Elegance

Matthew Fuller

In *Literate Programming*,[1] Donald Knuth suggests that the best programs can be said to possess the quality of elegance. Elegance is defined by four criteria: the leanness of the code; the clarity with which the problem is defined; spareness of use of resources such as time and processor cycles; and, implementation in the most suitable language on the most suitable system for its execution. Such a definition of elegance shares a common vocabulary with design and engineering, where, in order to achieve elegance, use of materials should be the barest and cleverest. The combination is essential—too much emphasis on one of the criteria leads to clunkiness or overcomplication.

Such a view of elegance is supported by Gregory Chaitin's formulation of program-size definition of complexity: A measure of the complexity of an answer to a question is the size of the smallest program required to compute it. The resulting drive to terse programs produces a definition of elegance being found in a program "with the property that no program written in the same programming language that produces the same output is smaller than it is."[2]

The benefit of these criteria of elegance in programming is that they establish a clear grounding for the evaluation of approaches to a problem. This set of criteria emerging from programming as a self-referent discipline it works on the level of disciplinary formalization, as a set of metrics that allow for a scale of abstraction. This formalization can also be politically crucial as a rhetorical and intellectual device that allows programmers to stake their ground in contexts where they might be asked to compromise the integrity of their work, and something that allows them to derive satisfaction from work that might otherwise be banal.

When writing code to test compilers, Knuth takes the opposite route. He writes test programs that are, "Intended to break the system, to push it to its extreme limits, to pile complication on complication, in ways that the system programmer never consciously anticipated." He continues, "To prepare such test data, I get into the meanest, nastiest frame of mind that I can manage, and I write the cruelest code I can think of; then I turn round and embed that and embed it in even nastier constructions that are almost obscene."[3] There is a clear counter-position between code that contains as much vileness as one could want and model code that is good. For users of software configured as consumers such "metaphysical" questions aren't often the most immediately apparent, although questions of elegance, as will be suggested below are also recapitulated at the scale of interface.

To return to the politics of elegance at the level of programming practice it is also useful to think about those contexts where paradoxically, in order to become more adequately self-referent, the process of writing software finds itself constituted in combination with other elements. In working conditions where programmers might be concerned with conserving elegance against other imperatives, such as the cutting of costs, the criteria are often posed in terms of benign engineering common sense, or the ethics of satisfying the needs of the user in the clearest way possible, or the onus of clarity to one's collaborators. Elegance is often invoked defensively. In each case however, elegance remains a set of parameters against which a program can be measured.

In the four criteria proposed by Knuth, elegance is constructed between the machine and the talents of the programmer, with the context of the program occurring as something already filtered into a problem definition. Elegance in this sense is defined by its containment within programming as a practice that is internally self-referent and stable.

Knuth's criteria for elegance are immensely powerful when evaluating programming as an activity in and of itself. It might be useful, however, to think about the terms against which they might be modifiable, or for the context of elegance to be allowed to roam, to make obscene couplings, to find other centers of gravity. In such cases, software is not simply software, and it in turn conjugates those other realities with which it mixes with computation. Different criteria for elegance pour into the domain of software, and those of software begin to manifest in combination with other scales of reality.

At the same time, something interesting happens to stability at the level of software. Further work by Gregory Chaitin has revealed that the decision as to whether a program is the shortest possible is complicated by a fundamental incompleteness.[4] As a program's complexity increases, and concomitantly that of the problem it deals with, there is an increasing difficulty in accurately stating the most concise means of answering it. At a certain threshold, the possibility of stating the tersest formula for arriving at an answer is undecidable. The elegance of software then, by at least one of the above criteria, is not absolutely definable at a mathematical level. This is not the same as saying, as of software debugging, "If you don't have an automated test for a feature, that feature doesn't really exist."[5] Elegance, because it cannot be proven, comes down to a rule of thumb, something that emerges out of the interplay of such constraints, or as something more intuitively achievable as style (in Knuth's terminology, an "art"). Like William Burroughs' proposal for an informal self-discipline of movement, "Do Easy,"[6] it is something that can be practiced and learned, the dimensions, weights, capacities of objects dancing in an endless dynamic geometry incorporating the body of the adept and the repositories of heuristics that have gone before in the form of languages, institutions, archives, books, and techniques. Eventually, a certain effortlessness is achieved.

Effortlessness is offered straight out of the box in the vision of computing which sees interaction with information as being best achieved through simple appliances that are easy to use and which operate with defined, comprehensible scopes. At this point, elegance gives way to another set of criteria, which provide powerful, occasionally even fundamental constraints. Such constraints

act as limiting devices that force a piece of software toward elegance. A condition of elegance, however, is that it charts a trajectory, often an unlikely one, through possible conditions of failure. Finding a way of aligning one's capacities and powers in a way that arcs through the interlocking sets of constraints and criteria, the material qualities of software, and the context in which it is forged and exists is key to elegance.

Achieving striking effects with an economy of means has been crucial to formulating elegance within software, particularly within the domain of graphic interaction. To produce a convincing animated sprite within a tiny cluster of pixels, to develop a bitmapped font working at multiple scales, or to develop a format allowing for the fast transfer and calculation of vector graphics over limited bandwidth requires a variation in criteria from those Knuth set for elegance at the level of programming. (For instance, one might be working for a predefined platform or a range of them, or within a particular protocol.) Equally, at the level of the operating system, a language, a data-structure, or within a program, defining the core grammars of conjunction and differentiation of digital objects each provide scalar layers wherein elegance might be achieved or made difficult. In such cases, elegance can be found in the solutions that allow a user to get as close to the bare bones of the underlying layer of the system, without necessarily having to go a layer deeper. In proprietary software, a good example of such elegance is the formulation of the Tool Kit, built into the ROM of the early Macintoshes, which defined the available vocabulary of actions, such as cut, paste, save, copy, and so on that were able to work powerfully across many different applications.[7] Such work builds upon the particularity of digital and computational materials. Crucially, however, it also abstracts from the many potential kinds of interaction with data that might be desirable to produce a limited range of operations that can be deployed across many different kinds of information. While the range of such a vocabulary of functions might be constrained, the concrete power that arises from the conjoint and recursive use of these operations elegantly directs the power of computation in a trajectory toward its conjugation with its outside. The outside in this case consists of the multiple uses of these functions in programs aimed at the handling of multiple kinds of data. Elegance then is also the capacity to make leaps from one set of material qualities and constraints to another in order to set off a combinatorial explosion generated by the interplay between their previously discrete powers.

Elegance can also be seen in the way in which a trajectory can be extended, developing the reach of an abstraction, or by finding connections with do-

mains previously understood as being "outside" of computation or software. A fine example of such elegance would be achieved if a way was found to conjoin the criteria of elegance in programming with constraints on hardware design consonant with ecological principles of nonpollution, minimal energy usage, recyclability or reusability, and the health requirements of hardware fabrication and disposal workers.[8] Good design increasingly demands that elegance follows or at least makes itself open to such a trajectory. The criteria of minimal use of processor cycles already has ecological implications.

While elegance, then, demands that we step outside of software, keep combining it with other centers of gravity, computation also suggests a means by which we can think it through, prior to its formulation. The virtual has become an increasingly significant domain for philosophical thought, but it is also one that is always simultaneously mathematical. Steven Wolfram's figure of the "computational universe"[9] suggests that it is possible to map out every possible algorithm, and every possible result of every algorithm. A concept of the virtual reminiscent of Linneas's attempts to graph the entirety of speciation, this is a decisive imaginal figure, if not quite a mapping, of the constraint of computability itself. It follows from Emile Borel's idea that it would be possible to construct a table containing every possible statement in the French language, and indeed from Turing's formalization of all possible computations. Needless to say, Borel's table did not account for irony, that multiple semantic layers can be embedded in the same string of characters. If an ironic computational universe is not the one we currently inhabit, it will inevitably occur as soon as computation snuggles up to its outside. The condensation of multiple meanings into one phrase or statement turns elegance from a set of criteria into a, necessarily skewed, way of life.

Here we can see a further clue to elegance within multiscalar domains, that is to say, how it is produced in most actual computing work. The transversal leap or arc characteristic of elegance does not necessarily depend on a structural, ethical, or aesthetic homomorphy between code, the problem it treats, and the materials it allies itself with (such as hardware, language and people). Elegance also manifests by means of disequilibrium, the tiny doses of poison, doping, required to make a chip functional, to make it hum: a hack can be elegant, a good hack is inherently so. Elegance exists in the precision madness of axioms that cross categories, in software that observes terseness and clarity of design, and in the leaping cracks and shudders that zigzag scales and domains together.

Notes

1. Donald Knuth, *Literate Programming.*

2. Gregory Chaitin, *Epistemology as Information Theory: From Leibniz to Omega.* See also Gregory Chaitin, *MetaMaths! The Quest for Omega.*

3. Donald Knuth, *Literate Programming,* 266–267.

4. Gregory Chaitin, "Elegant LISP Programs" in Cristian Calude, *People and Ideas in Theoretical Computer Science,* 35–52.

5. Eric Kidd, "More Debugging Tips."

6. William S. Burroughs, "The Discipline of DE."

7. The Mac ToolKit was programmed by Andy Hertzfeld; see his *Revolution in the Valley.*

8. See Basel Action Network, available at http://www.ban.org/; Silicon Valley Toxics Coalition, available at http://www.svtc.org/; Greenpeace, *Green My Apple Campaign,* available at http://www.greenmyapple.org/.

9. See Steven Wolfram, *A New Kind of Science.*