P.J. PLAUGER
"PROGRAMMING ON PURPOSE: ESSAYS ON SOFTWARE TECHNOLOGY," PRENTICE HALL, 1994.

WG14/N376
X3J11/94-061

4 The Central Folly

I learned a long time ago how to buy a television set. Sure, you worry about how the picture looks on the demo set. And sure, you check that it has all the gadgets you want. If you're sufficiently fastidious, you might paw through recent issues of *Consumer Reports* to reinforce your current prejudice. And if you care anything about decor, you might even note whether the cabinet clashes with your beer-can collection. But that's not where the real action is.

What I learned to do, after I had convinced myself a given model was a likely candidate, was to turn it around. On the back of nearly every television set ever made is at least one mysterious knob. It may be as modest as a screwdriver hole giving you access to some inner trimpot. It may be as grandiose as a knurled plastic gismo that more or less matches the knobs on the front. But that knob is there all right.

The knob invariably has some arcane label. BUZZ is a nice blend of the familiar and the ominous. FOCUS and VERT LIN are somewhat shopworn, but still dependable entrants. AGC SYNC STABILITY is one of my all-time favorites. Whatever the label says, you can be sure that it's not something that you really want to adjust. You'd be quite content for the television set to give you its best shot at controlling the parameter in question and not solicit your input on the subject.

There is a well known principle in drama that I like to think of as the Pistol Principle. If you, the playwright, cause an actor to call attention to a pistol in the top left-hand drawer of a Louis XIV desk sometime during Act I, then you'd better make sure that pistol gets used before the end of the last act. Otherwise, you are guilty of intellectual clutter.

You can bet that whoever designed the television set you are about to buy was just as sensitive to clutter as the most fastidious playwright. The designer is not competing in the marketplace of ideas, to be sure, but in the much tougher arena of consumer electronics. Those folks count tenths of a cent (or yen, these days) when pricing the cost of parts and assembly, even for a product that will retail in the hundreds of dollars. If there is a knob on the back of your soon-to-be television set, it is there for a reason.

What the knob tells you is that the designer had to compromise. Some part of the circuitry proved to be a little unstable, if not when the set was new then after it had baked in for a few thousand hours. The designer might have added more (or higher quality) components at a critical point, and

139

gone over the 17.3 cent budget for that subassembly. Or the designer might have started over from scratch, to avoid the fundamental problems leading to the instability, and risked delivering the design late (with a different instability). Or the designer could simply bring a knob out to the back, for that fine day when you discover you have to tweak the AGC SYNC STABILITY to watch channel 13.

Guess which is the cheapest alternative.

I was pleasantly surprised to learn that I could evaluate computer programming languages by much the same rules as television sets. No, programming languages don't have knobs on the back. But they do have the moral equivalent thereof.

Every programming language comes with a reference manual, at least, and one or more tutorials, at best. That documentation should tell you all you need to know about the language to put it to use. If it doesn't tell you enough, you're reduced to performing experiments on the current translator you happen to own. Or you switch to another language. If it tells you too much, you are too overwhelmed to get your bearings. Programming languages are among the most complex creations that a single person has to do battle with these days. The last thing you need, when mastering a language, is extraneous detail.

Intellectual clutter in a programming language is just as fatal as over engineered circuitry in mass-market electronic appliances. It can price you out of the market. You can bet that an earnest language documentor tells you only what you need to know. The knobs are kept to a minimum.

So when I evaluate a programming language, I look for the extra knobs sticking out the back. When I see pages and pages of discussion about how to deal with something that I don't care about, I've found the knob. If I can't relate the discussion to the problem I want to solve, then I know I'm being asked to work around a design compromise in the language. Sooner or later, I'm going to have to learn how to tweak that mysterious knob to get the results I want.

The mere presence of a lengthy and arcane discussion in a language tutorial tells me that the designer couldn't eliminate the compromise. The language was in danger of becoming even more complex, or of being late to market due to redesign. The cheapest way out was to try to explain the compromise, rather than eliminate it.

I call this compromise "the central folly." It lies at the heart of the language design, and it is arguably a fundamental mistake. Someone made a conjecture, early on in the design process, and had to stick to it. And later came to regret it.

With that lengthy preamble, I am now ready to introduce my (second) annual April Fool's essay. By long-standing tradition, I take this opportunity to savage other designers, in the thin disguise of good clean fun. My topic this time, as you must have guessed by now, is computer programming languages. And I intend to snipe at three of the biggest ducks in the bay — PL/I, Algol 68, and Ada. Are you ready?

I have this vision of how PL/I came into being. No, I don't really know its exact history, so I won't pretend to anything other than fantasy. Anyway, I imagine a committee forming in the early 1960s, under the benign guidance of IBM. On that committee are numerous FORTRAN and COBOL programmers. All are determined to make a new language worthy of System/360, one that will combine the best points of the most popular scientific programming language and the most popular business programming language. Each is willing to compromise on many broad design issues, provided his or her three favorite features go in as well.

I am convinced that PL/I was designed by a committee of users. I suspect that the only serious arguments they had while piling on all the features they could imagine was whether there was room for three kitchen sinks or four.

PL/I supports every data type you can imagine, and then some. For encoded values, you can choose any combination of:

- binary or decimal base
- fixed-point integers, fixed point with a scale factor, or floating point
- real or complex
- various precisions

You can also specify character strings, with or without an editing picture, and bit strings. In the early days, you could even perform arithmetic in pounds, shillings, and pence! (PL/I dropped Sterling fixed-point constants only after the British empire did.)

It was an interesting conjecture that you needed all of those data types supported directly in the language, if you were going to capture the hearts and minds of all of those FORTRAN and COBOL programmers. But that interacted with another conjecture to cause a few problems.

The other conjecture was that the language must be blindly subsettable. After pouring in features from two distinct cultures, the designers then wanted each culture to be able to use PL/I without learning about the other. Permeating the language design is the attitude, "What you don't know shouldn't hurt you." Every option should have a default. If you fail to specify it, the translator will guess what you probably intended.

One aspect of this conjecture is that keywords are not reserved names in PL/I. (FORTRAN has no reserved names, COBOL has tons of them.) If you write a keyword in a context where it is not expected, the translator will guess that you intend it to be an ordinary name. That attitude permits barbarisms such as:

141

```
    IF IF = THEN
       THEN THEN = ELSE
       ELSE ELSE = IF
```

which ascribes two distinct meanings to each of **IF**, **THEN**, **ELSE**, and the = operator. But what the heck. Any tool can be abused.

Another aspect of this conjecture is that you can write an expression with nearly any combination of data types. Many FORTRAN programmers enjoyed being able to mix integer and floating-point types, and let the translator guess how to combine them sensibly. Why not bring this luxury to the richer world of PL/I data types? The result is that the language explainers had to write pages and pages describing what happens when you combine **REAL FLOAT DECIMAL** operands with bit strings and Sterling fixed-point constants.

There's the knob on the back of the set.

Life is interesting enough with FORTRAN. If you convert a **REAL** to an **INTEGER**, implicitly by assignment, the translator guesses that you want to truncate the result toward zero. Rounding is often a better idea. If you convert an **INTEGER** to a **REAL**, the translator guesses that any low-order bits lost in the process are not worth mentioning. Reporting a loss of significance can sometimes be important. Nevertheless, the number of questionable conversions in FORTRAN is small and easily learned. The conscientious programmer learns when to be careful, or to use the explicit conversion functions.

PL/I, however, offers boundless opportunities for the translator to think up questionable conversions. My favorite eyebrow raisers usually involve some sequence that takes you from a **DECIMAL** form, through a bit string, to a single bit that you want to test. Picking up a meaningless leading zero bit along the way, that you eventually test instead of the good stuff, is frighteningly easy.

Things would not be so bad were programmers educated to write explicit conversions, but such is not the case. Part of the culture of PL/I, as I have seen it practiced, is that real programmers never write anything that doesn't have to be specified. (Imagine taking your favorite large Pascal or C program, erasing all of the conversion functions and/or type casts, and expecting the translator to guess how to put them back.) Getting the expressions right in the first place is hard enough, but maintaining PL/I, to me, often resembles tweaking a knob whose effect I don't understand.

My vision of the origins of Algol 68 also begins in the early 1960s. Again I see a committee, this time composed of numerous language theorists. All share a love for the elegant orthogonality of Algol 60 and a zeal for making a successor that will be even more elegant and even more orthogonal. All have lots of interesting ideas about how to specify a programming language. Here, the major concern is whether the language must have a kitchen sink, per se, or whether you can construct one from underlying primitives.

I understand that the working goal of the committee was to make a language called Algol 64. At least through the end of 1964. The fact that it was eventually called Algol 68 tells us that theoreticians are not immune to schedule overruns either. But what the heck. We're all human, and committees move slowly.

The cute thing about Algol 68 was that the committee ended up inventing a language to describe the language that describes the language. Algol 68 itself has a grammar with an infinite number of productions. (Try writing *that* on a four-sided reference card.) You need a meta-grammar to produce all of the productions that produce all of the valid sentences of the language. Why the committee felt it necessary to retread the English language to describe the meta-grammar, however, is beyond me.

Digging through pages of jargon about "softly deproceduring" and "stirmly hipping to void" is off-putting in the extreme. It is a real barrier to understanding. True, you occasionally unearth a real gem such as, "An assignation is the commonest form of confrontation." But it's not worth sapping through all of the mud along the way. There should probably be a law against quoting Lewis Carroll or W.S. Gilbert in a computer-language reference manual.

Algol 68 lets you declare all sorts of pointer types, a luxury to which we have grown accustomed with Pascal and C. They are called "reference variables" in Algol 68. Accessing a variable via a pointer is called *dereferencing*. Calling a function, given its name or a pointer to it, is called *deproceduring*. All fine and good.

The designers made an interesting conjecture early on, however. They concluded that if you merely mention the name of any variable in an expression, the translator should know what to do with it. If a variable points to another variable, you probably want to dereference it to get the contents of that other variable. You only want to copy it *as a reference variable* if you are assigning it to another reference variable of the same type. Similarly, if a variable names a procedure that has no arguments, then you probably want to deprocedure it, or call it, when you mention it in an expression. And that's always the case. Except when you don't.

Sure, you can also decorate the names with operators to say what you mean. But you don't have to. Instead, the description of Algol 68 contains pages of explanation about how the translator guesses what to do from context (strong, firm, weak, or soft) and from the type of each subexpression. There is even a wonderful railroad-track diagram, filling over half a page, that endeavors to teach you how to second guess the translator.

Have you spotted the knob on the back yet?

To fully appreciate the effect of this conjecture on the description of Algol 68, you have to repeat the gedanken experiment I suggested above. Take your favorite Pascal or C program and erase all of the indirection operators (^ or *) and all of the empty parentheses. Now explain simply how the translator should put them all back.

My view of Ada goes back about a dozen years. I imagine a committee of university consultants forming, under the benign guidance of the U.S. Department of Defense. On that committee are people who fund their research courtesy of the U.S. government. To this attentive audience, the DOD poses a challenge.

We're going to give you some money to study the state of the art of computer programming languages, says the DOD. *We want you to look at what everyone else has done in the way of program design and determine whether:*

- *they have already done a better job than you can possibly do in designing a programming language*
- *we should give you lots more money to spend the next several years designing the programming language you've always dreamed of*

You can guess the result.

No, I'm not going to make snide remarks about gold-plated kitchen sinks, or savage Ada in the usual ways. I believe that the people who worked on Strawman, Ironman, Steelman, and the various color-coded candidate languages had good intentions and did the best jobs they could, under the circumstances.

What I'm saying is that the deck was stacked, by those circumstances, in favor of yet another language designed by committee. Even though Jean Ichbiah gets full and proper credit for bringing considerable coherence to the design of Ada, he was in many ways hobbled by an over detailed specification, produced by a committee.

Ada was specified from the start to be a language in which you can write really large programs. It assumes that a typical program will be constructed from multiple modules written by different people. As a consequence, it worries quite a bit about name-space control. Lots of thought was given to controlling just what names are visible at any given point in an Ada program.

Opposing this concern, however, was an important conjecture — that the types and operators of Ada should be extendible. You should be able to introduce, say, the flavor of complex numbers that you like best, and extend the meaning of all the sensible arithmetic operators to cover them. You can, in other words, *overload* the meaning of the operators plus and minus, for instance, to cover complex operands as well.

Cpa
n
different r
other lang
introduce
notation f

When
space con
to have to
module th
able to ope
unqualifie
interesting
the 23 mod
three of th

What y
and other
how the tr
define mu
seems to b
then it mu
thinks best

There's

To me, t
with it. O
spe

p->

to

val

until the tr
minimum
away. But t
Nor is it th

The poin
a conjec
designer w
too much.
Wirth (Pasc
favor of lir
however, it
future user
on the com

Overloading operators is certainly a convenience. Nearly every language I can think of lets you write a plus operator in ways that have quite different meanings, depending upon the types of its operands. Algol 68 and other languages let you extend the overloading to cover types that you introduce as well. You can write programs that have a very agreeable notation for performing new forms of arithmetic.

When overloading and extendibility meet up with heavy-duty name-space control problems, however, you can expect complexities. It's no fun to have to qualify every plus operator, for instance, with the name of the module that is providing the appropriate definition. Rather, you want to be able to open up a module, as it were, and dump its contents into the general, unqualified name space. That's fine if you have only one module adding interesting new meanings to the plus operator. But what happens if ten of the 23 modules you are using overload plus in different ways? (Or what if three of these overload it the same way, which is different from the others?)

What you can do with operators, you can also do with function names and other creatures. Ada tutorials devote pages and pages to explaining how the translator can guess which meaning to ascribe to a name that you define multiple ways in the same region of program text. The basic rule seems to be, "If the translator has any chance at resolving the ambiguity, then it must permit the ambiguity and endeavor to resolve it the way it thinks best."

There's the knob on the back once again.

To me, this is like taking the **WITH** statement of Pascal and going wild with it. Or perhaps it can be compared to erasing as many structure specifiers from a C program as you can, changing

```
    p->e.o.left->val = x.z;
```

to

```
    val = z;
```

until the translator begs for mercy. It might be interesting to start with the minimum number of qualifiers, then add them until the diagnostics go away. But that probably isn't the most productive use of programmer time. Nor is it the most maintainable code.

The point of all of these gripes is the same. A language designer may have a conjecture about how people plan to use a language. Chances are, the designer will fear that a language will not be used if the user has to specify too much. When the designer is a single, gifted person such as Nicklaus Wirth (Pascal) or Dennis Ritchie (C), he or she can often arbitrarily rule in favor of linguistic simplicity. When the designer is part of a committee, however, it is harder to rule arbitrarily against the putative desires of the future user community. Particularly when there are vocal potential users on the committee.

145

I have characterized the conjectures in each of the three languages in terms of what guesses they require of the translator. It is my belief that high-level language translators have their hands full diagnosing obvious errors and optimizing for less-than-optimal computer architectures. They should not be asked to guess, particularly where a simple word to the wise from a programmer will make their job easier, and the program more readable. You will notice that two of the most successful languages of the past decade, Pascal and C, offer little in the way of shorthand in the areas I have discussed.

I believe that each of the three languages I've taken to task are important languages. I believe that each has many good design features. They have certainly influenced many others, usually to advantage. I have certainly learned many useful principles from studying all three.

Each bears the marks, however, of committee design. Each could be made stronger, I believe, by being asked to do less. And each is hampered by a central folly that causes you to tweak knobs better left hidden. □

Afterword: Rarely does a complex design avoid a central folly. Try your hand at spotting the central folly in UNIX, MS-DOS, C, and C++, just for practice. Learning how to spot such lapses in others can help make you a better designer. The sooner you twig to your own lapses, the better chance you have to mitigate them. At the least, being alert to central follies can make you more tolerant.