Aesthetics in Software Engineering

Technical Report UT-CS-06-579

Bruce J. MacLennan*

Department of Computer Science
University of Tennessee, Knoxville
www.cs.utk.edu/~mclennan

October 31, 2006

Abstract

This report discusses the important role that aesthetics may play in software engineering. We begin with an exploration of the practical importance of elegance for both the designers and users of software systems, and argue that it promotes software that is technically superior and a pleasure to use. Second, because of the abstract and formal character of software we draw analogies with aesthetics in the exact sciences, including mathematics, in which intelligibility coincides with beauty, and consider how this aesthetics may be applied to software. Third, we discuss means for making abstract aesthetic qualities perceptible, including visual programming languages and models grounded in human embodiment. Finally, we advocate ways to advance and teach the aesthetic dimension of software engineering.

1 Introduction

Software engineering is a historically new activity, so it does not have as long an aesthetic tradition as do the other arts and engineering disciplines. Therefore it is helpful to begin our aesthetic inquiry with analogies to longer-established disciplines, always keeping in mind the distinctive characteristics of software. In this article I will draw analogies principally from two sources. First, because software systems are large and complex, often constructed by teams, intended to serve a useful function, and capable of causing injury and economic loss if they fail, I will draw analogies from the structural engineering

* This report is an extended draft of an article invited for *Handbook of Philosophy of the Technological Sciences*, "Part IX. Normativity and Values in the Technological Sciences," "7. Aesthetic Values in Technology and Engineering Design," "5. Designing the Virtual: Software Engineering."

of towers and bridges (e.g., Billington, 1983), which shares these characteristics. This will lead to an exploration of the practical importance of elegance for both the designers and users of software systems. Second, because of the abstract and formal character of software I will draw analogies with aesthetics in the exact sciences, including mathematics (e.g., Heisenberg, 1975). Next I will discuss means for making abstract aesthetic qualities perceptible, including visual programming languages and models grounded in human embodiment. Finally, I will consider how we may advance and teach the aesthetic dimension of software engineering.

2 Importance of Aesthetics in Software Engineering

2.1 Designer's Perspective

Following Billington (1983) we may identify three dimensions along which designs may be evaluated: *efficiency*, *economy*, and *elegance* — "the Three E's." These correspond to three aspects of any artifact, the *scientific*, *social*, and *symbolic* ("the Three S's"). *Efficiency* deals with the physical resources used by the system, which in the case of software artifacts is primarily computer time and memory. Typically there are trade offs involved, with efficiency weighed against factors such as functionality, reliability, and maintainability. These are *scientific* issues because they concern the physical resource utilization of the system's design.

Economy refers to all aspects of the cost of the system, including hardware and human costs, in all phases, including development, use, and maintenance. These are *social* issues because costs depend on market forces, social processes, governmental policies, etc. Due to the uncertainties in these factors, the economy of a design is more difficult to evaluate than its efficiency, and it is subject to change and local context. Furthermore, it cannot be assumed that all costs can be reduced to a common denominator, such as money, as is often the case with human suffering.

This brings us to the explicitly aesthetic dimension of a design, its *elegance*, which depends on the aspects that Billington calls *symbolic*. Although we may take for granted that aesthetically appealing designs should be preferred, other things being equal, there are other compelling reasons for preferring elegant designs, but to understand them we need to review some of the characteristics of software systems. (See also MacLennan, 1999, pp. 156–60.)

Modern software systems can be enormously complex, often comprising millions of lines of instructions. Even a text editor, generally considered a basic software tool, can be hundreds of thousands of lines in length (e.g., the open source "vim 7.0" editor has approximately 300 000 lines of source code). The steady increase in software complexity has resulted from a number of factors (both scientific and social), including the increasing capacity and speed of computer systems, users' demands for new features and richer interfaces, and competing systems with more features.

Software systems with such large numbers of instructions are among the most complex systems ever constructed, and analytic tools for understanding them (such as program verifiers and test generators) are still quite limited. The complexity results in part from the fact that these millions of components interact with each other (and with other software and hardware systems) in real time, and that the number of interactions to be

considered increases with at least the square of the number of components. Furthermore, the components (e.g., computer instructions) are far removed from physical objects and interactions for which we have an intuitive basis for understanding (e.g., the physical components and interactions of a mechanical system). Therefore, our intuition is set adrift, and our analytical tools do little to anchor it.

Every analysis makes idealizing simplifications, and generally, the more complex the system, the greater will be the simplifications in its analysis. In the case of physical systems, for example, we may assume that the dynamics is linear, because that simplifies the mathematical analysis (or makes it feasible), even though we know that it is nonlinear. In the case of a software system, we may assume that any two numbers can be added and that the result will be correct, although we know that computer arithmetic is limited in range and precision. Similarly we may assume that input-output processes and other system services will operate correctly and be completed within real-time constraints.

It is important to realize that simplification is an inherent limitation in the analysis of complex systems, since an analysis is supposed to separate out the relevant features of the system, so that we can understand them better (with our limited cognitive capacities), from the irrelevant features (which we intend to ignore). Therefore the validity and usefulness of an analysis depends on decisions (sometimes tacit) as to what the analysis should include or omit, which derive from assumptions (often unconscious) as to what is relevant or irrelevant. Furthermore, since human cognitive capacities are limited, the more complex a system is, the more must be omitted from its analysis so that the analysis itself will not exceed our understanding. Thus there are inherent limitations to the analysis of very complex systems, such as modern software systems.

Similar problems arise in structural engineering, and Billington observes that the best structural engineers are guided by aesthetics as well as by mathematical analysis. In elegant designs the dispositions of masses and forces are manifest in the design, and therefore the designs that look good (look balanced, secure, stable, etc.) correspond to the designs that are safe, efficient, and economical. For example, although extensive mathematical analysis was used in the design of the Tacoma Narrows Bridge, it collapsed four months after it was completed because aerodynamic stability had not been included in the analysis (it was not considered relevant). In contrast earlier bridges, designed without the benefit of complex mathematical models but in accord with aesthetic principles, were aerodynamically stable. How is it possible that good aesthetics can lead to good engineering?

Billington observes that in structural engineering, designs are under-determined, that is, there are many designs that will solve a particular structural engineering problem, such as bridging a certain river (see also Ferguson, 1992, p. 23). Therefore, in contrast to Louis Sullivan's architectural maxim, *form follows function*, which suggests that the design is strongly determined by its function, Billington argues that the more appropriate structural engineering maxim is *function follows form*, because there are many structures that will accomplish a particular function. The same arguments are even more applicable in software engineering, in which typically many different software designs will satisfy the system requirements. Therefore in software engineering we have a great deal of freedom in the choice of solutions to a software problem.

In particular, software engineers (like structural engineers) can choose to work in a region of the design space in which experience has shown that designs that *look good* in

fact *are good* (e.g., safe, efficient, and economical). In the case of towers and bridges, such designs make the interaction of forces manifest, so that designers (and, as we will see, users) can perceive them clearly. Since aesthetic judgment is a highly integrative cognitive process, combining perception of subtle relationships with conscious and unconscious intellectual and emotional interpretation, it can be used to guide the design process by forming an overall assessment of the myriad interactions in a complex software system.

The discussion thus far has focused on the cognitive aspects of aesthetics, for an elegant software system is easier to understand and can be designed more reliably than an inelegant one. Thus there are practical, engineering reasons for striving for elegance. However, aesthetics also plays a less tangible role, which may be called *ethical*, for a design symbolizes a set of values. Specifically, if a designer is seeking an elegant design, then they are being guided by a set of aesthetic values (which imply engineering values in the chosen subset of the design space). A design may be robust or delicate, spare or rich in features, straight forward or subtle, ad hoc or general, and so forth, and the values exemplified in the design will call forth extensions and modifications consistent with those values. By keeping certain values, embodied in the design aesthetic, before the designers' eyes, these values will be kept in their attention and persist as conscious goals. Conversely, values incompatible with the aesthetic, or not exemplified by it, will tend to recede into the background of the designers' minds, and will be underrepresented in the design. Thus a software system may embody a coherent ethical-aesthetic character, which is difficult to state in words but can guide the aesthetically sensitive engineer.

Aesthetic appreciation can unite a software development organization through a common set of values embodied in a shared sense of elegance. We can see a similar role for aesthetics among mathematicians and theoretical scientists, who strive for proofs and theories that are elegant. For example, Heisenberg (1975, p. 176) says that science "also has an important social and ethical aspect; for many men can take an active part in it." Scientists, he says, are like the master masons who constructed the medieval cathedrals, for "[t]hey were imbued with the idea of beauty posited by the original forms, and were compelled by their task to carry out exact and meticulous work in accordance with these forms" (ibid.). Like cathedrals and scientific theories, large software projects are the result of the efforts of many people, and aesthetic standards provide criteria by which individual contributions can be objectively evaluated (ibid.).

Thus, in software engineering, as in mathematics and theoretical science, correctness is required, but among the correct solutions, the more elegant are preferred. (The education of mathematicians and theoretical scientists also provides models for how a shared sense of software elegance might be learned.) Therefore a shared aesthetic sense can unite a software engineering team in a common purpose.

2.2 User's Perspective

Hitherto I have stressed the importance of aesthetics for the *designers* of software artifacts, but it is also important for the *users*. In the modern information economy many people spend much of their working lives interacting with one or a few software systems (e.g., a word processor, database system, or reservation system); further, in their recreational time, people may be engaged with the same or other software artifacts (e.g., a web browser or computer game). Therefore the external aesthetics of software systems can have a significant effect on the quality of many people's lives. Other things (such as

functionality) being equal, most people would prefer to work with a beautiful tool than with an ugly one.

Furthermore, for many people the computer is not simply one tool in an otherwise uncomputerized occupation; rather, the computer and its software constitute, to a large degree, the entire occupation. In these cases the software system defines the work environment as fundamentally as the physical workspace does. Therefore, the aesthetics of the software systems deserves at least as much attention as that due the architecture, decor, etc. (From this perspective, many contemporary programs are the software equivalent of sweatshops: cluttered, dangerous, ugly, alienating, and dehumanizing.) As architecture deals with the functionality and aesthetics of physical space, organizing it for practicality and beauty, so software engineers organize cognitive (or virtual) space toward the same ends. Thus software aesthetics can have a major effect on quality of work and quality of life.

An elegant software design can also promote confident use of the system, for eventually users will acquire an aesthetic sense of the design space and will come to recognize that the designs that look good also are good. As is well known, many people approach software fearfully, and part of this fear arises from the fact that software is unpredictable (for them, but often also for the designers!). In elegantly designed software, however, the dynamical interaction of the parts is manifest in the external form, and so aesthetic comprehension of the form can guide the user's understanding of the system's operation. Therefore, as in mathematics and theoretical science, the goal is that beauty coincide with intelligibility, for then users (as well as programmers) will experience pleasure through understanding. This is possible because both beauty and intelligibility are grounded in the interrelation of the parts, as will be argued later (sec. 5.4; cf. also Heisenberg, 1975, pp. 169–70).

It is well-known that people's ability to use technological devices with pleasure, confidence, and fluency depends on their ability to build a *cognitive* or *conceptual model* of the device's behavior (Norman, 1988, ch. 7; 1998, ch. 8; 2005, ch. 3). An effective cognitive model of a system is not required to reflect its actual internal structure or operation, but it must be accurate enough not to mislead the user (thus resulting in a loss of confidence and in frustration). By implying an intelligible dynamical structure, an elegant design can help the user to form an effective cognitive model. Therefore an elegant design aids users' understanding of a system in much the same way it aids that of the system's designers.

Similarly, just as for the designers the aesthetics of a design has an ethical dimension and exemplifies certain values to the exclusion of others, so also the design aesthetics has ethical implications for users. At very least, by making some practices easy and others awkward, and by bringing some concerns into the foreground while leaving others in the background, the external aspect of the system will influence users in its use. Indeed, such non-neutrality is an unavoidable characteristic of the phenomenology of all tools (Ihde, 1986, chs. 5, 6; 1993; MacLennan, 1999, pp. 33–35). In addition to this, however, is the symbolic dimension, for by exemplifying particular aesthetic norms, the system keeps these before the eyes of the users, and increases the likelihood that they will be guided by these norms in their own work.

Finally, there is a social aspect for the users of an elegant design just as there is for the designers. As users come to appreciate the beauty of an elegant design, they will de-

velop an appreciation for its aesthetic principles and come to expect similar elegance in other software systems. Thus the users (and consumers) of software systems are included in a feedback loop that encourages the development of elegant software and discourages the inelegant. This will accelerate the development of software that is efficient, economical, reliable, and a pleasure to use. (Billington notes the role of an aesthetically educated public in improving bridge design in Europe.)

3 Software Engineering as "Platonic Technology"

With only slight poetic license, software may be called "Platonic technology." To see this, we may analyze the products of the arts (including technology) in terms of *form* and *matter*, in which "matter" refers to the raw material on which the art imposes its form. In these terms, the *matter* of software engineering is the hardware, which provides a relatively neutral ground comprising the data processing operations and storage, and the *form* is the software which organizes these resources into a dynamic process in order to accomplish some purpose. Traditionally software engineering has focused on the software independent of the hardware, that is, on the form independent of the matter, and in this sense it is Platonic in spirit if not in essence (as are mathematics, theoretical physics, etc.). Although there is great variety in hardware and it changes every few years, we still study and use algorithms that were designed decades ago (some even before the invention of electronic computers). Clearly, what matters most is the form, not the matter.

All arts have their formal and material characteristics, but software engineering is exceptional in the degree to which formal considerations dominate material ones. All the issues that are most fundamental in software engineering (e.g., correctness, efficiency, understandability, maintainability) depend primarily on the formal characteristics of the program and only secondarily on its material embodiment (i.e., the effect of the hardware on the software). Clearly, the hardware cannot be ignored (especially in cases in which the engineering is pushed to its limits), but in general hardware considerations are secondary and often an afterthought.

Software engineering is a new discipline and so it does not have a well-established aesthetic tradition. We may look to other arts for suggestions and analogies, but software's lack of essential material embodiment implies that perceptual qualities will not have so great a role as they do in the other arts. Rather, aesthetic considerations in software engineering will be comparable to those in mathematics and theoretical science (which are also Platonic in that abstract ideas constitute their primary subject matter).

Indeed, discussions of the aesthetics of mathematics and theoretical science often focus on such qualities as correctness (either consistency or empirical adequacy), generality, simplicity, and (abstract) beauty, and the same qualities are central to the aesthetic evaluation of software. (See, for example, Curtin, 1982; Farmelo, 2002, Pref.; King, 2006; Wechsler, 1988.)

4 A Platonic Aesthetics for Software Engineering

The foregoing observations suggest that we may apply Platonic theories of aesthetics in software engineering, a proposition that is not as anachronistic as it may seem. For as Heisenberg remarks in his essay, "The Meaning of Beauty in the Exact Sciences" (1975), these aesthetic theories stem from the same roots as the exact sciences, exemplified in the

discovery, attributed to Pythagoras, that harmonious sounds correspond to intelligible mathematical relationships. For contemporary examples of the coincidence of beauty and correctness in the exact sciences Heisenberg mentions relativity theory and quantum theory. Another reason for looking to classical aesthetics is that the classical notion of beauty (Grk., *to kallon*) included excellence and manifest fitness to purpose, and so it has a functional aspect that is very appropriate to software engineering.

Since in classical aesthetics (and continuing into the eighteenth century) there were two principal approaches to aesthetics stemming from Platonic philosophy and differing in whether they focused on "the many" or "the one," I will consider the applicability to software of each of these theories, which we may call the Pythagorean (for it derives from Pythagorean philosophy) and the Neoplatonic (as represented in Plotinus' *Enneads*).

4.1 Conformity of the Parts to One Another and to the Whole

The Pythagorean theory grounds the beauty of a composite object in the conformity of its parts to one another and to the whole. As Plato (*Philebus* 64e) says, "The qualities of measure (*metron*) and proportion (*symmetron*) invariably … constitute beauty and excellence." That is, aesthetic excellence is grounded in order, harmony, symmetry, and proportion among the parts. Furthermore, since beauty is an objective formal or structural property, it has an intellectual character, and so beauty and intelligibility are inherently related (*Phil*. 65d). Further, as Plato (*Phaedrus* 250d) observes, vision and hearing, the senses most closely connected with rational cognition, are also the senses by means of which we are able to perceive physical beauty. Thus, the most intelligible structure corresponds to the most beautiful structure, as we have seen to be the case for elegant software.

Therefore in judging the aesthetics of a software system, we should look first to the conformity of the parts. They should be *harmonious* in a Pythagorean sense, for in ancient Greek *harmonia* refers to things that are well fitted together into a whole (Liddell, Scott & Jones, 1968, s.v. ἁρμονία). We ask: Are they arranged *symmetrically*, so they will be easier to comprehend? Are they *ordered* in an easily comprehensible way? Are the numbers and sizes of the elements appropriate? To the extent that these structural qualities obtain, the aesthetically sensitive engineer will experience the system as elegant and even beautiful. Of course aesthetic appreciation will be enhanced by a visual presentation that makes the intellectual order manifest (as is discussed later).

In classical aesthetics, however, beauty is a function not only of the interrelation of the parts, but also of the relation of the parts to the whole. For example, Aristotle (*Poetics* 23, 1458a) argues that a beautiful thing is an organic whole, which he explicitly analogizes to a single living thing in its unity (*zôon hen holon*). That is, the parts are not only orderly, symmetric, and harmonious in their interrelationships, but they all serve essential and complementary roles as parts of a meaningful, unified whole; the parts and the whole are mutually determining. Certainly, these are desirable criteria in any functional system.

Aristotle also argues that a beautiful work of art, like a beautiful organism, must be of a certain appropriate size, and he relates this size to human cognitive capacities, such as our memory and our ability to discriminate the work's parts and to grasp the work in its entirety (*Poet*. 7, 1450b-1451a). Similarly, beautiful software is structured in such a way that we can grasp the components in their individuality and relation, and comprehend the

whole in our minds as an organic and functional unity. In summary, aesthetic evaluation considers the relation of the parts to each other, to the whole, and to the perceiver.

Science attempts to comprehend a multiplicity of phenomena under a single principle, expressed as a simple, elegant mathematical relationship among abstract ideas. Most commonly the phenomena are dynamical relationships and processes evolving in time, and so, as Heisenberg explains (in the case of Newtonian mechanics), "The parts are individual mechanical processes … And the whole is the unitary principle of form which all these processes comply with [and which is expressed] in a simple system of axioms" (Heisenberg, 1975, p. 174). In science, then, as in art, "Beauty is the proper conformity of the parts to one another and to the whole" (loc. cit.).

The goals of the software engineer are similar to those of the scientist in that both are attempting to give a static abstract description of material processes and interactions taking place in time. One difference, of course, is that the scientist is trying to describe naturally occurring phenomena, whereas the engineer is attempting to design a static structure (program) that will generate the desired temporal interactions.

As mechanical processes are described by the axioms of Newtonian mechanics, so a program, contingent on external events, describes a set of possible execution sequences. Individual execution sequences are the parts with respect to the infinite set of all sequences, for which the program provides an *intensive* (finite) definition. Beauty, then, resides in the conformity of the execution sequences to each other and to the program. They should form a harmonious ensemble (*extension*) and have a simple relation to the program (*intension*). For elegant programs the dynamic possibilities (extension) will be easy to visualize from the generative form (intension). The engineers will have a reliable intuitive understanding of the consequences of their design.

Conversely, in designing a program, software engineers have certain desired execution sequences in mind, and they have to expand these in their minds into a coherent infinite set of possible sequences (conformity of the parts to one another). From this multiplicity of possible dynamics they need to derive a finite and unified static generative form (conformity of parts to the whole). Beauty resides in the simplicity, harmoniousness, orderliness, and symmetry of these relations, which elicit simultaneous intellectual and aesthetic appreciation.

4.2 The Character of the Whole

The Neoplatonic theory of beauty, as represented in Plotinus (esp. *Enn*. I.vi, V.viii, VI.vii) is more difficult to apply in software engineering, and in fact with respect to its application in the exact sciences Heisenberg (1975, p. 183) remarks, "in our own time it is hard to speak of beauty from this aspect, and perhaps it is a good rule to adhere to the custom of the age one has to live in, and to keep silent about that which it is difficult to say." Nevertheless, it will be worthwhile to see what it could contribute to our understanding of software aesthetics.

From the perspective of the opposition of *the many* and *the one*, Pythagorean aesthetics focuses on *the many* — the relation of the parts to one another and to the whole — while Neoplatonic aesthetics focuses on *the one*, the whole without reference to its parts. For Plotinus argues that mere orderliness of parts is neither necessary nor sufficient for beauty, but that a thing is beautiful because it embodies an ideal form, an abstract unity, which we recognize through the beautiful thing. Further, Plotinus' view is that there are

no *independent* conditions of beauty, such as symmetry, separate from the unifying idea. Orderliness of the parts, then, is a consequence of their subservience to the unifying idea. The aesthetic experience arises from our appreciation of the organic unity of the thing as a reflection of an ideal form, an appreciation that depends in part on a "resonance" or "congruence" to (activation of) the form's representation in the perceiver's mind. In more modern terms, we experience a perceptual Gestalt, or recognize an archetypal form rooted in human cognition.

In the context of software engineering these archetypal forms would be, for example, complete procedures, operations, or patterns of interaction that are innate or deeply ingrained in the viewer's unconscious mind. Although it is difficult to catalog these archetypal structures, it is not necessary to do so, for to a large extent we know them when they see them, because they engage our neurocognitive structures and elicit recognition and an aesthetic response. When we see an algorithm whose correctness and efficiency are intuitively obvious, we may be responding to such an ideal form. Heisenberg (1975, p. 175) similarly observes that in science an aesthetic response to the whole often precedes intellectual exploration of the details. He asks (ibid.), "How comes it that with this shining forth of the beautiful into exact science the great connection becomes recognizable, even before it is understood in detail and before it can be rationally demonstrated?" It is not a result of conscious analysis, for "Among all those who have pondered on this question, it seems to have been universally agreed that this immediate recognition is not a consequence of discursive (i.e., rational) thinking" (op. cit., p. 177). Indeed, thinkers as diverse as Kepler, Pauli, and Jung (Heisenberg, op. cit., 177–80) have attributed the process "to innate archetypes that bring about the recognition of forms" (p. 178). Thus Pauli (1955, p. 153): "As *ordering* operators and image-formers in this world of symbolical images, the archetypes thus function as the sought-for bridge between the sense perceptions and the ideas and are, accordingly, a necessary presupposition even for evolving a scientific theory of nature." It is important to seek these structures, for they represent the channels in which our thought is predisposed to flow (whether innately or through learning).

5 Visual Programming Languages

Because of the abstract nature of software, we have been focusing on Plato's aesthetics, but even he acknowledged that sensuously perceivable beauty is a means toward apprehension of the intellectual beauty of abstract forms. Therefore I will consider briefly the role of *visual* beauty in software design.

Visual programming languages (VPLs), in which programs are represented as two-dimensional figures rather than as text, have been investigated since the earliest days of electronic computing (e.g., AMBIT/G, SKETCHPAD), and VPLs continue to be developed (e.g., Alice, StarLogo TNG), especially for introductory programming instruction (e.g., Eades & Zhang, 1996; Stasko, Domingue, Brown & Price, 1998). In these languages formal relations between program parts are represented as spatial relations between visual forms. Early VPLs represented programs as flowcharts, in which connecting edges represented possible paths of control flow, but after the introduction of structured programming around 1970 it became more popular to represent visually the hierarchical structure of the program, which reflects both the logical and dynamical structure of a structured program.

Often visual representations of hierarchical program structure take the form of some kind of tree diagram. Sometimes these are graphs, in which leaves represent atomic program components (individual programming language statements), interior nodes represent composite program components, and edges connect composite components to their immediate constituents. A more recent style, facilitated by improved computer graphics capabilities, represents program components by two-dimensional shapes reminiscent of jigsaw puzzle pieces, which can be interlocked only in conformity with the programming language's syntax (e.g., Alice, StarLogo TNG).

Visual representations of hierarchical program structure would seem to be ideal as a medium for elegant program design, and they are certainly superior in this regard to flowcharts. By representing abstract relations spatially, they create a correspondence between the domains of abstract forms and of spatial forms, and facilitate the visual perception (and aesthetic appreciation) of well-organized, symmetric, and balanced structures; that is, beauty coincides with intelligibility. Unfortunately, in practice these visual representations have limitations, for even small program modules can be quite deeply nested (and it can be argued that for *very small* modules visual representation is not important). As a consequence, the visual representations can be quite large in whatever dimension represents nesting depth. Due to the limitations of human visual perception and practical computer screen size, we are faced with the undesirable alternatives of displaying the entire structure, but with many tiny components, which are difficult to discern, or of displaying only a portion of the structure at one time and having to use devices such as panning and zooming to explore the structure sequentially. Neither is conducive to Gestalt recognition of the program's structure, or to an intuitive intellectual comprehension and aesthetic appreciation of it. Perhaps the problem is that VPLs result in a too literal representation of program structure in perceptible form, and that an aesthetically satisfying *expression* of the design will require a less literal representation.

6 Embodiment and Non-Platonic Aesthetics

Fishwick and his colleagues have explored a more metaphorical approach to programming aesthetics (e.g., Fishwick, 2002). Noting that graphs are "largely devoid of texture, sound, and aesthetic content," he seeks to make software "more useful, interesting, and comprehensive" by an approach that begins with a model; this is the "craft-worthy, artistic step." The model is intended to be the usual representation of the software design, the textual program being relegated to a secondary, marginalized status comparable to assembly language. However, since most software concepts are abstract and do not have real-world correspondents, they are represented *metaphorically*. Therefore, once the model is determined an aesthetic must be chosen as a foundation for the metaphors. For example, if architecture were chosen, then abstract control-flow relations in the program could be represented by corridors in a building through which avatars move. Notice that such a metaphorical representation recruits our embodied understanding of physical space and motion to improve our understanding of the program (see below). Similarly, our aesthetic understanding of architectural space guides the design of the program and our aesthetic and intellectual appreciation of it. The metaphorical model is the principal representation of the software, which becomes an object of aesthetic expression and appreciation, thereby enriching the experience of software. Fishwick notes that even three-dimensional visual programming languages tend to use simple iconography rather than sensuously rich objects: "One is aesthetically-challenged and Platonic whereas the other promotes famil-

iar sensory appeal." Fishwick (2006) contains recent contributions to *aesthetic computing* ("the impact and effects of aesthetics on the *field of computing*," p. 3)

Recent developments in psychology have illuminated the essential role played in cognition by embodiment, thus confirming insights from phenomenological philosophy and psychology (e.g., Gibbs, 2006). Much of our understanding of the world is rooted in our sensorimotor capacities, both those that are part of our genetic inheritance, and those that are acquired, especially in early childhood. Indeed, Lakoff & Núñez (2000) have argued that our understanding even in such abstract domains as mathematics is built on a network of interrelated metaphors grounded in sensorimotor skills. For example, at an intuitive level abstract sets are understood as physical containers, abstract trajectories as paths through physical space, and so forth.

All human beings have an enormous repertoire of sensorimotor skills, and it is normal to feel pleasure when acting skillfully, competently, and fluently, and to be dissatisfied otherwise; this is part of the feedback mechanism that increases the range and depth of our skills. Therefore to the extent that users' interactions with a system, such as a program, are accomplished through an existing repertoire of sensorimotor skills, they will feel competent and satisfied when they use it. In this way, aesthetic appreciation arises from the correspondence between people's embodied skills and the sensorimotor interface and abstract structure of the system, which is a different sort of resonance or congruence between the system and human cognitive structures.

Therefore aesthetic appreciation and satisfaction will be improved if a system and its parts, including the interface, behave similarly to the physical world, including the objects and processes that are familiar to most people. For example, if when we pull on or drag an object on a computer screen it behaves similarly to a physical object (e.g., in terms of stretching or inertia), then our sensorimotor skills will be engaged, and our skillful manipulation will be pleasurable (Karlsson & Djabri, 2001).

7 Applying Aesthetic Principles in Software Engineering

The foregoing remarks have merely sketched an approach to an aesthetic theory appropriate to software engineering, and so it will be worthwhile to say a little about how such a theory might be further developed. We can progress by four simultaneous activities, which we may call *experiment* , *criticism* , *theory* , and *practice* .

Experiment refers to learning by means of the self-conscious practice of the *art* of program design and the empirical evaluation of the results. For this to be effective, software engineers must be aware of aesthetic issues during the design, and they must evaluate the aesthetics of the resulting designs as experienced by themselves and others (evaluated phenomenologically and statistically). This entire activity presupposes greater aesthetic awareness in programmers.

Criticism plays an important role in all of the arts, most obviously to provide the general public with aesthetic evaluations, but more importantly to make various aesthetic issues salient, which influences the aesthetic sensibilities of both the producers and consumers of art. Even when artists disagree with criticism, they are encouraged to defend their aesthetic choices in word or deed. Thus criticism provides an important feedback loop that can improve artistic quality. To accomplish this we need more published *aes-*

thetic criticism of software, both of its external appearance and behavior, and of its internal structure and design, focusing on aesthetics in both cases.

Theory refers to the use of research results from cognitive neuropsychology and allied fields, which will continue to provide insights into the qualities that make something simultaneously intellectually comprehensible and aesthetically pleasing (that is to say, *elegant*). Theoretical understanding contributes by explaining the results of previous aesthetic experiments, and by suggesting new ones.

In spite of all the foregoing, the art of program design is neither a body of theory nor a set of design rules; rather, it is a *practice*. Both the long history of aesthetic debate and the analogy of aesthetic considerations in mathematics and the exact sciences suggest that beauty is an illusive concept. Therefore, in programming as in the other arts, while many aesthetic principles can be stated explicitly, others must remain implicit and essentially embodied in the practices of skilled artisans. This raises the question of how software aesthetics can be taught, which I'll consider briefly.

Explicit aesthetic principles can, of course, be taught and practiced, as they have been in the arts for centuries. This does not imply that software should adhere to rigid quantitative rules of symmetry and proportion, as were taught in the neoclassical aesthetics of the Renaissance, or that the principles are inviolable. Indeed, most aesthetic principles are qualitative guidelines that serve to focus the artist's attention on important issues, but do not compel their aesthetic choices, which are subject to a broader-based aesthetic judgment (as well as other, practical considerations).

Therefore, as in all forms of expert behavior (Dreyfus & Dreyfus, 1986), aesthetic rules are applied within an unformalizable context of practices and concerns, which constitutes the background to the principles, which are in the foreground. Thus we may expect that software designers will acquire their aesthetic sensibilities in much the same way that students in theoretical science and mathematics acquire theirs: by exposure to existing elegant designs and by mutual evaluation and criticism of prior and new designs. For maximum effectiveness, these educational activities should be incorporated in all computing courses and not confined to an aesthetics course. In this manner, over time, an authentic software aesthetics can evolve.

8 Conclusions

Although classical aesthetics may be criticized, as it has been since at least the eighteenth century, for its limitations as a general theory of aesthetics, I have argued that the Platonic character of software makes the classical theory an appropriate basis for an aesthetics for software engineering, in which beauty coincides with intelligibility, as it does in the exact sciences at their best. With this approach, abstract interrelationships are reflected in perceptible forms, so elegant structures engage both our cognitive faculties and our embodied understanding, leading to aesthetic appreciation and pleasure, which in turn serve as guides for the intuitive design of intellectually manageable software.

9 Bibliography

Billington, D. (1983). *The Tower and the Bridge*. New York, NY: Princeton University Press.

Curtin, D. (Ed.). (1982). *The Aesthetic Dimension of Science*. New York, NY: Philosophical Library, Inc.

Dreyfus, H., & Dreyfus, S. (1986). *Mind over Machine*. New York, NY: Free Press.

Eades, P., & Zhang, K. (Eds.). (1996). *Software Visualization*. New York, NY: World Scientific Publishing Company, Inc.

Farmelo, G. (Ed.). (2002). *It Must Be Beautiful*. New York, NY: Granta Books.

Ferguson, E. S. (1992). *Engineering and the Mind's Eye*. Cambridge, MA: MIT Press.

Fishwick, P. (2002). "Aesthetic programming: Crafting personalized software," *Leonardo* **35** (4): 383–390.

Fishwick, P. (2003). "Aesthetic computing manifesto," *Leonardo* , **36** (4): 255–256. [not cited]

Fishwick, P. (Ed.). (2006). *Aesthetic Computing* . Cambridge, MA: MIT Press.

Gelernter, D. (1998). *Machine Beauty: Elegance and the Heart of Technology* . New York, NY: Basic Books. [not cited]

Gelernter, D. (1998). *Aesthetics of Computing*. New York, NY: Phoenix House. [not cited]

Gibbs, R. W., Jr. (2006). *Embodiment and Cognitive Science*. New York, NY: Cambridge University Press.

Heisenberg, W. (1975). "The Meaning of Beauty in the Exact Sciences," in W. Heisenberg, *Across the Frontiers* , transl. Peter Heath. New York, NY: Harper & Row, 1975, pp. 166–83.

Ihde, D. (1986). *Consequences of Phenomenology* . Albany, NY: State University of New York.

Ihde, D. (1993). *The Philosophy of Technology: An Introduction* . New York, NY: Paragon House.

Karlsson, P., & Djabri, F. (2001). "Analogue styled user interfaces: An exemplified set of principles intended to improve aesthetic qualities in use," *Proceedings of Mobile HCI 2001: Third International Workshop on Human-Computer Interaction with Mobile Devices* .

King, J. (2006). *The Art of Mathematics*. New York, NY: Dover Publications.

Lakoff, G. & Núñez, R. E. (2000). *Where Mathematics Comes From: How the Embodied Mind Brings Mathematics Into Being* . New York, NY: Basic Books.

Liddell, H. G., Scott, R., & Jones, H. S. (1968). *A Greek-English Lexicon, With a Supplement, 1968*, 9th ed. Oxford: Oxford University Press.

MacLennan, B. J. (1999). *Principles of Programming Languages: Design, Evaluation, and Implementation* , 3rd ed. New York, NY: Oxford Univ. Press.

Norman, D. (1988). *The Psychology of Everyday Things* . New York, NY: Basic Books.

Norman, D. (1998). *The Invisible Computer*. New York, NY: MIT Press.

Norman, D. (2005). *Emotional Design*. New York, NY: Basic Books.

Pauli, W. (1955). “The influence of archetypal ideas on the scientific theories of Kepler,” in C. G. Jung & W. Pauli, *The Interpretation of Nature and Psyche* . New York, NY: Pantheon Books, pp. 147–240.
Stasko, J., Dominue, J., Brown, M. H., & Price, B. A. (Eds.). (1998). *Software Visualization*. New York, NY: MIT Press.
Wechsler, J. (1988). *On Aesthetics in Science*. New York, NY: Birkhauser Verlag.