"Who Cares About Elegance?" The Role of Aesthetics in Programming Language Design[*]

Bruce J. MacLennan

Computer Science Department
University of Tennessee, Knoxville
maclennan@cs.utk.edu

January 16, 1997

Abstract

The crucial role played by aesthetics in programming language design and the importance of elegance in programming languages are defended on the basis of analogies with structural engineering, as presented in Billington's *The Tower and the Bridge*.

1 The Value of Analogies

Programming language design is a comparatively new activity — it has existed for less than half a century, so it is often worthwhile to look to older design disciplines to understand better this new activity. Thus, my book *Principles of Programming Languages: Design, Evaluation, and Implementation*, grew out of a study of teaching methods in architecture, primarily, but also of pedagogy in other disciplines, such as aircraft design. Perhaps you have also seen analogies drawn between programming languages and cars (FORTRAN = Model T, C = dune buggy, etc.).

These analogies can be very informative, and can serve as "intuition pumps" to enhance our creativity, but they cannot be used uncritically because they are, in the end, just analogies. Ultimately our design decisions must be based on more than analogies, since analogies can be misleading as well as informative.

[*]This report may be used for any nonprofit purpose provided that its source is acknowledged. It will be adapted for inclusion in the third edition of my *Principles of Programming Languages*.

In this essay I'll address the role of aesthetics in programming language design, but I will base my remarks on a book about structural engineering, *The Tower and the Bridge*, by David P. Billington. Although there are many differences between bridges and programming languages, we will find that many ideas and insights transfer rather directly.

According to Billington, there are three values common to many technological activities, which we can call "the three E's": *Efficiency, Economy and Elegance.* These values correspond to three dimensions of technology, which Billington calls the *scientific, social* and *symbolic* dimensions (the three S's). We will consider each in turn.

2 Efficiency Seeks to Minimize Resources Used

In structural engineering, *efficiency* deals with the amount of material used; the basic criterion is safety and the issues are *scientific* (strength of materials, disposition of forces, etc.). Similarly, in programming language design, efficiency is a scientific question dealing with the use of resources. There are many examples where efficiency considerations influenced programming language design (some are reviewed in my *Principles of Programming Languages*). In the early days, the resources to be minimized were often runtime memory usage and processing time, although compile-time resource utilization was also relevant. In other cases the resource economized was programmer typing time, and there are well-known cases in which this compromised safety (e.g. FORTRAN's implicit declarations). There are also many well-known cases in which security (i.e. safety) was sacrificed for the sake of efficiency by neglecting runtime error checking (e.g. array bounds checking).

Efficiency issues often can be quantified in terms of computer memory or time, but we must be careful that we are not comparing apples and oranges. Compile time is not interchangeable run time, and neither one is the same as programmer time. Similarly, computer memory cannot be traded off against computer time unless both are reduced to a common denominator, such as money, but this brings in economic considerations, to which we now turn.

3 Economy Seeks to Maximize Benefit versus Cost

Whereas efficiency is a scientific issue, *economy* is a *social* issue. In structural engineering, economy seeks to maximize social benefit compared to its cost. (This is especially appropriate since structures like bridges are usually built at public expense for the benefit of the public.) In programming language design, the "public" that must be satisfied is the programming community that will use the language and the institutions for which these programmers work.

Economic tradeoffs are hard to make because economic values change and are difficult to predict. For example, the shift from first to second generation programming languages was largely a result of a decrease in the cost of computer time compared to programmer time, the shift from the second to the third generation involved the increasing cost of residual bugs in programs, and the fourth generation reflected the increasing cost of program maintenance compared to program development.

Other social factors involved in the success or failure of a programming language include: whether major manufacturers support the language, whether prestigious universities teach it, whether it is approved in some way by influential organizations (such as the US Department of Defense), whether it has been standardized, whether it comes to be perceived as a "real" language (used by "real programmers") or as a "toy" language (used by novices or dilettantes), and so forth. As can be seen from the historical remarks in my *Principles*, social factors are frequently more important than scientific factors in determining the success or failure of a programming language.

Often economic issues can be quantified in terms of money, but the monetary values of costs and benefits are often unstable and unpredictable because they depend on changing market forces. Also, many social issues, from dissatisfaction with poorly designed software to human misery resulting from system failures, are inaccurately represented by the single dimension of monetary cost. All kinds of "cost" and "benefit" must be considered in seeking an economical design.

4 Elegance Symbolizes Good Design

"Elegance? Who cares about elegance?" snorts the hard-nosed engineer, but Billington shows clearly the critical role of elegance in "hard-nosed" engineering.

4.1 For the Designer

It is well-known that feature interaction poses a serious problem for language designers because of the difficulty of analyzing all the possible interactions of features in a language (see my *Principles* for examples). Structural engineers face similar problems of analytic complexity, but Billington observes that the best designers don't make extensive use of computer models and calculation.

One reason is that mathematical analysis is always incomplete. The engineer must make a decision about which variables are significant and which are not, and an analysis may lead to incorrect conclusions if this decision is not made well. Also, equations are often simplified (e.g., made linear) to make their analysis feasible, and this is another potential source of error. Because of these limitations, engineers that depend on mathematical analysis may overdesign a structure to compensate for unforeseen effects left out of the analysis. Thus the price of safety is additional material and increased cost (i.e. decreased efficiency and economy).

Similarly in programming language design, the limitations of the analytic approach often force us to make a choice between an under-engineered design, in which we run the risk of unanticipated interactions, and an over-engineered design, in which we have confidence, but which is inefficient or uneconomical.

Many people have seen the famous film of the collapse in 1940 of the four-month-old Tacoma Narrows bridge; it vibrated itself to pieces in a storm because *aerodynamical* stability had not been considered in its design. Billington explains that this accident, along with a number of less dramatic bridge failures, was a consequence of an increasing use of theoretical analyses that began in the 1920s. However, the very problem that destroyed the Tacoma Narrows bridge had been anticipated and avoided a century before by bridge designers who were guided by aesthetic principles.

According to Billington, the best structural engineers do not rely on mathematical analysis (although they do not abandon it altogether). Rather, their design activity is guided by a sense of *elegance*. This is because solutions to structural engineering problems are usually greatly underdetermined, that is, there are many possible solutions to a particular problem, such as bridging a particular river. Therefore, expert designers restrict their attention to designs in which the interaction of the forces is easy to see. The design looks unbalanced if the forces are unbalanced, and the design *looks* stable if it *is* stable.

The general principle is that designs that *look* good will also *be* good, and therefore the design process can be guided by aesthetics without extensive (but incomplete) mathematical analysis. Billington expresses this idea by inverting the old architectural maxim and asserting that, in structural design, **function follows form**. He adds (p. 21), "When the form is well chosen, its analysis becomes astoundingly simple." In other words, the choice of form is open and free, so we should pick forms where elegant design expresses good design (i.e. efficient and economical design). If we do so, then we can let aesthetics guide design.

The same applies to programming language design. By restricting our attention to designs in which the interaction of features is manifest — in which good interactions look good, and bad interactions look bad — we can let our aesthetic sense guide our design and we can be much more confident that we have a good design, without having to check all the possible interactions.

4.2 For the User

In this case, what's good for the designer also is good for the user. Nobody is comfortable crossing a bridge that looks like it will collapse at any moment, and nobody is comfortable using a programming language in which features may "explode" if combined in the wrong way. The manifest balance of forces in a well-designed bridge gives us confidence when we cross it. So also, the manifestly good design of our programming language should reinforce our confidence when we program in it, because we have (well-justified) confidence in the consequences of our actions.

We accomplish little by covering an unbalanced structure in a beautiful facade. When the bridge is unable to sustain the load for which it was designed, and collapses, it won't much matter that it was beautiful on the outside. So also in programming languages. If the elegance is only superficial, that is, if it is not the manifestation of a deep coherence in the design, then programmers will quickly see through the illusion and loose their (unwarranted) confidence.

In summary, good designers choose to work in a region of the design space where good designs look good. As a consequence, these designers can rely on their aesthetic sense, as can the users of the structures (bridges or programming languages) they design. We may miss out on some good designs this way, but they are of limited value unless both the designer and the user can be confident that they are good designs. We may summarize the preceding discussion in a maxim analogous to those in my *Principles of Programming Languages*:

The Elegance Principle
Confine your attention to designs that *look* good because they *are* good.

5 The Programming Language as Work Environment

There are other reasons that elegance is relevant to a well-engineered programming language. The programming language is something the professional programmer will live with — even live *in*. It should feel comfortable and safe, like a well-designed home or office; in this way it can contribute to the quality of the activities that take place within it. Would you work better in an oriental garden or a sweatshop?

A programming language should be a joy to use. This will encourage its use and decrease the programmer's fatigue and frustration. The programming language should not be a hindrance, but should serve more as a collaborator, encouraging programmers to do their jobs better. As some automobiles are "driving machines" and work as a natural extension of the driver, so a programming language should be a "programming machine" by encouraging the programmer to acquire the smooth competence and seemingly effortless skill of a virtuoso. The programming language should *invite* the programmer to design elegant, efficient and economical programs.

Through its aesthetic dimension a programming language symbolizes many values. For example, in the variety of its features it may symbolize profligate excess, sparing economy or asceticism; the kind of its features may represent intellectual sophistication, down-to-earth practicality or ignorant crudeness. Thus a programming language can promote a set of values. By embodying certain values, it encourages us to think about them; by neglecting or negating other values, it allows them to recede into the background and out of our attention. Out of sight, out of mind.

6 Acquiring a Sense of Elegance

Aesthetics is notoriously difficult to teach, so you may wonder how you are supposed to acquire that refined sense of elegance necessary to good design. Billington observes that this sense is acquired through extensive experience in design, which, especially in Europe, is encouraged by a competitive process for choosing bridge designers. Because of it, structural engineers design many more bridges than they build, and they learn from each competition they loose by comparing their own designs with those of the winner and other losers. The public also critiques the competing designs, and in this way becomes more educated; their sense of elegance develops along with that of the designers.

So also, to improve as a programming language designer you should design many languages — design obsessively — and criticize, revise and discard your designs. You should also evaluate and criticize other people's designs and try to improve them. In this way you will acquire the body of experience you will need when the "real thing" comes along.

7 References

1. Billington, David P., *The Tower and the Bridge: The New Art of Structural Engineering*, Princeton: Princeton University Press, 1983. Chapters 1 and 6 are the most relevant.

2. MacLennan, Bruce J., *Principles of Programming Languages: Design, Evaluation, and Implementation*, second edition, New York: Holt, Rinehart & Winston (now Oxford University Press), 1987.