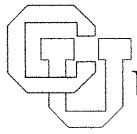
Five Essays on Software Engineering

Bruce K. Haddon

CU-CS-131-78



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

FIVE ESSAYS ON SOFTWARE ENGINEERING

by

Bruce K. Haddon
Department of Electrical Engineering

Department of Computer Science
University of Colorado
Boulder, Colorado 80309
U.S.A.

CS-CU-131-78 June, 1978

CONTENTS

Foreword

These essays were written while I was a postgraduate student taking the course CS582, Software Engineering. They were written to fulfill part of the requirements for this course, and I am indebted to Professor Paul Zeiger both for the inspiration provided by his conduct of this class, and for accepting "what comes naturally" as evidence of having performed the required classwork.

There is really no common thread to these essays. They are simply my response to a number of different issues raised in the course of classroom discussion, where I felt that I had more to say than could be reasonably explained in the classroom situation. They will have served a purpose if they spark the same response in the reader.

Bruce K. Haddon
June, 1978

Towards a Definition of "Software"

"defini'tion, n. Stating the
precise nature of a thing ... "
The Concise Oxford Dictionary.

"a program (...). That is the thing
commonly produced in garages, "
Frederick P. Brooks, Jr.
in The Mythical Man-Month.

The main problem with definitions is that they are usually conceived in order to fulfil some, dare I say ulterior, purpose. In my schooldays, the debating master warned against defining one's terms too narrowly, as this would define the opponents "out of the debate" and the argument would never be joined. On the other hand, too loose a definition does not allow a meaningful discussion that is free of bothersome qualifications. In short, a useful definition is one that allows meaningful statements to be made in the chosen area of discourse.

Now, in attempting to define "software" we undoubtedly have it in the back of our minds that we would like to say meaningful (and perhaps even interesting) things about "software engineering." (Let it be admitted that this is a chicken and egg situation, but pass it by for the moment.) Recognising in advance the kind of things we would like to say enables some bounds to be placed upon the meaning of the term to be defined. For a start, we know that "software" is a more general term than algorithm. Also, we are really only interested in using the term "software" in a computer context, and we wish to exclude from our definition that which we call hardware. Thus, we are looking for something between "a specification or description of a method for solving a problem" and "an entity that carries out operations upon data representations."

We probably also have a feeling that the production of software is related to an activity that we call and recognize as programming. This is compatible with the idea that "software engineering" has some concern with organizing the programming task. This means we immediately think of the tools and methods of programming--pencils, paper, erasers, support documentation, user manuals, etc. And since we hope that the output from all this activity is "software" we must really ask, and eventually answer, how much of all these things make up the actual product.

At this point, I would pause to consider whether the term "software" is really intended to mean something more (or less) than the word "program" does. I think that ten or so years ago I would have taken the position that the word "software" was

descriptive of that class of programs that was as essential to the operation of the computer as was the hardware, that is, those programs that are now classified as "systems software." As people now frequently and quite happily speak of "applications software," "user software," "software packages" and the like, I feel that we must recognize that "software" includes all programs. The use of the word "civil" in "civil engineering" bears the same relationship to a bridge as "software" in "software engineering" does to a regression program. It is an abstract term that makes less commitment to the characteristics of the physical objects involved.

Following the same line of thought, I think that we all feel that large programming projects attempt to produce "software." But at the other end of the scale, consider a person carefully entering strokes on a so-called "key-programmable" hand calculator. Is this person creating "software"--and if not, why not? If the individual is working alone, for his own ends, then his product is free of all the paperwork we usually associate with software production, and it would not make any economic sense to insist upon written records. (For example, an experienced user of such a calculator who knows that a quadratic equation may mave two real roots could probably create a program to find them faster than he could find, understand, copy and use someone else's.) Yet it is obvious that the packages of programs published by the calculator manufacturers are as much subject to the dictums of "software engineering" as any other programming product. All of this assuming that key-programmable calculators are allowed as instances of computers.

In answer to this last question, I think we must allow that they are. If someone throws a log over a stream, it is a bridge. Such a bridge may be of marginal interest to civil engineers, and they may not feel that such a structure exemplifies the finer points of their art, but its essential characteristics are no different from its bigger and costlier brothers. The analogy holds for key-programmable calculators and five-step programs.

There is another marginal situation that may reflect some light on the meaning of the term "software." Let us assume some alleged piece of software is recast as a module of sequential electronic circuitry. If this is accomplished by the burning of a ROM chip, it would probably be agreed that an irreversible recording has been made, a recording little different (size excepted) from that made by (irreversibly) punching holes in a deck of cards. A "software change" requires only that a new recording be made. If, however, such a change requires the addition or removal of particular electronic components, then it would also probably be agreed that a transition has been made from software engineering to hardware engineering, as the economics of circuit design, state

minimization and component costs and utilization will have become relevant. To state the same thing a little differently, the maintenance manual must now speak of electronic components rather than source language statements. A software product was created, and exists, but its role was that of a design tool for the hardware designer. As such, the documentation associated with this software product will look very different to that associated with software items that are end products in themselves.

The term "software" has already occurred many times in this discussion, but as yet has not been defined! It has been used to refer to representations of purposeful collections of computer instructions, the representation having been chosen in a way that ultimately makes the instructions available to a computer for execution.

This definition seems to suit our needs quite nicely. We do not really want to label as "software" something that cannot be executed--a handwritten program is not yet software, even though it is a long step towards its eventual production. A high-level language is a means towards more compact representation, so we can meaningfully speak of software written in a high-level language (and deduce other properties). The word "purposeful" connotes intent, which enables discussion of software with errors (which would be impossible if correctness were part of the definition -- indeed in this circumstance it would be difficult to prove that a given item was a piece of software at all) but allows us to exclude sets of random statements. We can recognize compilation, assembly, and link editing as transformations of representation that (hopefully) do not alter the intent of the software. "Software engineering" now has as its subject matter the practice and methods for generating and manipulating representations of this type. And finally, we can see that the thing defined is that which is produced by "programming."

Lastly, we can return to the question of whether the associated documentation should be considered to be an integral part of the software. The definition given above excludes it by the insistence on executability. Is this too narrow, or should the definition be expanded? The calculator example showed that it is meaningful to speak of software that has no documentation. The discussion relating to hardware illustrated that the type, style, and amount of documentation was determined by the intended use of the product, not the product itself. It is my opinion that the "engineering" part of "software engineering" by itself implies the existence of specification, design, use and maintenance documents to the degree that is commeasurate with the economic value of the product. All engineering disciplines have a similar requirement, and software engineering can borrow from their experience. In fact, we know that a goodly part of the effort that goes into

applications software in concerned with documentation as a problem area. To include documentation in the definition of software invites a dangerous amount of introspection.

So, in summary, I offer the following definition:

> soft'ware, n. A representation of a purposeful collection of computer instructions, the representation being transformable to a form executable by a computer.

"Perhaps it's Neither?"

With the advent of the industrial age, it was recognized by economists that there were two major types of commercial activity: the so-called primary industries (agriculture, mining, etc.) that produced raw material, and the secondary industries (manufacturing, processing, etc.) that reworked the raw materials into finished products. It was not that there was no secondary industry before this time, but that it was not economically significant to differentiate (in fact, there was little economics). Towards the turn of this century, another type grew to the point of economic significance, the service or tertiary industries (government, education, banking, maintenance, accounting, personal services, etc.).

It is obvious that computer software is not a primary product (i.e. it does not grow on trees). So now the question is: "Is computer software a good or a service?" Perhaps this question is not properly put, in that it does not have an excluded middle.

It is argued by Peter F. Drucker [1969] that a fourth economic classification is needed today, which he calls the "information and knowledge" industries. This classification includes, for example, communications media (newspapers, book publishing and libraries), education, market and economic analysis services, weather prediction, and computer software. All of these share the property that after the initial production effort, the product can be used and reused without requiring new production. A telephone cable does not have to be replaced after each conversation. If you pay a market analyst for a good tip, he is very hard put to prevent you sharing it with friends.

The magnetic tape upon which your software is written is very much analogous to a communication channel, and what you receive is information on how to perform a given task (provided you have the tools), this information embodying the knowledge and understanding that the writer of the software had of the methods of carrying out the task.

Drucker's thesis is that the growth of these industries is now so explosive that it is outstripping the economic models that we have to describe their dynamics and our understanding of the management problems involved. This, he says, is creating a "discontinuity" in the social structure that could become even more disruptive than the Industrial Revolution--the following extracts illustrate some of the points that he makes:

"The KNOWLEDGE INDUSTRIES which produce and distribute ideas and information rather than goods and services, accounted in 1955 for one-quarter of the US Gross National Product. (This was already three times the proportion of the national product that the country had spent on the "knowledge sector" in 1900.) Yet by 1965, ten years later, the knowledge sector was taking one-third of a much bigger national product. In the late 1970s it will account for one-half of the total national product. Every other dollar earned and spent in the American economy will be earned by producing and distributing ideas and information and will be spent on procuring ideas and information." (p321.)

"Traditional economic theory knows only 'commodities'. It does not know 'products'. A commodity is defined entirely by physical characteristics. Competition, therefore, is always between units that are clearly defined and distinguished and differ from each other only by their price. Products are, however, much more complex. They are usually not capable of being defined in physical terms alone. They are usually differentiated in the value they offer the buyer--And there the traditional commodity concept is simply not adequate." (p205.)

"Teaching is the only traditional craft in which we have not yet fashioned the tools that make an ordinary person capable of superior performance. In this respect, teaching is far behind medicine where the tools first became available a century or more ago. It is, of course, infinitely behind the mechanical crafts where we have had effective apprenticeship for thousands and thousands of years." (p42.)

"The information industry will create tremendous employment opportunities. We need, for instance, in the United States about 1 million computer programmers between now and 1975--as against 150,000 to 200,000 to date. The computer programmer is to the information industry what the worker on the assembly line was to the mass-production industry of yesterday: the semi-skilled but highly paid, highly productive worker." (p43.)

"Clearly we do not as yet know how to obtain economic performance from knowledge. We also do not know how to satisfy the knowledge worker and to enable him to gain the achievement he needs. Nor do we as yet fully understand the social and psychological needs of the knowledge worker.

"That we do not yet know how to manage knowledge workers for performance is hardly surprising considering how recent the shift to knowledge work has been. After all, it is less than a hundred years since we first began to concern ourselves with managing the manual worker." (p349.)

"We could, therefor, hardly expect to know how to define, let alone measure the output of knowledge work. For this task we need definitions--not to speak of measurements--that are quite different from those that we have learned to apply to manual work. The most useless and wasteful effort is that of an engineering team that with great speed, precision, and elegance turns out drawings for the wrong product. Knowledge work is not easily defined in quantitative terms, and may indeed be incapable of quantification altogether. The computer, it is reasonably certain, cannot measure the work of the programmer who runs the computer." (p350.)

Reference

Drucker, Peter F. The Age of Discontinuity. Pan Books Ltd, London (1969) pp477.

In Search of a Paradigm

> "'3' means '2+1', and '4' means '3+1'. Hence it follows (although the proof is long) that '4' means the same as '2+2'. Thus mathematical knowledge ceases to be mysterious."
>
> Bertrand Russell,
> in "The Philosophy of Logical Analysis".*

There have been many attempts to characterize a programmer's activities by likening them to those of other professions or crafts--this is yet another. Other endeavours have been aimed at formulating the structure of the ideal programming team, a structure intended to enhance the environment in which the programmer's activities take place. There is obviously a very strong interaction between the perceived nature of a programmer's work and the team structure considered ideal. Two strongly antithetic views on team structure are advanced by Weinberg [1972] and Brooks [1975] which undoubtedly mirror fundamental differences in appreciation of the programmer and the programmer's task. Before looking at these differences, let me first examine some possible analogous occupations.

The Programmer as Draughtsman

Perhaps it would be better to choose a Draughting Engineer then simply a draughtsman, but nevertheless, let me pursue the analogy. The draughtsman will often receive from the engineering designer a sketch of some item, with some notes as to critical dimensions, important relationships between parts of unit, and a brief outline of the function of the thing. The draughtsman job is then to produce a neat, tidy, understandable drawing which has all the detail filled in--exact radii and positions of holes, machining tolerances where they are critical, depth and shape of weld fillets, and many other essential things that must be taken care of if the item is to fulfill its designed functions. The draughtman's objective is to be able to hand the drawing to the machinist or diecutter or someone who will be able to create the actual object without further reference elsewhere.

Now, this is frequently the type of task set for a programmer, and one, incidentally, to which I have found that programmers respond very well. The design document specifies a rough outline

*Bertrand Russell, A History of Western Philosophy, Allen & Unwin (1946).

of the program or module, its purpose, and some indications of possible algorithms. The programmer is left all the "important" decisions, such as the final choice of algorithm, induction of the final (sub)modularization, and all the small details, such as what variables are needed, just where in the code the incrementation of a counter is done, and so on.

The analogy can be pushed a little further. If the engineering product is more complex, then a team effort may be required--and a communication problem is introduced. Draughtsman Bob says, "Hey, Jim, I need to fasten this piece, I'll put a bolt through here." Jim replies, "No, the nut will get in the way of my swing arm here." Bob cogitates and answers "OK, I'll use a countersunk head on that side, and put the nut on this side." The same interplay can be observed when programmers decide what information is to be passed across a module interface.

The analogy breaks down a little at the point where the product fails to work as hoped. In the engineering situation, the component that does not work is examined first, whereas in the programming case, first recourse is to the program text--the "drawing". (Although, fifteen years ago, it is likely that the first thing examined would have been a core dump, that is, the actual software product that failed). This difference seems to lead to an assumption that in the engineering case there was a "designer's" error, whereas in the software instance, a "programmer's" error. This view may not be universally true, but I have seen it happen more often than not.

The Programmer as Mathematician

Both the programmer and the mathematician produce a "paper" product. Both must produce their product in a form that will withstand the most rigorous scrutiny, and let's face it, a scrutiny more aimed at showing the reasoning embodied in the product is incorrect rather than correct. Both are at the mercy of typographical errors, although the programmer is in a slightly more fortunate position in this regard.

There is some thought that the product of the programmer is more complex than that of the mathematician, a position that I do not fully accept. The longest mathematical proof that I myself have ever produced ran to 63 pages of preliminary theorems and lemmas, the actual "proof" (to show that two definitions of certain nilpotent ideals in an algebra were in fact equivalent) occupied just the last ten lines. Only rarely have I written a program of this size. The longest mathematical proof I have ever experienced was that of the Prime Number Theorem, which took a lecturer 36 hours to develop (12 weeks at 3 hours a week), during

which time I made almost 200 pages of notes. Again, most of this time (space) was occupied by developing preliminary theorems, which have an existence in their own right. They formed a necessary background to the actual proof in question. In the software engineer's terminology they constitute the support modules embodying the abstractions required for the higher level application.

Now this is where the mathematician appears to have the edge on the programmer/software engineer. The mathematician appears to have the capacity and the inclination to formulate abstractions that have a greater conceptual unity, that have a better intuitive appeal than are (yet) available to the programmer. I can see two possible contributing factors to the mathematician's advantage.

Firstly, the mathematician has access to a better (more sophisticated?) notation in which to formulate his problems. And the overwhelming advantage of this notation is that itself possesses formal properties, that is, it can be demonstrated that certain manipulations of the symbols can be performed irrespective of the "meaning" of the symbols. Such manipulations do not produce anything new, but can have a dramatic effect on our perception of intent. For example:

$$ax^2 + bx + c = 0$$

and

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \quad a \neq 0$$

are formally equivalent, yet we all feel the second tells us something that the first does not. Many would argue that the second equation is the "solution" to the first, yet it is no different in nature to:

$$b = -ax - c/x, \quad x \neq 0;$$

which doesn't tell us anything we care to know. But they are both simple restatements of the original. In the same way:

```
while b ≠ 0 do begin S end;
```

and

```
if b ≠ 0 then repeat S until b = 0; (*b functional*)
```

are equivalent. (Note that the caveat "b functional" is similar in nature to the "x ≠ 0" or "a ≠ 0" introduced by the rewriting of the mathematical formulae.) But for some reason one of these programming language statements is no more revealing than the other. Some would take the position that the following "definition" (of while) is even more obscure:

```
            goto loop_end;
loop_start: S;
loop_end:   if b ≠ 0 then goto loop_start;
```

At the very least it seems certain that our notation is not working for us--it may be as one popular view has it, that our programming languages are the biggest obstacle that we have to overcome.

The second factor is related to the position of mathematics in our society and educational system. A practising mathematician will have learnt most of his basic tools and methodology in his high school years, and many of his most basic concepts much earlier. He will have learnt by drills and repetitive exercises, so that much of his knowledge will have become "intuitive". In fact, mathematicians place a high estimation on intuition, a measure of the value of an abstraction or notation being its intuitive appeal. Many programmers, on the other hand, avoid writing the same or similar programs if at all possible, thus denying themselves the opportunity to learn through practice, and they also find that "intuition is frequently unreliable."* These positions, I feel, arise from the superficiality of our understanding of the arts and sciences of programming. As Wulf remarks, "... if we really understood a program, we would understand why it is correct or incorrect."**

The programmers I have known have not been mathematicians, yet their mathematical facility has been more functionally sophisticated than their programming skills. The differentiation of a function of a function offers less challenge than deriving the loop invariant for a for statement. The latter, to the programmer, surely has greater utility. The teaching of programming earlier in the education curriculum may go someway toward correcting this, and the improving techniques of abstraction will provide some help. A 5000 year history to draw upon would also be an inestimable advantage.

The Programmer as Novelist

No matter how much a writer writes to fulfill some inner compulsion, publication is the only objective test of merit. And the corollary of publication is criticism. Although there is an adversary relationship between writer and critic, it is a productive relationship. Ideally there is an exchange between the protagonists that sharpens both their skills and their appreciation of each other's roles. Even when less than ideal, both can learn by listening to the other, even if they do not particularly like what they hear.

---

* Barbara Liskov and Stephen Zilles. An Introduction to Formal Specifications of Data Abstractions. In Current Trends in Programming Methodology: Volume 1 (Raymond T. Yeh, ed.). Prentice-Hall, Inc., Englewood Cliffs, N.J. (1977).

**William A. Wulf. Languages and Structured Programs. op. cit.

Whether a programmer likes it or not, he is in the same situation. The behaviour of a program intended for any but personal use is subject to examination by many critics (with a vested interest in performance). A successful program will be under pressure for extension, an unsuccessful one for correction, so inevitably the actual source text will be subject to some analysis. This analysis will be critical, in both the dramatic and literary sense. A programmer must face this prospect at the outset, and write to be read.

Now, there is a popular impression that programmmers are naturally secretive about their code. This is contrary to my experience--I have seen many programs that were not what I would call "readable", but never a programmer who was not willing to show and explain every nuance and subtlety of his creation. The main difficulty encountered appears to be the finding of others who will listen. Weinberg (p. 6) comments that programmers are usually not avid program readers. A programmer seeking to show off his code is not really looking for someone to say that the method won't work, or is implemented incorrectly, or there is a known better method. But these are the sorts of things that a critic will say anyway (and we noted above that the intrusion of a critic is inevitable) and I have seen that a programmer will accept this in return for having someone take the time to "coo over his baby."

The trick is to convince the programmmer that he will have an easier time finding readers, and that they will be less prone to false criticism, if the code is readable. And once he starts consciously improving the readability of his code, so will he improve the quality, and with that, his morale. At this point, you have a "programming novelist".

The Ultimate Team vs. The Ultimate Programmer

Weinberg is of the opinion that most programmers invest too much ego in their programs to be comfortable with exposing their programs to others, or even to cope without tension with the errors that could be detected by the computer system. To my mind, this attributes less maturity to a programmer than to an infant learning to walk. The infant has a considerable ego investment in getting onto his feet, perhaps is under strong external (parental) pressures, and has to cope with both extreme personal limitations and repeated rebuff by an inanimate environment. Yet he perseveres; the experience is rewarding in its outcome, and, more importantly, teaches lessons about the nature of life and the process of learning, both of which we consider to be necessary and valuable features of the child's development.

Brooks' picture of the "chief" programmer is a stark contrast. His programmer is a person sufficiently productive and self-confident to be able to keep up to nine other people occupied with his output. His responsibilities go beyond just the programming task. The frustrations that so worry Weinberg have no place in this scenario. Unfortunately, it is beyond my experience to know how well such a team would work, but having worked within smaller units with similar features (punch operator, secretary to whom to dictate the documentation, back-up programmer to read and independently test the code) I know that productivity is greatly enhanced by such an organization.

It is difficult to reconcile these two different views of the same "animal". I feel that Weinberg's view may have been influenced by the psychologist's tendency to examine "mental institution" cases, and dependency on data acquired from and by students subject to the ambivalences and uncertainties of the learning situation, particularly those affected by the mid-to-late sixties' euphoria with high-level languages and third-generation machines, which promised an end to the evils of earlier times. We are seeing today the effects of the let-down from this "high". Brooks is also biassed, in the other direction, by association with very competent, ultra-professionals. The programmers of my association are somewhere in between.

The Programmer as Programmer

None of the analogies outlined above really describe a programmer, for a programmer is none of these--he is, in fact, a programmer. But no matter how low he is in any organizational structure, the professional programmer does need to have the ability to bring some design talent to the project, for he will have to make all those miniscule decisions that make up the warp and weft in the fabric of the overall design. He must be conversant with the notations and abstractions of his trade, be able to make statements about and prove the properties implicit in these notations and abstractions, utilizing a well-developed intuition to guide him through the maze of complexity.

As importantly, he must interact with his environment. Not only need he cope with the limitations of himself and his machines, and learn the lessons imposed by these limitations, he should go further and expose his work to the critical examination of his peers, and be prepared to learn the lessons that that exposure will generate.

The above has not offered a definitive view of the programmer and his task. This is something that each programmer must find for himself, relating his own circumstances with his own nature and goals. Enlightenment is in the search for a paradigm--the substance adds little more.

References

Brooks, Frederick P., Jr. The Mythical Man-Month. Addison-Wesley Publishing Company, Reading, Massachusetts (1975) pp. 195.

Weinberg, Gerald M. The Psychology of Computer Programming. Van Nostrand Reinhold Company, New York (1972) pp. 288.

Reading Programs for Fun and Profit

> "Reading is one of the highest functions of the human brain--of all creatures on earth, only people can read.
>
> "Reading is one of the most important functions in life, since virtually all learning is based on the ability to read.
>
> "It is truly astonishing that it has taken us so many years to realize that the younger a child is when he learns to read, the easier it will be for him to read and the better he will read."
>
> Glenn Doman,
> in "How to Teach Your Baby to Read."*

My first (and only) formal class in assembly language programming started with the lecturer asking whether everybody had a copy of the manufacturer's manual. About five out of forty had it. So we were admonished to obtain one, and the class was dismissed. The following week, the same question was put. This time the ratio was reversed, so we were handed some sheets of exercises, and asked whether there were "Any questions?" There were none, so again the class was dismissed. It did not meet again. After a fashion, I completed two exercises, and passed the course.

Having struck up a friendship (that lasts to this day) with one of the staff programmers, I asked him to teach me to "program the machine." He handed me a listing, and told me to return when I understood it. That took two months, the listing was of the machine's monitor program. I knew how to "program the machine".

The reading habit persists. Fifteen languages later, I still start out reading programs, with a manual at my elbow to explain what I am looking at. I read critically, asking whether the code I am examining constitutes good usage, whether it reflects the spirit of the language designer's intent, whether the available language constructs are being used when applicable, whether advantage is being taken of machine features (good in assembly language programs, possibly bad in high-level language programs). Of course, at the beginning, my judgements in these matters may be

---

*Dolphin Books, Doubleday & Company, Inc. Garden City, New York (1975).

amiss, but this approach has developed a fine appreciation for the helpful comment, the transparent structure, the overall clarity of a program. It has also created an awareness of the strengths and weaknesses of various languages and language constructs for diverse applications and situations.

This method of learning can be compared to the child's learning of his native tongue. Understanding comes before formal study. Patterns of usage are copied from others. Literary appreciation is a faculty that has to be developed by continual exposure and critical appraisal. Finally, reading can become an enjoyable pursuit in itself.

I suggested in an earlier essay that programmers are only too ready to show off their code, their main obstacle being the finding of willing readers. And potential readers shy away from volunteering, since they do not know what is expected of them. This situation is in large measure the fault of our educators, for although they teach the writing of programs, they do not teach reading. How many so-called programming courses are in essence like my first assembly language course, casting the student loose in a sea of language constructs, with little or no idea how to organize them? The student is expected to communicate in (to him) a foreign language without ever having seen how the parts can be used to make a whole. From whence comes the appreciation of good usage? Or the ability to distinguish a useful comment from the obvious comment? It is my experience that it takes little to impart a sufficient appreciation of a programming language to a beginner for him to understand a well-written program, and, that almost as soon as he can do that, he can re-use statements from that program to construct another. At this point, he needs to learn to find his way around the defining manual, to check that each statement does what he thinks it does.

If we are to encourage the reading of programs, particularly in the professional ranks as a means of controlling errors and improving maintainability, the function and the duties of the reader need to be defined. I will call this person the "critic", and I intend it in the literary sense. The writer of the program I will call the "author", to complete the analogy. The author writes the program, the critic writes a critique. In managing programmers and programming teams, I have often appointed people to the role of critic, sometimes myself, sometimes trainees, sometimes more than one, and sometimes changing the critic in the course of the project. In the ensuing discussion, I will describe the critic's role, and his interaction with the author.

There are five distinct phases in the critic's job, each of which may be iterated or returned to more than once. Remember that at each phase the critic need not always be the same person,

or even one person. In the environments in whch I have applied this approach, the projects have not been overly large, the largest being the design and writing of an operating system of approximately 50000 lines. I could envisage that in a large organization, some of the critic's functions could be permanently institutionalized in the organization's structure. However, I will write simply of an author, and a critic. I lack names for the phases, so will just number them.

Phase 1 The author supplies a written description of the functions of the program, in the form of compiler- or assembler-acceptable commentary. (All the descriptions mentioned here and below are in this form, and are intended to become a permanent part of the program text. In all cases, the critic is responsible for ensuring the format agrees with the project's internal documentation standards.) The critic discusses this description with the author and requests changes until the critic is satisfied that the description says what the author intends.

Note that the critic is not responsible for ensuring that this description agrees with the program's specification. That is a task for the author's team leader or supervisor. The canny supervisor will ask to see the description only after the critic has approved it. At this point, the supervisor can be sure that the author has stated his intentions in a comprehensible form, thus easing his task of checking that the author is intending to write the desired program. This check may cause some further iteration in this phase, but better now than later.

Phase 2 The author supplies a written specification of the program's internal structure, interface specifications, data (and file) descriptions, and algorithms to be used. The critic applies the same standards to each as he did in phase 1, and further attempts to reconcile the structures and algorithms with the description agreed upon in phase 1. He should also check for internal consistency. These descriptions might be given to the critic all together, or one-by-one as they are written. Either way, phase 2 should be expected to occupy a fairly long period, as this is the time that the author will be wrestling with his overall design problems.

The critic must be shown no code at this time. This is not to say that the author may not be writing code. In fact, this is the time that the author may "write one to throw away." Writing code may help him organize the program, and to write suitably definitive descriptions. He may get away with writing the code first and the documentation later (as many programmers are wont to try) or he may have to abandon some or all of the code to satisfy the critic in this or later phases. Either way, the project

proceeds as if all the design decisions were made and written down before the code was written.

The critic should pay particular attention to any specified side-effect within a routine. The description of the routine should both fully explain the intended effect and the necessity for using a side-effect rather than communicating via the parameter interface. I am not necessarily suggesting that side-effects not be used (input/output routines, random number generators, symbol table handers and so on perforce must utilize some side-effects), but they do increase program complexity quite markedly, and are so easily forgotten when later looking at the routine invocation.

Phase 3 The author supplies the critic with listings of the individual modules. These must be syntax-error free compiler or assembler listings, at the highest available diagnostic level of the language processor, with symbol table and cross-reference options (hopefully available). The critic's job at this stage is not to follow the flow of control, but to simply check each type, data or routine identifier for appropriate choice of name, correct definition, and consistency of use (thus the need for cross-references).

If the mnemonics used by the author for names do not connote the right meanings to the critic, they must be changed. The overall intelligibility of the program to later readers (particularly maintainers) depends heavily on these choices. I have found that programmers who are not aware of the reading problem will tend to use shortened abbreviations, which carry a sentence-full of information to him, but nothing to the reader, e.g. NIITP = "next input item to process". This imposes a double load on the reader, to learn firstly the identifier, and secondly its meaning. Natural language words impose much less load. In the above instance, 'NEXT' is the logical choice, but if this is required for something else, using words like 'CAT' for the current item and 'DOG' for the one following it are more easily learned than the unfamiliar acronym. (Note to language designers--English averages about one bit of information per character, so adding one character of allowable length to identifiers approximately doubles the number of useful words--including a break character together with a generous limit on length allows descriptive phrases to be used.)

It may seem that this phase comes a little late in the sequence. Surely, you might ask, checking identifiers after the code has been prepared and compiled is like the horse and the stable door. In practice, once the author is aware that he is writing to be read, and knows what will give his critic the least difficulty, he will rarely have to make a change to satisfy the

critic. In cases of doubt, the author can consult with the critic beforehand. This part of this phase should be more a formality than the start of a heavy rewrite (or at least some use of the editor to perform systematic changes). The amount of space devoted to the issue is just to emphasize the importance.

The main functions of the phase are the "correctness of definition" and "consistency of use" checks, firstly, to pick up the obvious errors, and secondly, to familiarize the critic with the basic data structures of the program, the values they assume, and the operations performed upon them.

Phase 4 Now the critic actually starts to read the code. It is my earnest recommendation that this be done bottom-up, that is, commencing with the lowest level routines. The critic should ensure that each routine does exactly as its interface description says that it does, nothing more and nothing less. Once 100% confidence is established in the description, it may be referred to whenever an invocation of the routine is encountered, with there being no further need to reread the code of the routine.

At this point, the critic should find the reading fairly easy work--he is familiar with the objectives of the program, has a good idea of what is done in each routine, knows what data the program manipulates and the kind of operations it applies to each datum. The critic should be able to concentrate on the flow of control with relatively little distraction.

The critic needs to be aided during this phase by suitable running commentary within the program text. He should note all places where he feels he had to work too hard to see the purpose or workings of a statement, or where the commentary was misleading. The benefit of the doubt belongs with the critic--he represents all later readers. The critic should also feel free to ask for the removal of irrelevant commentary, although in this case the benefit of any doubt belongs to the author.

The reason for starting at the bottom is to counter the human tendency for seeing what one wishes to see. If an invocation of a routine is seen before the routine has been checked against its description, there is some pressure to accept the description at face value if it accords with the way the routine is being used, or worse, to read the code of the routine and overlook the description. In addition, the routine will be examined with the prejudicial knowledge of what it is meant to do in the particular context of the invocation. The perceptual phenomenon of "completion" (the tendency to see things the way they should be, rather than the way they are) will interfere with accurate reading.

The outcome of this phase could be a return to any of the earlier phases, either to update descriptions, redesign data structures, restructure the modules, pick new names, or simply to fix errors.

While the critic is reading the text, which could take some time, the author (or other responsible party) can be debugging. But don't tell the critic the outcome! The critic's reading and the debugging crosscheck each other. Ideally, the critic will find all the errors that the debugging does, and vice versa. If not, someone is not trying hard enough. Of course, as in all things, perfection is not possible, but significant deviation in diagnostic performance between the reading and the debugging indicates problems in the project. (The really aggressive author can "bebug" the program to measure the critic's performance.)

Phase 5 The critic is given the external documentation to read (user manual, maintenance manual, etc). This is basically a check for completeness, as the critic is in a good position to see that all features are described, all limitations are mentioned, and that the documentation accords with the critic's understanding of the program.

There are a number of other people who should also read the external documentation before it reaches its "final" form. The author, of course, irrespective of whether he originated the document; the designer who wrote the program specifications, and who probably has some responsibility to see that the documentation meshes with the documentation of other programs associated with this one; a representative of the user community (at least one), to ensure the manuals speak a language that they understand. But the critic is the only person in a position to give an informed, interested, though independent opinion.

-o-o-O-o-o-

Throughout, the critic's role is to interact with the author and that which the author writes. Care has been taken to see that the critic does not become involved in the author's interaction with other people, such as the author's supervisor, the testing department (if this is a separate function), the documentation editor etc. Whether you are a chief programmer's co-pilot, a member of a co-operative team, or just an office-mate unofficially helping a friend, I do believe that the above schema should be followed at least in essence, if your role as critic is to be productive. To people interacting with the author, the critic should be invisible, apart from the observable readability of the author's code.

The critic is no replacement for other methods of quality control, either of the software or the associated documentation. We can hope that the quality will be higher, or the production time be shorter.

The above five-phase scheme is derived from experience with simply reading programs. Inevitably, when picking up a program to read, the first thing sought is a description of what the program does, its inputs, options, by-products and outputs. The next thing sought is some idea of how it does it, where decisions are taken, where data transformations are made, the way these are distributed through the program. In a program written to be read, these steps are straightforward. If not, this information must be painfully deduced from evidence found in later stages of the reading.

The next natural step is to start reading the code, but this will only be effective if the function of the data structures is known. Without definitions of these functions, successive guesses must be made till finally the code is understood. Thus the emphasis placed on phase 3 of the critic's task.

Phase 5 is placed where it is simply because that is the time the critic can most usefully perform that task. If you are a later reader, you will undoubtedly read the user documentation first (if there is any) as a way of obtaining a definition of the function of the program.

Either as a critic or a maintainer, or even as a casual reader, the benefit to be derived from reading programs is to see how someone else makes use of a language or machine, thus broadening your experience and outlook at nowhere near the cost of writing the program yourself. And you will gain a very deep appreciation of the difficulties that you can impose on a reader when you yourself are the author. If you bring this appreciation to your own writing, you too can write readable programs.

Short Quiz. Can you remember the four identifiers used in the example in the discussion of phase 3? If three come easily, then the point is made.

This I did next

"'The time has come,' the Walrus said,
'To talk of many things:"
Lewis Carrol,
in "The Walrus and the
Carpenter."

"'Oh no, no, you are not seeing it. Your kind of visualising is not right for seeing this. Think of it abstractly. What is happening on this photograph of an explosion is that the first differential coefficient vanishes identically, and that is why what becomes visible is the trace of the second differential coefficient.'"
John von Neumann, as quoted by
Jacob Bronowski,
in The Ascent of Man.*

The program had been written, rewritten, punched, assembled, punching errors eliminated, loaded and now the testing was completed. The testing was done by the somewhat old-fashioned method of dumping the memory and single-stepping through the program. The errors found had been corrected, and the program re-tested. The last pass through the troublesome program paths revealed no masked errors. So what to do now? For about half-an-hour I sat at the machine console, idly re-reading the program text, really at a loss as to what to do next. Life appeared to be empty of further challenge.

After a period, I came to thinking of what there is in programming, and particularly this program, that creates such a state, such an internal tension, that the aftermath has all the symptoms of depression. Other writers have made the observation that amateur and beginning programmers exhibit many of the signs of addiction. Perhaps the difference between these and the professional programmer is that the latter has learnt to control the exhibition of the signs, being reasonably assured of a continuing source of his "drug".

This analogy is, of course, much too facile. I feel that the "high" of the programmer is much more akin to that of the creative artist, involving the same elements of personal commitment, ego projection, stylistic expression and dedication to perfection.

---

*Little, Brown and Company, Boston (1973).

The absolute marriage of the intellect, the activity of the body, and the creative urge in programming brings about the conditions in which self-limitations are surmounted (however imperfectly) to achieve a work that we can, and do, judge by aesthetic standards.

The apparent "post-program" depression is really the relaxation experienced by the high-jumper descending from the bar into the padding. The mind has been focussed, the energies rallied, and gravity overcome. Now is the time to relax, before having to attempt to do better.

The program was written to fulfill a definite need. For the last five years, one of the important tasks for the Librascope computer operated by the Electrical Engineering Department has been the production of typewritten documents on the Selectric typewriter attached to it. The mode of operation is that a "type tape" is prepared, the contents of which are transmitted to the typewriter at 15 characters/sec by a small interrupt-driven program running in background, occupying the high memory locations. A set of switches on a separate control panel is used to select such options as stopping at the end of the page, pausing temporarily (e.g. while the ribbon of the typewriter is changed), and so on.

In the course of time, a number of desirable (but absent) features had been identified. The implementation of the most necessary of these was originally envisioned as a modification of the existing program, but after examination, it quickly became apparent that the existing program was the "one to throw away". With the constraints imposed by the structure of the existing program removed, it became possible to consider implementing all desired features.

Apart from limitations on design-programming-testing-documentation time (due to other commitments, personal effort was restricted to about an hour a day for two or three months), the design had to conform to two constraints: the time from interrupt to delivery of the character to the typewriter should not, on the average, exceed 500 microseconds; and the size of the program ideally should not exceed 400 words (the space occupied by the existing program), or at worst not exceed that figure by more than 20%.

The main disability with which I came to this project was never having programmed the machine in assembly language before, and thus was neither familiar with the details of the machine nor the assembly language. Having the existing program was a help, but as it was not documented and only sparsely commented, to understand it would require a fairly heavy investment in becoming acquainted with the machine and the assembly language anyway. In addition,

apart from the sketches of the logical connections, the hardware added to the machine to interface the typewriter was not documented.

The first step then was to read the machine manual and the assembly language manual. At the same time, I read as much of the MONITOR as I could, both to test my understanding of what I had read, and to see how the instruction set was put to use. The side benefit was investigating the connections that a background program must necessarily make with the MONITOR to obtain interrupt service.

The program structure to be used, in outline, was obvious: a mainline to handle the majority of characters, branching to processors for special actions, a level or so of subroutines underneath to handle tape input operations and other management details. Along the mainline, the time constraint would be dominant, all else would be governed by the space constraint. The execution of paths other than the mainline would occur one or more orders of magnitude less often, so would have small impact on the average speed.

At this point, many details were left to be resolved, but it was clear what the tape input routines had to do. So I decided to start immediately with a bottom-up implementation, devoting most of my hour/day to programming, and relegated further "thinking" to background while eating, sleeping etc. So I proceeded with this, defining macros to handle things like subroutine entry/exit, i/o channel selection, and spins (two-instruction loops waiting for the i/o channel to cease to be busy). These routines were ready for assembly after two weeks.

The assembler would not accept the macro definitions, although they conformed to the description in the manual. The MONITOR code used no macros, so no guidance was available there. The existing program used just one. The format of its definition was different to what I had used, and the code of the assembler was sufficiently obscure that I could not deduce the format from there. So I rewrote the macro definitions, imitating as best I could the one example that I had. The assembler now complained that I had too many macros. The manual said I could have 1000. So I removed definitions till the assembler stopped complaining. The number left was zero. Eventually I discovered that there was a later edition of the assembler manual (the one I had been using being written before the assembler was available), where I learnt that the format I was now using was correct. I finally found that the assembler reads predefined macros from the system library, and apparently there are 999 definitions there, and these leave just enough room to define one more macro, if the definition does not exceed one line. No documentation exists on how to change this

limit, so I abandonned macros as a technique, and wrote all the instructions myself.

At this point, the code assembled, although trouble was looming. This code was about 140 words long, compared to the 60 or so in the existing program that did essentially the same, although, to be fair, I had included error recovery code which was not present in the other.

I went ahead and tested the code, to see whether the correct paths were taken on detecting errors and endfile. All went well, with no logical errors being discovered. I pronounced this part finished, with the mental reservation that I may need to compact the code (not that I could see at this time where I could do so).

In the meanwhile, I had been jotting down algorithms, and also test sequences of code to get a feeling for the suitability of the algorithms for the machine, for handling the operations determined by control characters coming from the tape, and had got to the point where all the behavioural conditions were satisfied. So I finalized the code for these processors, punched and assembled them. Another 160 words of code! With allocation of space for the input record and other data, the total size would be about 330 words. In the existing program, the translation table (from the tape character code to the typewriter code) occupied 64 words, so I was looking at approximately 390 words, without having looked at the mainline code, the control switch decoding and action processors for them, or the interrupt handling. In the existing program these things took around 190 words, handling only 5 control switches, whereas the specification for the new program defined 15 control switches. My program looked as though it was going to be 600-700 words long.

I began to think very much about the efficiency of my use of the instruction set and data space. The most obvious place to begin was with the translation table. The existing program's method of indexed look-up and indirect jumps, requiring one word per entry, obviously used space poorly. I could see that the same information could be reasonably easily packed into sixteen words, imposing an unpacking cost (in time and space) in the mainline. Part of the information required in the translation table was which characters were control characters requiring additional processing. A side-effect of the packing was having to decode this after extracting the entry (the original simply holding the address of the control processors in the table). This led to the observation that once I knew what the special processor was, I also knew what the associated translation was, hence I did not need to have both in the table. This meant I could put either the translated character in the table, or the index of the control processor, plus a bit to say which was which. The translation

table could now fit in ten words, at the expense of some additional instructions to do the unpacking and decoding (which turned out to be 27 words).

The problem now was to generate the translation table, as its structure was now quite complex. The best way to handle that, I decided, was to design a simple, unpacked representation, and to convert the simple to the complex by code in the initialization phase. The initialization code need only be present during the loading of the program, so would not cost against the space limitation. (The code to do this transformation ended up occupying 120 words.)

This saving of space was very gratifying, but represented only a small dent in the 200-300 excess that I appeared to be generating. But still, now that the translation table format was fixed, and the input format was already fixed by the necessity of remaining compatible with programs already generating type tapes, I could write the mainline code. The requirements for this code were so rigid that there was little problem and little scope for doing anything other than what had to be done. With one exception (described later) this code has not varied from the first manuscript.

While worrying all this out, I had been rereading the machine manual, looking for features that I could utilize to compact the code already written. The first major breakthrough was the discovery that an index register could be dedicated to implementing a stack, the address held by the index register being incremented and decremented automatically as words were stored or fetched. This was attractive for aesthetic reasons, as my previous experience with a stack-oriented machine led me naturally to the idea of stacking subroutine return addresses. I could allocate as many words to hold return addresses as the greatest depth of subroutine calling, instead of one per subroutine. Then I could play the game of direct jumps from subroutine to subroutine where the last action in the subroutine was a call on another.

As a result, I rewrote all the subroutine calls, entries and exits to use this new concept (wishing all along that I had been able to make them macros as I had originally intended), making the direct jump optimization where I could. The saving was about 10 words. This was good, so I started to reorganize the tape input routines so that I could use the optimization more often. Regularities started to appear that were previously hidden, so now I could make more common routines. Also, multiple entry points were possible, as the location of the return address was not related to the entry point, so I could combine routines that had the same final sequence, instead of just those that had the same

initial sequence. The final result was that the tape input routines occupied 74 words, about 50% of the initial attempt.

Consider the following example: let SAVE be the instruction that puts a return address on the stack, and RETURN be the instruction that takes it off the stack, and exits the subroutine (actually going to the second instruction beyond the SAVE instruction). The following comes from the read (a tape record) routine:

```
                IF (i/o_error) JUMP (to error_label)
(ok_label)      SAVE
                JUMP (to clear_interface routine)
                RETURN (from read routine)
(error_label)   SAVE
                JUMP (to clear_interface routine)
                (continue with error recovery)
```

The SAVE-JUMP-RETURN sequence can be replaced by simply JUMP, relying on the clear_interface routine to do the return, giving:

```
                IF (i/o_error) JUMP (to error_label)
(ok_label)      JUMP (to clear_interface routine)
(error_label)   SAVE
                JUMP (to clear_interface routine)
                (continue)
```

This can be further compressed to give:

```
                IF NOT (i/o_error) JUMP (to ok_label)
(error_label)   SAVE
(ok_label)      JUMP (to clear_interface routine)
                (continue with error recovery)
```

for a total saving of 3 (out of 5) instructions. (As it represented an important assertion, I wished to retain ok_label, otherwise the JUMP to it could have gone directly to the clear_interface routine.) The intent is now no longer quite as clear, but, as Knuth points out, there is no problem if you know how it was derived.

These transformations of the code had been fairly mechanical, and I was satisfied that none of the original testing had been invalidated, so I did not retest. In the interim, the mainline had been tested, revealing no errors.

I now turned my attention to the size of the control character processors. Although I had written and punched them, I had not tested, feeling that I would need to rework them. On returning to them, it was obvious that I had been using state flags very

heavily, the main purpose being to ensure that after taking action and returning to a common point to await the interrupt response, control could be returned to the control processor to finalize the action before continuing. This structure looked very nice and regular, repeated each time for each processor, and enabled nice assertions to be made about the state based on the state flags--but so expensive! I decided to heed my own advice from the days when I was writing operating systems--do not think of an interrupt-driven program as a subroutine called by the interrupt, but make it a cyclic process and let it call the interrupt, i.e. bury the wait for the interrupt in a subroutine.

This I did, and by doing so, I was able to gain some other marginal benefits (which I shall not detail here). But the principal payoff was being able to let the return address kept by the wait for interrupt routine represent the state. I could now start in earnest looking for ways to shorten the code. The main tool was looking for assertions, which held at labels, being true elsewhere. Since the state was represented by an address, and an address represented a place in the code, the structuring model was that of a set of finite state machines. The result: 92 words, down from 160, at the "cost" of having 30 labels (each representing a state), most of which have at least two paths into them. The desk checking consisted of ensuring that the required assertions held at each jump to a label, and that given the correct conditions, the routines were exited with the job done.

Some of the code to test the control switches was distributed through the control character routines (there being defined interactions), so I added the remainder of the code for the control switches, and went now directly to a "system test" (feeling that there could not be much wrong in any of this code).

The immediate result was that the machine "hung" completely. stopping somewhere in the MONITOR. A dump indicated that the initialization had completed correctly. A half-a-day of stepping through the MONITOR showed that it contained an error (at least six years old) that was precipitated by having the flip-flop set that enabled the automatic stacking mode. This had been left on at the end of the initialization (as I had assumed it would be cleared in the program termination sequence of the MONITOR). Clearing this flip-flop before returning to the MONITOR made this problem go away (but there could still be problems if the initialization is interrupted by the operator--but fixing the MONITOR is beyond my responsibilities).

On the next trial, I started getting characters typed on the typewriter, but all were spaces except for sixteen characters every tape record. A dump showed that this was exactly what was in the input buffer, so there had to be a fault in reading from

the tape. Immediately, I suspected that I had done something wrong while changing the subroutine calling sequences and the subsequent optimization of these routines. Eventually, I found that the instruction that was loading the i/o channel with the number of words to be read was setting the counter to read two words (sixteen characters) instead of ten words. This instruction performed a register-to-register copy, and was itself correct. I replaced it with a memory-to-register transfer instruction, and lo, the counter was being set correctly. Why, I still do not know.

The next test was to check correct reaction to the control switches. Result: no reaction at all. I wrote a small test program to report just the switch settings. According to the machine manual, the switches set bits in a register. This register can be transferred to memory by a register-to-memory instruction, or the bits can be treated as individual flip-flops by a state testing instruction. Apparently the machine was installed sans the hardware to implement the individual bit accessing mode. I now had to rewrite the parts of the program that tested control switch bits directly, and substitute code that tested the bits in a memory copy of the switch register. This rewrite impacted the space constraint, as the state testing instructions allowed for testing whether a switch were on or off. Using the memory copy, I must first load it to the A-register, and then I could only test whether the corresponding bit were set. Que sera, sera.

I could now return to the testing of switch reactions. I found I could still not get a reaction to one particular switch, but now I had my suspicions in the right place. I opened the panel holding the switches, and immediately located the disconnected wire. Reconnecting this wire fixed the problem. Continuing, I found a reaction that indicated that one switch would not turn off. This time, a dump showed that it was indeed switching off, so I single-stepped through the corresponding control routine, and found the first genuine programming error--a test for zero jumped the wrong way. A patch demonstrated that this was the only error in the routine.

While all this was going on, I had discovered that one of the heavier users of the typewriter prepared tapes on the CDC6400 at the Computer Center, and then did a code translation on the Librascope to get type tapes acceptable to the typing program, a procedure prone to error (for reasons related to the reliability of the machine) and delay (as only one person knew the details of running the conversion program). I began to consider the feasibility of adding a second translation table to the typing program that I had written, to make the CDC tapes acceptable without prior translation. The earlier decisions had decoupled

the program from the details of the table, so it was merely a question of setting up another table (easily done, as I had the necesssary code in the initialization phase) and selecting it (again easily done, as the "foriegn" tapes had a distinctive format). It took a couple of days (two hours of actual time) to make these additions, costing a total of 16 words (11 of which were for the second translation table and its access pointer), bringing the total to 461 words.

Being very confident at this point of the correctness of the program, I mounted a trial run using a CDC prepared tape. Result: garbage typed on the typewriter, eventually a hung machine, and no ideas. So I started single-stepping through everything, the setting up of the tables in the initialization phase, the interrupt response, reading of the tape record, unpacking of the input characters, translation table access, decode of the table entry, decision that this was not a control character--hold it! The character that I was looking at should be a control character. Now, the index of the second translation table was being set up correctly in an index register, the translation table was being accessed correctly to get, in this case, the index of the control character processor, the translation table was being again accessed for the bit indicating this to be a control processor index and not a typewriter character--but the bit was not set. The initialization phase was setting it, and a dump now showed it set. Then I noticed that I had the bit pattern from the corresponding position in the other translation table, so somehow the index register had become set to zero, instead of being still one (it had to have been one to get the control processor index properly). There were just two instructions between the first and second access to the translation table, a test, and a shift--then light dawned. The second table accessing instruction did not specify any indexing, hence was getting a zero value by default (which didn't matter given only one table).

I made the fix, and then all went well. At this point I was prepared to release the program for use. The only doubts in my mind were concerning the accuracy of the second translation table, as I had had to deduce this from the data cards to the conversion program, and what I could actually see on the sample tape from the Computer Center. While pondering this, I saw that I could save another word in the loop that selected the translation table on the basis of the format of the tape, which would bring the size to a nice "round" 460. So I made that change. Also, I saw that one of the control processors for special characters had two exit points, both requiring the re-evaluation of the switch settings, but that they went to different points. I felt that it would be clearer to the reader if they both went to the same point. Making this change made the code look far more symmetric.

Having done this, I mounted a demonstration. Disaster, as not only did the program behave erratically, the material typed looked as though the wrong translation table was being used. Looking at the code, I saw firstly, that the change I had made in the selection loop to save a word had violated the loop invariant (which was that the pattern being tested was in the A-register). By changing the invariant, and suitably altering the rest of the loop, I was still able to save the word, and again, on the desk, demonstrate the correctness. Then I looked at the other change, and again saw that a label assertion was being violated. The common point to which both exits went had the assertion that the switch register copy was in the A-register. The exit that I had changed satisfied a "don't care" with respect to the A-register. The program was taking the random stuff in the A-register as control switch settings, and reacting accordingly (i.e. randomly). This explained why the two exits had been different. Returning to the original arrangement fixed this problem. (So much for roundness of numbers or symmetry of code as criteria by which to program.)

Just to be sure, I single-stepped through the paths that had been troublesome, but this revealed no masked errors. So what to do now? For about half-an-hour I sat at the machine console, idly re-reading the program text, really at a loss as to what to do next. Life appeared to be empty of further challenge.