Professional Software *Programming Practice*

HENRY LEDGARD *with* JOHN TAUER

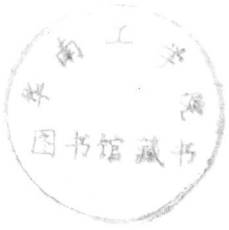
Volume II

Professional Software

Volume II

Programming Practice

HENRY LEDGARD *with* JOHN TAUER



Addison-Wesley Publishing Company

Reading, Massachusetts • Menlo Park, California • Don Mills, Ontario
Wokingham, England • Amsterdam • Sydney • Singapore • Tokyo
Madrid • Bogotá • Santiago • San Juan

Library of Congress Cataloging-in-Publication Data

Ledgard, Henry F., 1943—
Professional software.

Contents: v. 1. Software engineering — v. 2.
Programming practice.
Includes bibliographies and indexes.
1. Computer software—Development. 2. Electronic
digital computers—Programming. I. Title.
QA76.76.D47L43 1987 005 87-1760
ISBN 0-201-12231-6 (v. 1)
ISBN 0-201-12232-4 (v. 2)

ABCDEFGHIJ-AL-8987

Foreword

In asking me to write a Foreword, the author has put me on the spot in a sweet but singularly painful fashion: if a writer needs a psychiatrist (at least afterwards, if not before), then in this position a person needs a priest. Confess, have you read the book before writing the Foreword, or is this all a friendly platitude, a backscratching exercise, extracted through the flattery of invitation?

Well, here I am on fairly clear ground. I have not only read the book, but I have heard most of its contents in one or another form. The author evolved some, but assuredly not all, of this material and his basic approach to good software engineering practice when he gave "master class" courses, under my direction, for one of the world's largest and most prestigious electronics companies.

The impact of Henry Ledgard's approach on an audience of computer scientists and amateur programmers (the categories overlap somewhat) was remarkable. Students came to jeer and stayed to cheer. The normal preconception went something along the lines that: "We've all written programs and we don't need this chap to tell us how to structure software, use an order-code, comment our documentation, name our variables, and so

forth. That's all cosmetics, subjective stuff, isn't it? No two people will ever agree on cosmetic issues; it's a matter of style..." I have seen the author taking on a pack of graduate computer scientists with their bloodlust registering an off-scale value, and amateur programmers erecting a gallows in the corner for him (it fell down, naturally, when they alpha tested it on Fred). Later, when I had made them quality-assure each other's software—mere programs would have been a better description—they were more subdued.

"That Ledgard told us all about this, didn't he?" they asked. "About the trouble we'd have with old Fred's drivel. There's absolutely no way I could touch that program—nor could Fred, incidentally. Why doesn't Henry Ledgard write a book about it—something we could really get our teeth into...?"

He has, and this is it. It should be dedicated to Fred (or Fredrika) the phantom programmer, and nobody with serious pretention to a career in software engineering (or its management) should miss reading it. That is a true confession.

Allen Macro, Capelle aan den IJssel, Holland

Preface

Volume I of this work treats a number of process issues (e.g., the software lifecycle and programming teams) in software engineering. This, Volume II, treats the code itself.

Over the years, I have reviewed, read, or worked with a considerable amount of code. The code has been written by both students and professionals in many different languages. I have seen the impenetrable code, the struggle to modify parts of the software, the expensive throw away, and the serendipity.

This work is a result of this experience. It is my considered view of some fundamental issues in programming craftsmanship.

What is Professional Practice?

Software development is certainly a matter of design—that is, the choice of appropriate algorithms, data structures, and general organization. But the quality of software is not only a technical matter. If one has a proven algorithm or a proven data structure, how can one show it in the clearest possible manner? What is a good decomposition of the overall approach?

How do we write programs so that the intent is clear in the written form? The journals are filled with papers that identify the problem of software as being the maintenance of it.

As I mentioned in Volume I of this work, there is an identifiable difference between a quantitative and a qualitative approach in answering the question, what are we doing? The difference may not be as sharp as the schools that have emerged in the social sciences, but the dichotomy in programming certainly exists. Our discipline is, in part, certainly quantitative. The concern for power, new tools, more efficient algorithms, new language designs, support for program correctness, and the like——these are vital. But we must also sharpen our understanding of qualitative issues, for we can easily be shadowed in a forest of innovations.

This work addresses questions like these:

- What is a good procedure?
- What is a good package?
- Are global variables that harmful?
- Should we strive for more comments in programs?
- Why is naming difficult?

The general objective is to look as deeply as we can into the quality and craftsmanship of professional programmer's most critical product—the code itself.

Issues such as commenting, program layout, and naming seem like modest, even humble, tasks. But, in practice, I submit, such humble issues have a great impact. In any given project, these issues arise with such frequency that, in some cases, they can obscure the focus and substance of the very project itself. Good ideas can be buried in impenetrable code. The unwitting programmer, who is not aware of the scope and subtlety of these issues, may not even realize the self-created complexity of the result. And when the product of this effort is passed on, others, too, are dragged along.

Good professional practice is both quantitative and qualitative.

Pascal, Ada, Modula-2, and C

The programming examples in this work are written in an extended variant of Pascal. The variant includes a notation for packages and uses underscores in variable names. This language is meant as a communication language.

Each higher-level language has its own syntax, semantics, and constraints. Where these languages motivate special concerns, examples in

specific languages are given. These languages are standard Pascal, Ada, Modula-2, and C.

This book has been typeset using a monospaced font (both bold and nonbold) for programs. Monospaced typefaces, with or without bold, are most appropriate for programs. They promote readability and, I believe, give the best appearance for printing programs.

The Preface to Volume I provides acknowledgments to those who have assisted or inspired this two-volume work.

H. L.

To The Reader

This book is my considered opinion about professional practice. It is derived from teaching students and professionals and from participating in numerous software efforts. The thoughtful reader may, in places, have good reason to hold other views. This should not confuse our common goal, the pursuit of excellence.

Contents for Volume I Software Engineering

Contents

1

Something is Wrong, Hear

When a friend of mine was nearing the end of his graduate studies, a kindly mentor gave him advice roughly along these lines: "No matter what you do in the future, whether you choose to be a programmer, a writer, a teacher, take time—nay, *make* time—for reflection." This work is, I suppose, a product of many hours of reflection, wondering about the practical issues that programmers face each day and the way they deal with them.

Programming requires that we translate the language that we speak every day into a language that works for a computer. In so doing, we are describing reality in a different, abstract way to solve problems. But there is a craftsmanship in programming that is often more difficult to grasp than the abstraction. It seems to me that terms such as clarity, simplicity, balance, symmetry, and precision are useful synonyms for the overworked adjective "beautiful."

These beauties come to mind in Lincoln's Gettysburg Address. Suppose for some reason he had written:

> Eighty-seven years ago, our antecedents created a novel nation-state in this hemisphere, the principles being that the citizenry should live in freedom, and that every citizen would be equal in every manner to one another.

Says the same thing, doesn't it? But something's wrong. Somehow the simplicity and elegance are missing. And yet, today, no sportswriter would dare to submit copy to an editor saying that the course record at Such-and-Such Golf Club was broken when an unknown amateur shot "three score and three." There are words for the times and times for the words.

The substance of this discussion is that, in many moments of reflection on this issue or that abstraction, on this line of code or that program, I have come to feel that something is wrong here. A few cases are easy to spot:

- An expression that carries on for four or five lines.
- Two lengthy procedures that are identical except for a line or two.
- A subroutine that is five times longer than it should be.
- An expression that is illogical to read.

Others—in fact most—are more difficult to see.

So, before we begin the subsequent essays on programming practice, some examples are offered. The purpose of these examples is threefold:

- To sketch the territory of issues treated in this work.
- To raise certain questions about programming practice.
- To sharpen awareness of program quality.

On each example, you might ask: Why does a programmer take a particular route in the first place? What was the underlying logic? And why, when someone *else* looks at the work, do problems of reading, interpretation, and understanding arise?

Some of the examples that follow are short, innocuous statements—others are longer and require some insight. That is, if you take some time for reflection about what is being attempted in each example, you will be asking yourself, "What is wrong here?" You may answer by posing other questions.

For these first three examples, let me raise questions that I have asked. Let us examine Example 1.

- Is the procedure misnamed?
- Does it update a table?
- Does it do other things besides printing?
- Is it a procedure that has one purpose or multiple purposes?

If you can't give a procedure a simple and clear name, there is something wrong with its formulation; that is, it is not a one-purpose procedure. Look at the example one more time. In name `UPDATE_TABLE`, we should be *updating,* but we are, in fact, *printing!* Something's wrong here.

Example 2 requires a little more mental concentration.

Example 1

```
procedure UPDATE_TABLE;
{ This procedure prints... }
```

Example 2

```
procedure CHECK_LINE(LINE:STRING; LINE_LENGTH: INTEGER;
                     STR: STRING; STR_LENGTH: INTEGER; var
                     FOUND: BOOLEAN; var START_POS: INTEGER);
```

Example 3

```
SET_LINE (12);
WRITE ('Enter next amount:');
READLN (INVAL)
```

- Which parameters are input?
- Which are output?
- Is there a punctuation error?
- For the *reader,* what does it *look* like? Is it messy, or does it have balance and symmetry?

Well, there is some logic here. The first four parameters are input, and the last two are output. But why does the reader have to look for them? The programmer gave little thought to how the reader could identify the parameters without undue effort.

Example 3 presents a prompting message given to the user on line 12, and the value is supplied immediately after the prompt (with no intervening space).

- What is special about line 12?
- Will the user be uncomfortable to see `amount:51`?
  (*Hint:* Why not `amount: 51`?)
- Is `INVAL` a good name?
  (*Hint:* Why not `INPUT_VALUE`? Or better, call it what it is: `DEPOSIT` or `WITHDRAWAL`. If it's a `COW`, why call it a `BOVINE`?)

The purpose of this chapter is not to answer all the questions or, for that matter, to ask them. As you proceed through the subsequent essays, some questions (but not all) will be raised and answered. For example, here is a variation on the mysterious numbers of Example 3.