Programming Methodology

A Collection of Articles by Members of IFIP WG2.3

Edited by

David Gries

Programming Methodology

A Collection of Articles by Members of IFIP WG2.3

Edited by

David Gries

Springer-Verlag
New York Heidelberg Berlin

David Gries

Cornell University
Department of Computer Science
Upson Hall
Ithaca, New York 14859
USA

AMS Subject Classifications: 68A05, 68A10, 68A20
(C.R.) Computing Classifications: 4.0, 4.2, 4.35, 5.24

Library of Congress Cataloging in Publication Data

Main entry under title:

Programming methodology.

(Texts and monographs in computer science)
Bibliography: p.
Includes index.
1. Electronic digital computers—Programming—
Addresses, essays, lectures. I. Gries, David,
1939–
QA76.6.P7516 001.6'42 78-16539
ISBN 0-387-90329-1

Printed in the United States of America.

ISBN 0-387-90329-1 Springer-Verlag New York

ISBN 3-540-90329-1 Springer-Verlag Berlin Heidelberg

Preface

This volume is being published for two reasons. The first is to present a collection of previously published articles on the subject of programming methodology that have helped define the field and give it direction. It is hoped that the scientist in the field will find the volume useful as a reference, while the scientist in neighboring fields will find it useful in seriously acquainting himself with important ideas in programming methodology. The advanced student can also study it—either in a course or by himself—in order to learn significant material that may not appear in texts for some time.

The second reason for this volume is to make public the nature and work on programming methodology of IFIP Working Group 2.3, hereafter called WG2.3. (IFIP stands for *International Federation for Information Processing*.) WG2.3 is one of many IFIP Working Groups that have been established to provide international forums for discussion of ideas in various areas. Generally, these groups publish proceedings of some of their meetings and occasionally they sponsor a larger conference that persons outside a group can attend.

WG2.3 has been something of a maverick in this respect. From the beginning the group has shunned paperwork, reports, meetings, and the like. This has meant less publicity for IFIP and WG2.3, but on the other hand it has meant that meetings could be devoted almost wholly to scientific discussions.

Moreover, meetings have not centered on formal presentation of completed, published material; instead, the emphasis has been on the presentation and discussion of research underway. Thus, members could receive their colleagues' constructive criticisms at a much earlier stage than usual. Many members feel that this mode of operation has furthered their own research endeavors, and have accordingly acknowledged WG2.3 in their publications.

This volume, then, is the first formal "output" from WG2.3. It contains articles by members of the group that are deemed to be significant and

exemplary work of programming methodology and of WG2.3. Unfortunately, lack of space prohibits the incorporation of material by all members. Many have written important books, articles, and technical reports that simply could not be included.

Each of the next five Parts consists of an Introduction and a series of articles devoted to one aspect or area of programming methodology. Part I contains a number of largely nontechnical articles, many of them based on lectures, which give thoughts, opinions, and viewpoints on various aspects of the field. This Part should give the reader a definite view of where the experts think programming has been and where it is or should be going.

Parts II through V then cover four different areas of programming methodology in detail. These certainly do not define the complete field of programming methodology (which is nowhere defined), but they represent significant aspects of the subject. Part II contains articles on the use of *correctness proofs* in programming and the related topic of defining a programming language so as to facilitate proofs. Part II is the largest of the five Parts, reflecting the importance of the subject and the major role played by members of WG2.3 in its development. The articles in Part III attack the problem of *harnessing parallelism* so that it can be used effectively—particularly in operating systems. Part IV is devoted to the topic of (programmer-defined) *data types* and their use in programming. Finally, the articles in Part V deal with different aspects of creating large programs and/or systems of programs, and is entitled Software Development.

Following Part V is a list of references, which is split into two sections. The first is a WG2.3 Bibliography—a list of publications relevant to programming methodology by members of WG2.3. Most of this was compiled by Sol J. Greenspan and Jim J. Horning (see [Horning 77b*]); the reader might wish to obtain this report, which contains annotations not included here.

All the publications cited by articles in this volume are listed either in the WG2.3 Bibliography or in the second list of references following this bibliography, and all references within the text are to one of these lists. Examples will illustrate the nature of the text references: [Burstall 72b*] refers to the second (because of b) 1972 article by Burstall (Algebraic description of programs with assertions, verification, and simulation); the "*" indicates that it is to be found in the WG2.3 Bibliography. The reference [Mills 72] refers to the 1972 article by Mills (Mathematical foundations of structured programming), which appears in the second list of references (no "*" is present).

In compiling this volume, I have had the help of many people. Mike Woodger, the first chairman of WG2.3, was influential in getting this project underway and constructed an initial list of potential articles. Jim Horning, the current chairman, continued to support the project and (along with Greenspan) provided most of the references in the WG2.3 Bibliography. I have had the advice and criticism of WG2.3 members and

of Jim Donahue, Greg Andrews, and Manfred Paul both on my selection of articles and on my Introductions. Needless to say, however, I take full responsibility for any mistakes, for the selection and arrangement of the articles, and for the omission of many other excellent articles in order to keep the volume to a reasonable size. It should be mentioned that the volume would not have been possible without the dedicated and creative work of the authors of the articles.

David Gries

Acknowledgments

The editor, the authors, IFIP, and the publishers acknowledge with thanks permission to reprint the copyrighted articles in this book that have been published in various journals, proceedings, and books. With a few exceptions, the articles are reprinted from *Acta Informatica*, the *Communications of the ACM*, the *Computer Bulletin*, *IEEE Trans. on Software Engineering*, *Lecture Notes in Computer Science* (Springer-Verlag), books by Academic Press and Prentice–Hall, and the *Proc. of the International Conference on Reliable Software*. Below we give the individual credits.

Brinch Hansen, P. Structured multiprogramming. By permission of the ACM, from *CACM* **15** (July 1972), 574–578.

Brinch Hansen, P. The programming language Concurrent Pascal. By permission of the Institute of Electrical and Electronics Engineers, Inc., from *IEEE Trans. Software Eng.* **1** (June 1975), 199–207.

Buxton, J. N. Software engineering. By permission of the author, from *Proc. 1974 CERN School of Computing*, CERN Rpt. No. 74-23 (Nov 1974), 394–401.

Dahl, O.-J. An approach to correctness proofs of semicoroutines. By permission of the author, from *Math. Foundations of Computer Science. LNCS* 28 (1975), 157–174.

Darlington, J. and Burstall, R. M. A system which automatically improves programs. By permission of Springer-Verlag, from *Acta Informatica* **6** (1976), 41–60.

Dijkstra, E. W. The humble programmer. By permission of the ACM, from *CACM* **15** (Oct 1972), 859–886.

Dijkstra, E. W. Correctness concerns and, among other things, why they are resented. By permission of the author, from *Proc. Int. Conf. on Reliable Software*, *SIGPLAN Notices* **10** (June 1975), 546–550.

Dijkstra, E. W. Guarded commands, nondeterminacy, and formal derivation of programs. By permission of the ACM, from *CACM* **18** (Aug 1975), 453–457; also appeared in Yeh, R. T. (ed.), *Current Trends in Programming Methodology I*. Prentice–Hall, 1976, 233–242.

Gries, D. On structured programming. By permission of the ACM, based on a letter to the editor by the author in *CACM* **17** (Nov 1974), 655–657.

Guttag, J. V. and Horning, J. J. The algebraic specification of abstract data types. By permission of Springer-Verlag, from *Acta Informatica*, to appear.

Hoare, C. A. R. An axiomatic basis for computer programming. By permission of the ACM, from *CACM* **12** (Oct 1969), 576–580, 583.

Hoare, C. A. R Proof of a program: FIND. By permission of the ACM, from *CACM* **14** (Jan 1971), 39–45.

Hoare, C. A. R. Towards a theory of parallel programming. By permission of Academic Press, from Hoare, C. A. R. and Perrott, R. N. (eds.), *Operating Systems Techniques*. Academic Press, 1972.

Hoare, C. A. R. Proof of correctness of data representations. By permission of Springer-Verlag, from *Acta Informatica* **1** (1972), 271–281.

Hoare, C. A. R. Monitors: an operating system structuring concept. By permission of the ACM, from *CACM* **17** (Oct 1974), 549–557.

Hoare, C. A. R. The engineering of software: a startling contradiction. By permission of the British Computer Society, based on a note by the author in the *Computer Bulletin* (Dec 1975).

Lehman, M. M. Programs, cities, students—limits to growth? By permission of the author, from *Inaugural Lecture College Series*, Vol. 9, 1970–1974, 211–229.

Owicki, S. and Gries, D. An axiomatic proof technique for parallel programs. By permission of Springer-Verlag, from *Acta Informatica* **6** (1976), 319–340.

Parnas, D. L. On a "buzzword": hierarchical structure. By permission of the author, from *IFIP 1974*, 336–339.

Parnas, D. L. On the design and development of program families. By permission of the Inst. of Electrical and Electronics Engineers, Inc., from *IEEE Trans. Software Eng.* **1** (March 1976), 1–9.

Randell, B. System structure for software fault tolerance. By permission of the Inst. of Electrical and Electronics Engineers, Inc., from *IEEE Trans. Software Eng.* **1** (April 1975), 220–232; also appeared in Yeh, R. T. (ed.), *Current Trends in Programming Methodology I*. Prentice–Hall, 1976, 195–219.

Reynolds, J. C. Programming with transition diagrams. By permission of the author.

Reynolds, J. C. User-defined types and procedural data structures as complementary approaches to data abstraction. By permission of the author, from Schuman, S. A. (ed.), *New Directions in Algorithmic Languages 1975*, Inst. de Recherche d'Informatique et d'Automatique, Rocquencourt, 1975, 157–168.

Ross, D. T. Structured analysis (SA): a language for communicating ideas. By permission of the Inst. of Electrical and Electronics Engineers, Inc., from *IEEE Trans. Software Eng.* **3** (Jan 1977), 16–34.

Turski, W. M. Software engineering—some principles and problems. By permission of the author, from *Mathematical Structures–Computational Mathematics–Mathematical Modelling*. Publ. House of the Bulgarian Academy of Sciences, Sofia, 1975, 485–491.

Wirth, N. Program development by stepwise refinement. By permission of the ACM, from *CACM* **14** (April 1971), 221–227.

Contents

One method of achieving the desired effect would be to sort the whole array. If the array is small, this would be a good method; but if the array is large, the time taken to sort it will also be large. The Find program is designed to take advantage of the weaker requirements to save much of the time which would be involved in a full sort.

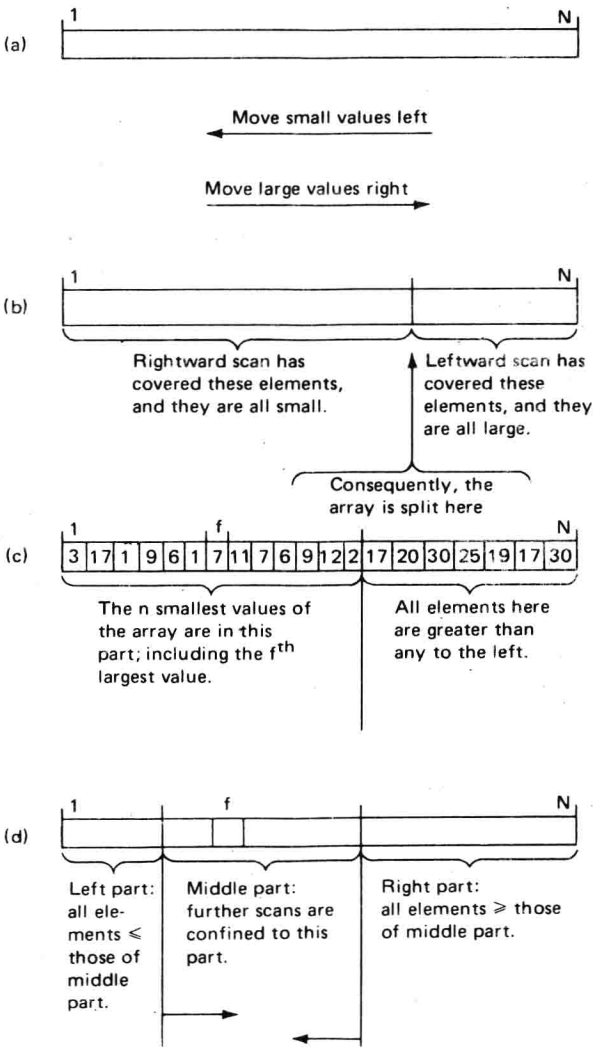
The usefulness of the Find program arises from its application to the problem of finding the median or other quantiles of a set of observations stored in a computer array. For example, if N is odd and f is set to $(N+1)/2$, the effect of the Find program will be to place an observation with value equal to the median in $A[f]$. Similarly the first quartile may be found by setting f to $(N+1)/4$, and so on.

The method used is based on the principle that the desired effect of Find is to move lower valued elements of the array to one end—the "left-hand" end—and higher valued elements of the array to the other end —the "right-hand" end. (See Table I(a)). This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered which is small will remain where it is, but any element which is large should be moved up to the right-hand end of the array, in exchange for a small one. In order to find such a small element, a separate scan is made, starting at the right-hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered is moved down to the left-hand end in exchange for the large element already encountered in the rightward scan. Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be large. When this condition holds, we will say that the array is *split at* the given point into two parts (see Table I(b)).

The reasoning of the previous paragraph assumes that there is some means of distinguishing small elements from large ones. Since we are interested only in their comparative values, it is sufficient to select the value of some arbitrary element before either of the scans starts; any element with lower value than the selected element is counted as small, and any element with higher value is counted as large. The fact that the discriminating value is arbitrary means that the place where the two scans will meet is also arbitrary; but it does not affect the fact that the array will be split at the meeting point, wherever that may be.

Now consider the question on which side of the split the fth element in order of value is to be found. If the split is to the right of $A[f]$, then the desired element must of necessity be to the left of the split, and all elements to the right of the split will be greater than it. In this case, all elements to the right of the split can be ignored in any future processing, since they are already in their proper place, namely to the right of $A[f]$ (see Table I(c)). Similarly, if the split is to the left of $A[f]$, the element to be found must be to the right of the split, and all elements to the left of the

Table 1


(a)
1
N
Move small values left
Move large values right
(b)
1
N
Rightward scan has covered these elements, and they are all small.
Leftward scan has covered these elements, and they are all large.
Consequently, the array is split here
(c)
1
f
N
3 17 1 9 6 1 7 11 7 6 9 12 2 17 20 30 25 19 17 30
The n smallest values of the array are in this part; including the f^th largest value.
All elements here are greater than any to the left.
(d)
1
f
N
Left part: all elements ≤ those of middle part.
Middle part: further scans are confined to this part.
Right part: all elements ≥ those of middle part.

split must be equal or less than it; furthermore, these elements can be ignored in future processing.

In either case, the program proceeds by repeating the rightward and leftward scans, but this time one of the scans will start at the split rather than at the beginning of the array. When the two scans meet again, it will be known that there is a second split in the array, this time perhaps on the other side of $A[f]$. Thus again, we may proceed with the rightward and leftward scans, but we start the rightward scan at the split on the left of $A[f]$ and the leftward scan at the split on the right, thus confining attention only to that part of the array that lies between the two splits; this will be known as the *middle part* of the array (see Table I(d)).

When the third scan is complete, the middle part of the array will be split again into two parts. We take the new middle part as that part which contains $A[f]$ and repeat the double scan on this new middle part. The process is repeated until the middle part consists of only one element, namely $A[f]$. This element will now be equal to or greater than all elements to the left and equal to or less than all elements to the right; and thus the desired result of Find will be accomplished.

This has been an informal description of the method used by the program Find. Diagrams have been used to convey an understanding of how and why the method works, and they serve as an intuitive proof of its correctness. However, the method is described only in general terms, leaving many details undecided; and accordingly, the intuitive proof is far from watertight. In the next section, the details of the method will be filled in during the process of coding it in a formal programming language; and simultaneously, the details of the proof will be formalized in traditional logical notation. The end product of this activity will be a program suitable for computer execution, together with a proof of its correctness. The reader who checks the validity of the proof will thereby convince himself that the program requires no testing.

3 Coding and proof construction

The coding and proof construction may be split into several stages, each stage dealing with greater detail than the previous one. Furthermore, each stage may be systematically analyzed as a series of steps.

3.1 Stage 1: problem definition

The first stage in coding and proof construction is to obtain a rigorous formulation of what is to be accomplished, and what may be assumed to begin with. In this case we may assume:

(a) The subscript bounds of A are 1 and N.
(b) $1 \leqslant f \leqslant N$.

The required result is:

$$\forall p,q(1 \leqslant p \leqslant f \leqslant q \leqslant N \supset A[p] \leqslant A[f] \leqslant A[q]) \qquad [\text{Found}]$$

3.2 Stage 2: the general method

(1) The first step in each stage is to decide what variables will be required to hold intermediate results of the program. In the case of Find, it will be necessary to know at all times the extent of the middle part, which is currently being scanned. This indicates the introduction of variables m and n to point to the first element $A[m]$ and the last element $A[n]$ of the middle part.

(2) The second step is to attempt to describe more formally the purpose of each variable, which was informally described in the previous step. This purpose may be expressed as a formula of logic which is intended to remain true throughout the execution of the program, even when the value of the variable concerned is changed by assignment.[1] Such a formula is known as an *invariant*. As mentioned above, m is intended to point to the leftmost element of the middle part of the array; and the middle part at all times contains $A[f]$; consequently m is never greater than f. Furthermore, there is always a split just to the left of the middle part, that is between $m-1$ and m. Thus the following formula should be true for m throughout execution of the program:

$$m \leqslant f \;\&\; \forall p,q(1 \leqslant p < m \leqslant q \leqslant N \supset A[p] \leqslant A[q])$$

[m-invariant]

Similarly, n is intended to point to the rightmost element of the middle part; it must never be less than f, and there will always be a split just to the right of it:

$$f \leqslant n \;\&\; \forall p,q(1 \leqslant p \leqslant n < q \leqslant N \supset A[p] \leqslant A[q])$$

[n-invariant]

(3) The next step is to determine the initial values for these variables. Since the middle part of the array is intended to be the part that still requires processing, and since to begin with the whole array requires processing, the obvious choice of initial values of m and n are 1 and N, respectively, indicating the first and last elements of the whole array. The code required is:

$m := 1; \quad n := N$

[1] Except possibly in certain "critical regions."

(4) It is necessary next to check that these values satisfy the relevant invariants. This may be done by substituting the initial value for the corresponding variable in each invariant, and ensuring that the result follows from facts already known:

$$1 \leqslant f \leqslant N \supset 1 \leqslant f \;\&\; \forall p,q(1 \leqslant p < 1 \leqslant q \leqslant N \supset A[p] \leqslant A[q])$$

[Lemma 1]

$$1 \leqslant f \leqslant N \supset f \leqslant N \;\&\; \forall p,q(1 \leqslant p \leqslant N < q \leqslant N \supset A[p] \leqslant A[q])$$

[Lemma 2]

The quantified clause of each lemma is trivially true since the antecedents of the implications are always false.

(5) After setting the initial values, the method of the program is repeatedly to reduce the size of the middle part, until it contains only one element. This may be accomplished by an iteration of the form:

while $m<n$ **do** "reduce middle part"

(6) It remains to prove that this loop accomplishes the objectives of the program as a whole. If we write the body of the iteration properly (i.e. in such a way as to preserve the truth of all invariants) then all invariants will still be true on termination. Furthermore, termination will occur only when $m<n$ goes false. Thus it is necessary only to show that the combination of the truth of the invariants and the falsity of the while clause expression $m<n$ implies the truth of Found.

$$m \leqslant f \;\&\; \forall p,q(1 \leqslant p < m \leqslant q \leqslant N \supset A[p] \leqslant A[q])$$

$$\&f \leqslant n \;\&\; \forall p,q(1 \leqslant p \leqslant n < q \leqslant N \supset A[p] \leqslant A[q]) \;\&\; \neg m < n$$

$$\supset \forall p,q(1 \leqslant p \leqslant f \leqslant q \leqslant N \supset A[p] \leqslant A[f] \leqslant A[q]) \quad \text{[Lemma 3]}$$

The antecedents imply that $m=n=f$. If $1 \leqslant p \leqslant f \leqslant q \leqslant N$, then either $p=f$, in which case $A[p] \leqslant A[f]$ is obvious, or $p<f$, in which case substituting f for both m and g in the first quantified antecedent gives $A[p] \leqslant A[f]$. A similar argument shows that $A[f] \leqslant A[q]$.

At this point, the general structure of the program is as follows:

$m := 1$; $n := N$;

while $m<n$ **do** "reduce middle part"

Furthermore, this code has been proved to be correct, provided that the body of the contained iteration is correct.

3.3 Stage 3: reduce the middle part

(1) The process for reducing the middle part involves a scan from the left and from the right. This requires two pointers, i and j, pointing to elements $A[i]$ and $A[j]$ respectively. In addition, a variable r is required to hold the

arbitrary value which has been selected to act as a discriminator between "small" and "large" values.

(2) The i pointer is intended to pass over only those array elements with values smaller than r. Thus all array elements strictly to the left of the currently scanned element $A[i]$ will be known always to be equal to or less than r:

$$m \leqslant i \;\&\; \forall p(1 \leqslant p < i \supset A[p] \leqslant r) \qquad [i\text{-invariant}]$$

Similarly the j pointer passes over only large values, and all elements strictly to the right of the currently scanned element $A[j]$ are known always to be equal to or greater than r:

$$j \leqslant n \;\&\; \forall q(j < q \leqslant N \supset r \leqslant A[q]) \qquad [j\text{-invariant}]$$

Since the value of r does not change, there is no need for an r-invariant.

(3) The i pointer starts at the left of the middle part, i.e. at m; and the j pointer starts at the right of the middle part, i.e. at n. The initial value of r is taken from an arbitrary element of the middle part of the array. Since $A[f]$ is always in the middle part, its value is as good as any.

(4) The fact that the initial values satisfy the i- and j-invariants follows directly from the truth of the corresponding m- and n-invariants; this is stated formally in the following lemmas:

$$f \leqslant N \;\&\; m \leqslant f \;\&\; \forall p,q(1 \leqslant p < m \leqslant q \leqslant N \supset A[p] \leqslant A[q])$$
$$\supset m \leqslant m \;\&\; \forall p(1 \leqslant p < m \supset A[p] \leqslant A[f]) \qquad [\text{Lemma 4}]$$

$$1 \leqslant f \;\&\; f \leqslant n \;\&\; \forall p,q(1 \leqslant p \leqslant n < q \leqslant N \supset A[p] \leqslant A[q])$$
$$\supset n \leqslant n \;\&\; \forall q(n < q \leqslant N \supset A[f] \leqslant A[q]) \qquad [\text{Lemma 5}]$$

The first of these is proved by setting q to f and the second by setting p to f.

(5) After setting the initial values, the method is to repeatedly add one to i and subtract one from j, until they cross over. This may be achieved by an iteration of the form:

while $i \leqslant j$ **do** "increase i and decrease j"

On exit from this loop, $j < i$ and all invariants are intended to be preserved.

If j and i cross over above f, the proposed method assigns j as the new value of n; if they cross over below f, i is assigned as the new value of m.

if $f \leqslant j$ **then** $n := j$

else if $i \leqslant f$ **then** $m := i$

else go to L