R., Notes on data structuring, in O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Structured Programming, Academic Press (1972).

[7] Wegbreit, B., The treatment of data types in EL1, Comm. ACM 17 (1974) 251-264.

[8] Shaw, M., W.A. Wulf & R.L. London, Abstraction and verification in Alphard: Iteration and generators, Technical Report, Carnegie-Mellon University and University of Southern California Information Sciences Institute (1976).

[9] Schuman, S.A., Toward modular programming in high-level languages, ALGOL Bulletin 37.4.1 (1974) 30-53.

[10] Hommel, G., S. Jähnichen & W. Koch, SLAN - Eine erweiterbare Sprache zur Unterstützung der strukturierten und modularen Programmierung, in Programmiersprachen Fachtagung 1976, Informatik Fachberichte 1, Springer-Verlag, Heidelberg (1976).

[11] Dehottay, J.-P., H. Feuerhahn, C.H.A. Koster & H.M. Stahl, Syntaktische Beschreibung von CDL2, Forschungsgruppe Softwaretechnik, Technische Universität Berlin (1976).

[12] Bosch, R., D. Grune & L.G.L.T. Meertens, ALEPH, A Language Encouraging Program Hierarchy, The International Computing Symposium 1973, Davos, 73-79, North-Holland, Amsterdam (1974).