DESIGN PRINCIPLES

PER BRINCH HANSEN

(1977)

This is the opening chapter of the author's book on concurrent programming. The essay describes the fundamental principles of programming which guided the design and implementation of the programming language Concurrent Pascal and the model operating systems written in that language.

This book describes a method for writing concurrent programs of high quality. Since there is no common agreement among programmers about the qualities a good program should have, I will begin by describing my own requirements.

Program Quality

A good program must be *simple*, *reliable*, and *adaptable*. Without simplicity one cannot expect to understand the purpose and details of a large program. Without reliability one cannot seriously depend on it. And without adaptability to changing requirements a program eventually becomes a fossil.

Fortunately, these essential requirements go hand in hand. Simplicity gives one the confidence to believe that a program works and makes it clear how it can be changed. Simplicity, reliability, and adaptability make programs *manageable*.

In addition, it is desirable to make programs that can work efficiently on several different computers for a variety of similar applications. But *efficiency*, *portability*, and *generality* should never be sought at the expense of simplicity, reliability, and adaptability, for only the latter qualities make

P. Brinch Hansen, *The Architecture of Concurrent Programs*, Chapter 1 Design Principles, Prentice Hall, Englewood Cliffs, NJ, (July 1977), 3–14.

it possible to understand what programs do, depend on them, and extend their capabilities.

The poor quality of much existing software is, to a large extent, the result of turning these priorities upside down. Some programmers justify extremely complex and incomprehensible programs by their high efficiency. Others claim that the poor reliability and efficiency of their huge programs are outweighed by their broad scope of application.

Personally I find the efficiency of a tool that nobody fully understands irrelevant. And I find it difficult to appreciate a general-purpose tool which is so slow that it cannot do anything well. But these are matters of taste and style and are likely to remain so.

Whenever program qualities appear to be in conflict with one another I shall consistently settle the issue by giving first priority to manageability, second priority to efficiency, and third priority to generality. This boils down to the simple rule of limiting our computer applications to those which programmers fully understand and which machines can handle well. Although this is too narrow a view for experimental computer usage it is sound advice for professional programming.

Let us now look more closely at these program qualities to see how they can be achieved.

Simplicity

We will be writing concurrent programs which are so large that one cannot understand them all at once. So we must reason about them in smaller *pieces*. What properties should these pieces have? Well, they should be so small that any one of them is trivial to understand in itself. It would be ideal if they were no more than *one page* of text each so that they can be comprehended at a glance.

Such a program could be studied page by page as one reads a book. But in the end, when we have understood what all the pieces do, we must still be able to see what their combined effect *as a whole* is. If it is a program of many pages we can only do this by ignoring most of our detailed knowledge about the pieces and relying on a much simpler description of what they do and how they work together.

So our program pieces must allow us to make a clear separation of their detailed behavior and that small part of it which is of interest when we consider combinations of such pieces. In other words, we must distinguish between the *inner and outer behavior* of a program piece.

Program pieces will be built to perform well-defined, simple functions. We will then combine program pieces into larger *configurations* to carry out more complicated functions. This design method is effective because it splits a complicated task into simpler ones: First you convince yourself that the pieces work individually, and then you think about how they work together. During the second part of the argument it is essential to be able to forget how a piece works in detail—otherwise, the problem becomes too complicated. But in doing so one makes the fundamental assumption that the piece always will do the same when it carries out its function. Otherwise, you could not afford to ignore the detailed behavior of that piece in your reasoning about the whole system.

So *reproducible behavior* is a vital property of program pieces that we wish to build and study in small steps. We must clearly keep this in mind when we select the kind of program pieces that large concurrent programs will be made of. The ability to repeat program behavior is taken for granted when we write sequential programs. Here the sequence of events is completely defined by the program and its input data. But in a concurrent program simultaneous events take place at rates not fully controlled by the programmer. They depend on the presence of other jobs in the machine and the scheduling policy used to execute them. This means that a conscious effort must be made to design concurrent programs with reproducible behavior.

The idea of reasoning first about *what* a piece does and then studying *how* it does it in detail is most effective if we can repeat this process by explaining each piece in terms of simpler pieces which themselves are built from still simpler pieces. So we shall confine ourselves to *hierarchical structures* composed of *layers* of program pieces.

It will certainly simplify our understanding of hierarchical structures if each part only depends on a small number of other parts. We will therefore try to build structures that have *minimal interfaces* between their parts.

This is extremely difficult to do in *machine language* since the slightest programming mistake can make an instruction destroy any instruction or variable. Here the *whole store* can be the interface between any two instructions. This was made only too clear in the past by the practice of printing the contents of the entire store just to locate a single programming error.

Programs written in *abstract languages* (such as Fortran, Algol, and Pascal) are unable to modify themselves. But they can still have broad interfaces in the form of *global variables* that can be changed by every statement (by intention or mistake).

We will use a programming language called *Concurrent Pascal*, which makes it possible to divide the global variables into smaller parts. Each of these is accessible to a small number of statements only.

The main contribution of a good programming language to simplicity is to provide an abstract *readable notation* that makes the parts and structure of a program obvious to a reader. An abstract programming language *suppresses machine detail* (such as addresses, registers, bit patterns, interrupts, and sometimes even the number of processors available). Instead the language relies on *abstract concepts* (such as variables, data types, synchronizing operations, and concurrent processes). As a result, program texts written in abstract languages are often an order of magnitude shorter than those written in machine language. This *textual reduction* simplifies program engineering considerably.

The fastest way to discover whether or not you have invented a simple program structure is to try to *describe* it in completely readable terms—adopting the same standards of clarity that are required of a survey paper published by a journal. If you take pride in your description you have probably invented a good program structure. But if you discover that there is no simple way of describing what you intend to do, then you should probably look for some other way of doing it.

Once you appreciate the value of description as an early warning signal of unnecessary complexity it becomes self-evident that program structures should be described (without detail) *before* they are built and should be described by the *designer* (and not by anybody else). *Programming is the art of writing essays in crystal clear prose and making them executable.*

Reliability

Even the most readable language notation cannot prevent programmers from making mistakes. In looking for these in large programs we need all the help we can get. A whole range of techniques is available

correctness proofs
proofreading
compilation checks
execution checks
systematic testing

With the exception of correctness proofs, all these techniques played a vital role in making the concurrent programs described in this book work.

Formal proofs are still at an experimental stage, particularly for concurrent programs. Since my aim is to describe techniques that are immediately useful in professional software development, I have omitted proofs here.

Among the useful verification techniques, I feel that those that reveal errors at the earliest possible time during the program development should be emphasized to achieve reliability as soon as possible.

One of the primary goals of Concurrent Pascal is to push the role of *compilation checks* to the limit and reduce the use of *execution checks* as much as possible. This is not done just to make compiled programs more efficient by reducing the overhead of execution checks. In program engineering, compilation and execution checks play the same roles as preventive maintenance and flight recorders do in aviation. The latter only tell you why a system crashed; the former prevents it. This distinction seems essential to me in the design of real-time systems that will control vital functions in society. Such systems must be highly reliable *before* they are put into operation.

Extensive compilation checks are possible only if the language notation is *redundant*. The programmer must be able to specify important properties in at least two different ways so that a compiler can look for possible inconsistencies. An example is the use of declarations to introduce variables and their types before they are used in statements. The compiler could easily derive this information from the statements—provided these statements were always correct.

We shall also follow the crucial principle of language design suggested by Hoare: *The behavior of a program written in an abstract language should always be explainable in terms of the concepts of that language and should never require insight into the details of compilers and computers.* Otherwise, an abstract notation has no significant value in reducing complexity.

This principle immediately rules out the use of machine-oriented features in programming languages. So I shall assume that *all programming will take place in abstract programming languages.*

Dijkstra has remarked that *testing* can be used only to show the presence of errors but never their absence. However true that may be, it seems very worthwhile to me to show the presence of errors and remove them one at a time. In my experience, the combination of careful proofreading, extensive compilation checks, and systematic testing is a very effective way to make a program so dependable that it can work for months without problems. And that is about as reliable as most other technology we depend on. I do not know of better methods for verifying large programs at the moment.

I view programming as the art of building *program pyramids* by adding one brick at a time to the structure and making sure that it does not collapse in the process. The pyramid must remain *stable* while it is being built. I will regard a (possibly incomplete) program as being stable as long as it behaves in a predictable manner.

Why is program testing so often difficult? Mainly, I think, because the addition of a new program piece can spread a burst of errors throughout the rest of a program and make previously tested pieces behave differently. This clearly violates the sound principle of being able to assume that when you have built and tested a part of a large program it will continue to behave correctly *under all circumstances.*

So we will make the strong requirement that *new program pieces added on top of old ones must not be able to make the latter fail.* Since this property must be verified before program testing takes place, it must be done by a compiler. We must therefore use a language notation that makes it clear what program pieces can do to one another. This strong *confinement of program errors* to the part in which they occur will make it much easier to determine from the behavior of a large program where its errors are.

Adaptability

A large program is so expensive to develop that it must be used for several years to make the effort worthwhile. As time passes the users' needs change, and it becomes necessary to modify the program somewhat to satisfy them. Quite often these modifications are done by people who did not develop the program in the first place. Their main difficulty is to find out how the program works and whether it will still work after being changed.

A small group of people can often succeed in developing the first version of a program in a low-level language with little or no documentation to support them. They do it by talking to one another daily and by sharing a mental picture of a simple structure.

But later, when the same program must be extended by other programmers who are not in frequent contact with the original designers, it becomes painfully clear that the "simple" structure is not described anywhere and certainly is not revealed by the primitive language notation used. It is important to realize that *for program maintenance a simple and well-documented structure is even more important than it is during program development.* I will not talk about the situation in which a program that is neither simple nor well documented must be changed.

There is an interesting relationship between programming errors and changing user requirements. Both of them are sources of *instability* in the program construction process that make it difficult to reach a state in which you have complete confidence in what a program does. They are caused by our inability to fully comprehend at once what a large program is supposed to do in detail.

The relative frequencies of program errors and changing requirements are of crucial importance. If programming introduces numerous errors that are difficult to locate, many of them may still be in the program when the user requests changes of its function. And when an engineer constantly finds himself changing a system that he never succeeded in making work correctly in the first place, he will eventually end up with a very unstable product.

On the other hand, if program errors can be located and corrected at a much faster rate than the system develops, then the addition of a new piece (or a change) to the program will soon lead to a stable situation in which the current version of the program works reliably and predictably. The engineer can then, with much greater confidence, adapt his product to slowly changing needs. This is a strong incentive to make program verification and testing fast.

A hierarchical structure consists of program pieces that can be studied one at a time. This makes it easier to read the program and get an initial understanding of what it does and how it does it. Once you have that insight, the consequences of changing a hierarchical program become clear. When you change a part of a program pyramid you must be prepared to inspect and perhaps change the program parts that are on top of it (for they are the only ones that can possibly depend on the one you changed).

Portability

The ability to use the same program on a variety of computers is desirable for economic reasons: Many users have different computers; sometimes they replace them with new ones; and quite often they have a common interest in sharing programs developed on different machines.

Portability is only practical if programs are written in abstract languages that hide the differences between computers as much as possible. Otherwise, it will require extensive rewriting and testing to move programs from one machine to another. Programs written in the same language can be made portable in several ways:

1. by having *different compilers* for different machines. This is only practical for the most widespread languages.

2. by having a *single compiler* that can be modified to generate code for different machines. This requires a clear separation within the compiler of those parts that check programs and those that generate code.

3. by having a *single computer* that can be simulated efficiently on different machines.

The Concurrent Pascal compiler generates code for a simple machine tailored to the language. This machine is simulated by an assembly language program of 4 K words on the PDP 11/45 computer. To move the language to another computer one rewrites this interpreter. This approach sacrifices some efficiency to make portability possible. The loss of efficiency can be eliminated on a microprogrammable machine.

Efficiency

Efficient programs save time for people waiting for results and reduce the cost of computation. The programs described here owe their efficiency to

special-purpose algorithms
static store allocation
minimal run-time checking

Initially the loading of a large program (such as a compiler) from disk took about 16 sec on the PDP 11/45 computer. This was later reduced to 5 sec by a disk allocation algorithm that depends on the special characteristics of program files (as opposed to data files). A scheduling algorithm that tries to reduce disk head movement in general would have been useless here. The reasons for this will be made clear later.

Dynamic store algorithms that move programs and data segments around during execution can be a serious source of inefficiency that is not under the programmer's control. The implementation of Concurrent Pascal does not require garbage collection or demand paging of storage. It uses static allocation of store among a fixed number of processes. The store requirements are determined by the compiler.

When programs are written in assembly language it is impossible to predict what they will do. Most computers depend on hardware mechanisms to prevent such programs from destroying one another or the operating system.

In Concurrent Pascal most of this protection is guaranteed by the compiler and is not supported by hardware mechanisms during execution. This drastic reduction of run-time checking is only possible because all programs are written in an abstract language.

Generality

To achieve simplicity and reliability we will depend exclusively on a machine-independent language that makes programs readable and extensive compilation checks possible. To achieve efficiency we will use the simplest possible store allocation.

These decisions will no doubt reduce the usefulness of Concurrent Pascal for some applications. But I see no way of avoiding that. To impose *structure* upon yourself is to impose *restrictions* on your freedom of programming. You can no longer use the machine in any way you want (because the language makes it impossible to talk directly about some machine features). You can no longer delay certain program decisions until execution time (because the compiler checks and freezes things much earlier). But the freedom you lose is often illusory anyhow, since it can complicate programming to the point where you are unable to cope with it.

This book describes a range of small operating systems. Each of them provides a special service in the most efficient and simple manner. They show that Concurrent Pascal is a useful programming language for minicomputer operating systems and dedicated real-time applications. I expect that the language will be useful (but not sufficient) for writing large, general-purpose operating systems. But that still remains to be seen. I have tried to make a programming tool that is very convenient for many applications rather than one which is tolerable for all purposes.

Conclusion

I have discussed the programming goals of

simplicity
reliability
adaptability
efficiency
portability

and have suggested that they can be achieved by careful design of program structure, language notation, compiler, and code interpreter. The properties

that we must look for are the following:

structure:	hierarchical structure
	small parts
	minimal interfaces
	reproducible behavior
	readable documentation
notation:	abstract and readable
	structured and redundant
compiler:	reliable and fast
	extensive checking
	portable code
interpreter:	reliable and fast
	minimal checking
	static store allocation

This is the philosophy we will follow in the design of concurrent programs.

Literature

For me the most enjoyable thing about computer programming is the insight it gives into problem solving and design. The search for simplicity and structure is common to all intellectual disciplines.

Here are a historian and a biologist talking about the importance of recognizing structure:

"It is a matter of some importance to link teaching and research, even very detailed research, to an acceptable architectonic vision of the whole. Without such connections, detail becomes mere antiquarianism. Yet while history without detail is inconceivable, without an organizing vision it quickly becomes incomprehensible ... What cannot be understood becomes meaningless, and reasonable men quite properly refuse to pay attention to meaningless matters."

William H. McNeill (1974)

"There have been a number of physicists who suggested that biological phenomena are related to the finest aspects of the constitution of matter, in a manner of speaking below the chemical level. But the evidence, which is almost too abundant, indicates that biological phenomena operate on the 'systems' level, that is, above chemistry."

Walter M. Elsasser (1975)

A linguist, a psychologist, and a logician have this to say about writing and notation:

"Omit needless words. Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subject only in outline, but that every word tell."

William Strunk, Jr. (1959)

"How complex or simple a structure is depends critically upon the way in which we describe it. Most of the complex structures found in the world are enormously redundant, and we can use this redundancy to simplify their description. But to use it, to achieve the simplification, we must find the right representation."

Herbert A. Simon (1969)

"There is something uncanny about the power of a happily chosen ideographic language; for it often allows one to express relations which have no names in natural language and therefore have never been noticed by anyone. Symbolism, then, becomes an organ of discovery rather than mere notation."

Susanne K. Langer (1967)

An engineer and an architect discuss the influence of human errors and cultural changes on the design process:

"First, one must perform perfectly. The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program."

Frederick P. Brooks, Jr. (1975)

"Misfit provides an incentive to change ... However, for the fit to occur in practice, one vital condition must be satisfied. It must have time to happen. The process must be able to achieve its equilibrium before the next culture change upsets it again. It must actually have time to reach its equilibrium

every time it is disturbed—or, if we see the process as continous rather than intermittent, the adjustment of forms must proceed more quickly than the drift of the culture context."

Christopher Alexander (1964)

Finally, here are a mathematician and a physicist writing about the beauty and joy of creative work:

"The mathematician's patterns, like the painter's or the poet's, must be beautiful; the ideas, like the colours or the words, must fit together in a harmonious way. Beauty is the first test: there is no permanent place in the world for ugly mathematics."

G.H. Hardy (1967)

"The most powerful drive in the ascent of man is his pleasure in his own skill. He loves to do what he does well and, having done it well, he loves to do it better. You see it in his science. You see it in the magnificence with which he carves and builds, the loving care, the gaiety, the effrontery. The monuments are supposed to commemorate kings and religions, heroes, dogmas, but in the end the man they commemorate is the builder."

Jacob Bronowski (1973)

References

Alexander, C. 1964. *Notes on the Synthesis of Form.* Harvard University Press, Cambridge, MA.

Bronowski, J. 1973. *The Ascent of Man.* Little, Brown and Company, Boston, MA.

Brooks, F.P. 1975. *The Mythical Man-Month. Essays on Software Engineering.* Addison-Wesley, Reading, MA.

Elsasser, W.M. 1975. *The Chief Abstractions of Biology.* American Elsevier, New York.

Hardy, G.H. 1967. *A Mathematician's Apology.* Cambridge University Press, New York.

Langer, S.K. 1967. *An Introduction to Symbolic Logic.* Dover Publications, New York.

McNeill, W.H. 1974. *The Shape of European History.* Oxford University Press, New York.

Simon, H.A. 1969. *The Sciences of the Artificial.* The MIT Press, Cambridge, MA.

Strunk, W., and White, E.B. 1959. *The Elements of Style.* Macmillan, New York.