PROGRAMMING METHODOLOGY

Making a Science Out of an Art

by David Gries and Fred B. Schneider

It doesn't take too long for an intelligent, scientifically oriented person to learn to cobble programs together in FORTRAN, BASIC, or Pascal. Sure, there are mistakes, but everyone makes mistakes, so one simply spends the necessary time debugging. And one gets better at programming simply by doing lots of it. So what do they teach about programming? What is there to it?

This attitude is common and may even be reasonable for casual programming. For any serious programming, however, it invites disaster. A casual program bears little resemblance to the system of a thousand to a million lines of codes that a professional must be able to write (and read) in concert with as many as fifty other people. Such a program must be correct, as simple as possible, and capable of being readily understood, modified, and used by others.

How *does* one write programs that satisfy these requirements? That was the subject of a NATO conference held in Germany in 1968. At the conference the term *software crisis* was often heard, for there was indeed a crisis. The programming industry was being asked to develop larger and more complicated systems of programs, and they didn't have the expertise to do so effectively. The conference led to world-wide recognition that programming was indeed a difficult intellectual activity. The term *software engineering* was coined there to denote the collection of technical and managerial techniques used in the "software life cycle"—in the planning, analysis, design, implementation, testing, documentation, distribution, and maintenance of a programming system—and research in all these aspects began in earnest.

A SCIENCE CONCERNED WITH MENTAL TOOLS

At Cornell, prompted partly by our lack of understanding of how to *teach* programming, we became involved in the study of methods for developing and understanding programs, a field that has become known as *programming methodology*.

Programming methodology has been a central theme in the Cornell department for fifteen years and has influenced our work in other areas. For example, ideas about the process of program development influence thought on compiler construction, programming-language design, structured editors, debugging tools, "pretty printers" (which print a program in an indented format in accordance with the program structure), and computer verification of the correctness of a program or, indeed, of any mathematical proof. These related areas deal with *supplemental* tools used by the programmer; programming methodology in its narrowest sense is more concerned with the *mental* tools that are needed.

Research done so far has convinced us that programming can become a science, based on the knowledge and application of principles, rather than an art, which can be learned simply by watching and doing. We have discovered that programming at its best is a mathematical activity, requiring from the programmer all the taste, elegance, and desire for simplicity that characterizes mathematicians. Our exper-

ience has greatly influenced how we teach programming and how we present algorithms in higher-level courses. So far, most of the research has dealt with small programs, but larger ones are being considered.

In this article we describe some of the basic ideas involved in programming methodology. We use only one small example, but this should be enough to whet your appetite for more. Toward the end we present a couple of problems with the solutions we developed; we encourage you to try to solve them before looking at the solutions.

TAMING COMPLEXITY: THE FIRST NECESSITY

As any programmer will tell you, even a ten-line program can be complex and difficult to understand. Think, then, of the complexity of a ten-thousand-line program! Somehow, the programmer must master the complexity, must prevent it from rearing its ugly head.

The amount of work required to understand a program must be proportional to its length. This will only be the case if the program structure and the interactions between the program segments are kept *simple*. And the longer the program, the more important it is to keep things simple. Computer science already has a branch called *computational complexity*; in contrast, we like to call the field of programming methodology *computational simplicity*.

How do we achieve simplicity? The general method is to introduce suitable notations and use *abstraction*: various aspects of a problem are brought to the fore and others are hidden in the background to be dealt with later. New formalisms are developed, along with notations that allow the expression of concepts and the manipulation of formulas in various ways in order to prove things about them. In essence, mathematics is used, as in any scientific field, to master complexity.

In our research we have turned mostly to formal logic to help us determine what is meant by *correctness* of a program, for without knowing that, it is difficult to write correct programs. This has led to definitions of programming languages in terms of correctness rather than in terms of how a program is executed. And from these mathematical definitions, theories and principles for developing programs have arisen.

This does not mean that every program must be developed and proved correct in a formal manner. It does mean, however, that the programmer with a sound knowledge of the theory and principles behind program correctness and program development can use them in an informal manner, relying on the formalism when it is needed—when the problems become more complex.

WHAT DOES PROGRAM CORRECTNESS MEAN?

A program (or a segment of one) is correct if its execution, begun in any "reasonable" state, ends in a desired final state. That is: if its input variables have proper values, then so will its output variables.

We describe sets of reasonable or desired states by true-false statements, called *assertions*, about the program variables. To illustrate, let us suppose we want a program S to store in an array the cubes of the first n natural numbers, where integer value n is at least 0 and the array is denoted by $b[0..n-1]$. (By convention, if $n = 0$ the array is assumed to be empty.) For example, if we execute the program with $n = 4$, the resulting array will be $b[0] = 0$, $b[1] = 1$, $b[2] = 8$, $b[3] = 27$. Below, we specify S by giving a *precondition* P that describes the set of possible initial states and a *post-condition* R that describes the corresponding final states. In assertion R, the phrase $0 \leq i < n$ means we are interested only in integers at least 0 and less than n; for such integers i, $b[i] = i^3$.

P: $n \geq 0$
R: (**for all** i: $0 \leq i < n$: $b[i] = i^3$)

We say that S is correct with respect to P and R, written as $\{P\}\ S\ \{R\}$, if execution of S begun in a state in which P is true terminates in a state in which R is true. Nothing is said about execution of S begun in a state in which P is not true.

HOW CAN CORRECTNESS BE PROVED?

It is difficult to prove $\{P\}\ S\ \{R\}$ using only our operational understanding of how S is executed. Given some initial state, we can execute the program by hand (or let the computer do it) to determine what the final state is, but to prove correctness using this approach we would have to execute the program once for each possible initial state, and most of us don't have time for that! No, a way must be found that allows us to deduce correctness without relying on the notion of execution, and this calls for a mathematical theory of correctness. For each kind of statement, we need a definition that gives the pairs of pre- and post-conditions related by it.

The theory will tell us, for example, that the following are true about the assignments $x := 0$ and $x := x + 1$ to integer variable x:

$\{0 \bullet y = 0\}\ x := 0\ \{x \bullet y = 0\}$,
$\{x+1 > 0\}\ x := x+1\ \{x > 0\}$.

(Note that $0 \bullet y = 0$ is always true, so that precondition is equivalent to *true*. Similarly, the second precondition is equivalent to $x \geq 0$.)

The possible pre- and post-conditions for a statement should be related by a simple syntactic transformation, and not only by meaning, so that one really can manipulate statements the way one does arithmetic or logical statements. For example, the statement $x := e$, which assigns the value of expression e to variable x, is *defined* by the rule

$\{R^x_e\}\ x := e\ \{R\}$
(for all assertions R).

In this expression R^x_e is the assertion obtained by simultaneously replacing every occurrence of "x" in R by "e". Thus, given that R is to be true after execution of $x := e$, we can determine easily what has to be true before execution: R^x_e. We see that this holds for the two examples given above. For example, in

$\{P\colon 0 \bullet y = 0\}\ x := 0\ \{R\colon x \bullet y = 0\}$,

P is the result of substituting 0 for x in R.

It is rather neat that this simple notion of textual substitution, which is a basic concept of mathematical logic, can be used so simply to define the assignment statement.

Other statements are defined similarly. For example, sequencing of two statements $S0$ and $S1$ is defined:

If $\{P\}\ S0\ \{Q\}\ S1\ \{R\}$,
then $\{P\}\ S0;\ S1\ \{R\}$
(for any assertions P, Q, and R).

This definition allows us to compute the precondition for a sequence of assignments simply by beginning with the postcondition and iteratively working "backward", using the assignment statement definition. For example, it allows us to prove that the sequence $t := x;\ x := y;\ y := t$ exchanges the values of variables x and y. (Below, X and Y denote the final values of x and y, respectively.)

$\{y = X \text{ and } x = Y\}$
$t := x;$
$\{y = X \text{ and } t = Y\}$
$x := y;$
$\{x = X \text{ and } t = Y\}$
$y := t;$
$\{x = X \text{ and } y = Y\}$

Problem 1
COMPUTING CUBES

Write a program to store the cubes of the first n natural numbers in array $b[0..n-1]$. Use only addition operations. (See page 26 for a solution.)

Problem 2
THE MAXIMUM-SUM SEGMENT

Suppose we are given integer array $b[0..n-1]$ for $n \geq 0$. Let $S_{i,j}$ denote the sum of the values of segment $b[i..j-1]$. (If $i = j$, the segment is empty and the sum is 0.) Write a program to store in variable s the largest sum $S_{i,j}$ over all segments $b[i..j-1]$ of array $b[0..n-1]$. (See page 27 for a solution.)

WHAT ABOUT PROGRAM DEVELOPMENT?

Proving a program correct after it has been written is difficult. It makes more sense to develop a program and its correctness proof hand-in-hand, with the *proof* leading the way. When doing this, it is important to write the program specification as precisely as possible (in terms of pre- and post-conditions) because *the specification should drive program development.* To convey this idea through an example, we will consider again the problem of writing a program to store the cubes of the first n natural numbers in array $b[0..n-1]$, with the restriction that since exponentiation and multiplication are expensive, only additive operations should be used in the program. *Try writing a program for PROBLEM 1 yourself before looking at our development on the following page.*

In the event that you have a little trouble, we should point out that SOLUTION 1 was developed using various principles of programming methodology that have only been outlined here. Naturally, you might have difficulty applying them yourself at this point. However, any programmer well versed in the methodology would derive essentially the same program as the one in SOLUTION 1 in perhaps twenty minutes. *How did your solution compare?*

We give one more example without the program development. *Try to develop PROBLEM 2 yourself before reading our solution on a following page.*

The program for PROBLEM 2 has an interesting history. Jon Bentley at Carnegie-Mellon and Bell Laboratories

SOLUTION TO PROBLEM 1

The first step is to specify the program formally by writing pre-and post-conditions:

Precondition P: $0 \leq n$
Postcondition R: (**for all** i: $0 \leq i < n$: $b[i] = i^3$)

Assuming the use of a loop to calculate the elements of array b, our correctness ideas require writing an assertion that indicates what is true of b just before and after each iteration of the loop. To find this assertion, we introduce a fresh variable k (say), which in this case will be what is often called a "loop counter", put suitable bounds on it, and replace n in R by k, yielding the following two assertions $P0$ and $P1$:

$P0$: $0 \leq k \leq n$
$P1$: (**for all** i: $0 \leq i < k$: $b[i] = i^3$)

We can make $P0$ and $P1$ true by setting k to 0. Also, when $k = n$, R is true. And we write the following program:

```
k := 0;
while k ≠ n do begin b[k] := k³;
                     k := k + 1
                end
```

$P0$ and $P1$ are known as *loop invariants,* for they are "invariantly true" before and after each iteration of the loop. One understands the loop in four steps: (0) show that the loop invariants are true just before execution of the loop; (1) show that each iteration leaves them true, so that they are true before and after each iteration and thus upon loop termination; (2) show that the desired result R follows from the loop invariants and the falsity of the loop condition; and (3) show that the loop terminates.

The process used here to argue about correctness should be used to argue about the correctness of *every* nontrivial loop. It is the programmer's task to annotate each nontrivial loop with the necessary loop invariants, because they are a necessary part of understanding the loop. It may seem like a lot of work, but, we maintain, it is simply formalizing what a programmer does anyway when reasoning about why a loop works.

We now have a correct program. However, it uses exponentiation k^3. To get rid of it, we simply introduce a fresh variable x (say) and its definition:

$P2$: $x = k^3$

With this new loop invariant, we can replace the assignment $b[k] := k^3$ by $b[k] := x$. However, $P2$ also depends on k. Before k is increased by 1, x has to be changed to contain $(k + 1)^3$ in order to maintain the truth of $P2$. Since

$$(k + 1)^3 = k^3 + 3 \bullet k^2 + 3 \bullet k + 1$$

and since $x = k^3$ just before each iteration, x can be changed to contain $(k + 1)^3$ by executing the assignment

$x := x + 3 \bullet k^2 + 3 \bullet k + 1$.

This yields the program

```
k := 0; x := 0;
while k ≠ n do begin b[k] := x;
                     x := x + 3•k² + 3•k + 1;
                     k := k + 1
                end
```

The program still contains exponentiation and multiplication. Repeating (twice) the process that introduced x yields the program

```
k := 0; x := 0; y := 1; z := 6;
while k ≠ n do begin b[k] := x;
                     x := x + y;
                     y := y + z;
                     z := z + 6;
                     k := k + 1
                end
```

where there are five loop invariants:

$P0$: $0 \leq k \leq n$
$P1$: (**for all** i: $0 \leq i < k$: $b[i] = i^3$)
$P2$: $x = k^3$
$P3$: $y = 3 \bullet k^2 + 3 \bullet k + 1$
$P4$: $z = 6 \bullet k + 6$

SOLUTION TO PROBLEM 2

The formal specification, using some notation that should be fairly obvious, is

Precondition P: $0 \leq n$
Postcondition R: $s = MAX(i,j: 0 \leq i \leq j \leq n: S_{i,j})$

The program is

```
k := 0; c := 0; s := 0;
while k ≠ n do begin
    c := max(c + b[k], 0);
    s := max(s, c);
    k := k + 1
                end
```

The program is understood in a manner similar to that described for understanding the cube program, using the following loop invariants:

$P0$: $0 \leq k \leq n$
$P1$: $s = MAX(i,j: 0 \leq i \leq j \leq k: S_{i,j})$
$P2$: $c = MAX(i: 0 \leq i \leq k: S_{i,k})$

discovered that a statistician was using a program for this problem that required time proportional to n^3 —the program was actually computing the sums of all segments of the array. Several days later, Bentley returned with an algorithm that required time proportional to n^2, and later one that required time $n \bullet \log(n)$. Another statistician then showed Bentley a program, similar to ours, that required time proportional only to n. During a visit to Cornell, Bentley asked us to write a program for his problem. Several of us who were experienced with the programming methodology came up with the program, independently, in about a half-hour. (Of course, it took time to present it as cleanly as SOLUTION 2 is presented.) We never had to think of the n^2 or $n \bullet \log(n)$ algorithms; the methodology led us quite directly to the solution we have shown here.

WHERE TO LEARN MORE ABOUT PROGRAMMING

Some of the concepts underlying programming methodology have found their way down to undergraduate courses at Cornell such as CS100 and CS211, although in an informal manner, and more will do so as we gain experience and hone our skills. Programming methodology is taught at the upperclass level in course CS400, introduced this spring, as well as at the graduate level; both courses are based on *The Science of Programming*, which was written by Gries in 1981. An additional graduate course, CS613, extends the concepts to deal with *concurrency*, which arises in operating systems, networks, and databases, where various programs are executed simultaneously and communicate through shared data or message-passing. Schneider is writing a text on the subject of this course.

Because of the youth of the field, computer science enjoys the problem that many research results are incorporated quite rapidly into education. Last year's research problem has moved into this year's first-year graduate course and will be in next year's senior-level course. Our students, even in the early undergraduate years, are learning about a relatively new discipine as it develops.

The practicality of theoretical research is demonstrated every day not only in advanced computing centers, but in university classrooms where tomorrow's practitioners are learning their skills.

David Gries is chairman of the Department of Computer Science and a specialist in programming methodology (see the biographical sketch on page 11).

Fred B. Schneider is an associate professor in the department. A specialist in concurrent programming, operating systems, and distributed systems, he has also written another article in this issue in collaboration with three of his colleagues (see pages 18–22). He is on the College Board Committee on Advanced Placement in Computer Science, which prepares an examination that reflects much of what is taught at Cornell in the introductory courses.