Springer Books on PROFESSIONAL COMPUTING

Leon S. Levy

Taming the Tiger

Software Engineering and Software Economics

Springer-Verlag

TP31
L668

AT&T

Taming the Tiger

Software Engineering and Software Economics

Leon S. Levy

AT&T Bell Laboratories

Springer-Verlag
New York Berlin Heidelberg
London Paris Tokyo

Leon S. Levy
AT&T Bell Laboratories
Warren, NJ 07060

Series Editor
Henry Ledgard
Human Factors Ltd.
Leverett, Massachusetts 01054
U.S.A.

With 9 Figures

Printed and bound by R.R. Donnelley & Sons, Harrisonburg, Virginia
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-378-96468-1 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-96468-1 Springer-Verlag Berlin Heidelberg New York

Acknowledgment

As usual in any scientific endeavor, many people and institutions contribute directly and indirectly to any research effort. Some have a larger share which should be noted:

- Professor J.R. Clark of Fairleigh Dickinson University who introduced me to the discipline of economics and its modes of thought and who read early versions of this work and helped to improve it;
- Professor Todd Idson under whose guidance the chapter on transfer pricing was written;
- *AT&T Bell Laboratories* where I developed the notion of *ultra high programmer productivity techniques* and who encouraged this work by citing it in the distinguished staff award presented to me in 1983;
- *Ben Gurion University of the Negev* in Beer Sheba, Israel, where I was a visiting Professor in the academic year 1983–84 and where the first draft of this paper was composed, and particularly Professor Michael Lin who first furnished me with the proof of my generalized synergy theorem;
- *Fairleigh Dickinson University* where my studies in the M.B.A. program have contributed to my understanding of the managerial and economic aspects of software engineering;
- Dr. Martin Freeman of Signetics Corporation and Mike Bianchi of AT&T Information Systems who discussed many of these ideas with the author while they were in the formative stages;

- Dr. James B. Salisbury, of AT&T Bell Laboratories, who gave me moral support;
- my brother, Joel, who read portions of the manuscript and offered valuable comments; and, especially,
- my wife, Millie, without whose patience and encouragement none of this would have been possible.

Contents

Chapter 1

Introduction

A small program is presented to motivate the concerns for programmer productivity and program quality that are the central issues of this set of essays. The example is one which demonstrates the *performance* aspect of programming.

In order to achieve program quality, where a program is understood and known to be correct, we need a *primary program description.* This primary program description not only describes the program but is also used to generate the program. The method of applying primary program descriptions to produce programs is called *metaprogramming* and is described in Chapter 3.

In the later chapters, we show how the method can be analyzed from an economic point of view to address the issues of productivity as well.

Introduction

In thinking about programming over the last decade, I have concluded that very little is *known* about the process of programming or the engineering of software [1]. The consequence of having very little established truth to use as a basis for thinking about programming is that almost every conclusion must be reasoned out from first principles. Also, you cannot rely solely on textbooks but must use experimentation and direct observation to gain some experience with which to proceed.

It is also likely that if you proceed to examine the reality, you will conclude, as I and others have, that much of the accepted practice in the field is grossly inefficient. Moreover, in many cases, the quality is not very high, neither in the products nor in the documentation.

In the next few pages I talk through the development and modification of a very small program, less than one page at double spacing. The design and analysis that I discuss is a model of the development of larger programs. This process is very labor intensive and, consequently, the productivity must of necessity be low and the quality difficult to control.

---

[1] Much of the material in this introduction is taken from [Broome].

The programs that most interest me are several orders of magnitude larger than the example program. If we are to find a way to radically improve both the quality and productivity of programming, then we must find a way to make programming less labor intensive. However, before considering solutions let us examine this small program fragment:

Example. The following "toy" program should serve to illustrate much of what is wrong with the current software practice. If I were to choose a larger program to make the point we would be overwhelmed by its sheer size. Here we can concentrate on some fundamentals.

The program is one which inverts a permutation [2]. It is

[2] A permutation is a rule which "rearranges" a set of numbers. The rule which takes 1 to 2, 2 to 3, and 3 to 1, is a permutation of the set of numbers 1,2, and 3. The inverse of this permutation is the rule which determines what number is permuted into the given number. The inverse permutation takes 1 to 3, 2 to 1, and 3 to 2. The way that the inverse permutation may be easily calculated is to arrange the permutation as follows:

1	2
2	3
3	1

To determine what the permutation is we can look for a number in the left hand column and find the number that it is permuted to by moving across the row into the right hand column. To determine the inverse problem, look in the right hand column to find the number and then move across the row into the left hand column.

convenient to represent such a permutation of numbers as an array, `A`, where `A[i]` stores the number to which `i` is permuted. In practice, if the set of elements being permuted is not too large one can construct the inverse permutation as a second array, `B`, and then write a simple program which proceeds through the array `A` and for each index `i` in `A` makes the assignment `B[A[i]] := i`. Thus, `B[A[1]] := 1`, or `B[2] := 1`. This method uses two memory cells for each element of the permutation, one cell for the original permutation, and one cell for the inverse element.

If the set of elements being permuted is large and you are interested primarily in computing the inverse permutation, then you may write a subroutine to invert the permutation, *in place*. While this subroutine is running, the values stored in the elements of `A` will be unpredictable, but when the subroutine has concluded, the element `A[i]` will contain the inverse of `i` under the permutation. The in place permutation inversion will use only one cell for each element. (Of course, after the inverse permutation has been computed the original permutation will not be available.)

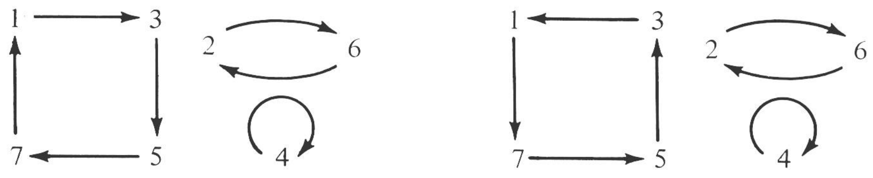
Readers interested in more information about permutations, can see [Levy 1980c].

Here is the body of a subroutine to invert a permutation:

```
for m := n step -1 until 1 do
begin
    i := A[m];
    if i < 0 then A[m] := -i
    else if i <> m then
    begin
     k := m;
     while i <> m do
     begin
         j := A[i]; A[i] := -k;
         k := i; i := j;
     end;
     A[m] := k;
  end
end
```

Program 1. Invert A Permutation in Place

Now each permutation consists of one or more disjoint cycles. An example of a permutation, A, with its cycle structure is shown in Figure 1.1.a. The corresponding inverse permutation, B, with its cycle structure is shown in Figure 1.1.b. The cycle structure of B is



(a) A permutation (b) The inverse permutation

Figure 1.1 A Permutation and its Inverse

similar to **A** except that the arrows are reversed.

The algorithm of our initial in-place permutation inverter can be described as follows:

```
for each m between 1 and n do

     if the cycle beginning at m
                  has not been inverted then
     invert and mark
           each element of the cycle (except for m)

     else remove the marker from this element.
```

This algorithm inverts one cycle at a time, marking each element with a negative sign (–). When the algorithm later encounters an element with a marker, the marker is removed.

Program 1 has few variables. The idea of the algorithm is simple but the program causes us difficulty. How do we know that the program is correct? How can we retain the link between the algorithm and the program so that when the program is written the design is not thrown away?

The communication medium between the programmer and the computer is often the source of the problem. When we must contort our ideas to fit the syntax of a particular programming language, the idea is sometimes lost. Luckily we are beginning to realize that a program's purpose is not to instruct the computer but to have the computer execute our program.

Nonetheless, even if we accept the constraints that machines and programming languages impose on us, something can still be done to better represent the idea of Program 1.

One major cause of errors in a program is the concern for (machine) efficiency. Only if we know that a program is correct can we then be concerned about efficiency. If a program is not correct, it matters little how fast it runs.

In line 5 of Program 1 there is a test for `i <> m` so that we can avoid inverting a singleton cycle. But the average number of singleton cycles in a permutation of size n is just 1, [Knuth]. So this attempt to increase efficiency has actually decreased efficiency!

Because of the early introduction of the variable, `i`, it is not readily apparent that the test `i <> m` is a test for a singleton cycle. An attempt such as this to avoid reevaluation of `A[m]` is usually unnecessary because most compilers can recognize a duplicate expression and avoid a recalculation for us.

In order to obtain a better program we should return to the design stage. Another outline of the algorithm is:

```
for m := 1 to n do
      if cycle at m has not been inverted  then
        invert and mark every element of the cycle
      remove the marker from this element
```

It is more natural to take m from 1 to n than to go the opposite way. We can avoid special cases by marking (with a minus sign (-)) *every* element of the cycle, whereas Program 1 leaves element m unmarked.

The most difficult part of Program 1 is that for chasing around a cycle, especially the part for moving from one element to the next. All ways of moving around a cycle are similar in that they have the following outline:

```
start at m
while not done do
begin
    process the current element
    move to the next element
end
```

As an example, consider the problem of finding an element x which we know to be somewhere in the cycle. The program part:

```
i := m;
while A[i] <> x do
     i := A[i];
```

will return i such that A[i] = x. Other than the array, A, and the value, x, this program part requires one other variable, the variable i. As we shall see, the number of additional variables increases with the complexity of the process.

As a second example of loop chasing, assume that we want to break a cycle into all singleton cycles. The program part

```
i := m;
j := A[i];
while A[i] <> i do
begin
     A[i] := i;
     i := j;
     j := A[i];
     end
```