Post-Modern Software Development

Software development is an art. The most noteworthy computer science monograph of the 20th century was, after all, Knuth's "The Art of Computer Programming" [http://www.americanscientist.org/template/BookReviewTypeDetail/assetid/26575;jsessionid=aaa6UkBHpVJBaY] [1]. One of the starting points for this column is Knuth's Turing award, a discussion of art in computer science [2].

Art is a word with many meanings. It originally referred to skill of joining or fitting. (Much [too much for we traditionalists] of software creation these days is the art of connecting existing pieces.) The meaning of "art" expanded to the system of principles and rules for attaining a desired end. Art stands in contrast to science, engineering, manufacturing, and fashion. Science distills knowledge into principles and laws; art recognizes that there are human choices in activities. Combine art with an attention to economy and one gets engineering, like the computer science holy grail of "software engineering." Do something following a well-defined and low-skill plan and one gets manufacturing. Choosing among equivalent possibilities is fashion. Designing a computer is an art. Designing one that people will buy is engineering. Building one from that design is manufacturing. Picking the color for the computer case is fashion. (Art is also a synonym for necromancy, a topic of clear relevance to computer science.)

Art also refers to the use of skill to create that which is esthetically or intellectually pleasing. Fine arts show an intellectual progression through the ages, shaped by new technology (e.g., casting, cameras, computers), shifting economic forces (the decline of the church, the rise of the merchant class, ultimately the emergence of mass media with the entire population as customers for art), new understandings (e.g., perspective, the physics of light and sound) and evolving response to the ideas of the previous generation (e.g., baroque, realistic, impressionist, "modern" art). In fine arts, prior themes are revisited with new twists. Science and engineering show an unconditional progression; nothing will make us return to Newtonian mechanics, view of non-Euclidean geometry as heresy, or replace integrated circuits with discrete transistors. Disciplines like education and organizational management follow fashions—old "truths" reemerge. "The Management Style of Attila the Hun" and "The Management Style of Saint Francis of Assisi" are equally likely to be business best sellers at some arbitrary point in the future.

The art of software development

The aesthetic metric in science is accuracy and simplicity. Art encompasses aesthetic metrics such as beauty, intellectual progression and quality of workmanship. Engineering includes reliability and economy of construction. We expect our software systems to satisfy a large range of ilities, including an aesthetic of understandability; ease of construction, maintenance and evolvability; an economy of execution; reliability; security from attack; interoperability; and so forth. Software has a special place in the range of artifacts, as it intimately connects the mathematical, physical and psychological realms. Psychology plays a dual role in software systems, both in software creation and use.

The history of software development has shown aspects of art, science, engineering and fashion (but very little manufacturing). Such intellectual developments show eras: the baroque gives way to the rococo, romanticism, modern art, post modern art, and so forth. In software, the early emphasis on functional development yielded to structured programming, and, over the last twenty years, the rise of computer science's modernism, object-oriented technology. Along the way we've seen offshoots like functional, logic and rule-based systems. Artistic development has been characterized by first, improved ability for rendering concrete realism and later attempts to express more in a work of art than the literal interpretation of its content—to express richer relationships and to tie the work of art into the context of its developer and environment.

One can see a similar progression in software development. Earliest programming languages were concerned with efficient realism: it was hard to render even highly structured problems into code; efficient use of machine resources was a dominant design criterion. Programming was *linear*: things said in the program tended to correspond, one to one, to things that happened when the program executed. Programming was *planar:* One could easily trace the potential execution paths in the program and identify which conditions would give rise to which code being called. As software systems became more plastic and complex, new technologies came to dominate. Today we have objects. Programmers are instructed to think of the elements of their domain and of their implementation as "things" with "state" and "behavior," and to code that state and behavior. Linearity and planarity decreased. Inheritance allows statements asserted in distant ancestors to intrude in program execution; dynamic object binding draws bridges over the program surface.

Post-Modern Programming

In all domains, old ideas give way or evolve to new ones. What is the post-modern programming equivalent? That is, what comes after object-orientation? Broadly, object orientation suffers from several limitations:

- All meaning is wrapped up in the code. There are few ways to say anything about the system that aren't about how the system executes. Comments, UML diagrams, and similar documents are, of course, an exception to this. I'm not suggesting skipping comments in code, but such elements are notoriously unreliable and non-automatable. One the other hand, type systems represent a first step toward *annotation*. By declaring something to be a type, the programmer conveys more than just implementation. Additionally, tools have evolved to help process the type information. In the future, we are likely to see richer uses of annotation in programming, with annotations tied not only to program execution but also to program analysis and understanding. Such annotations may range from simple propositional elements to describing invariants that the system ought to maintain, and ought to include constraints about how program elements can be composed and extended.
- The great wisdom of objects was to bring together all everything about something—the code of an object includes its data, behavior and interfaces. Unfortunately, the real world isn't that simple. Many things we care about with

respect to code are not neatly localized into a single place in an object-oriented decomposition. Current technologies force scattering of these concerns throughout a system. Future programming environments may provide mechanisms for expressing and enforcing *crosscutting concerns* and the ability to make *quantified statements* about the overall behavior of systems, having those assertions enforced throughout the system.

- Things aren't really that simple as to just be atomic. The real world (and real data structures) are composites of collections of elements (e.g., the wheels are part of a car, objects in themselves but also having a special relationship with the car and each other), collections, and masses. The opposite side of this coin is that models of elements in the real world and partial computations on these elements have a conceptual existence beyond a single execution of a program. Post-modern programming will need ways to provide such persistence.
- Software doesn't work. My browser crashes periodically. Even the Mars Rovers, with their inherent difficulty of on-sight repairs, have software bugs. I won't begin to get into my experiences with a popular desktop operating system. Post-modern software systems may come to recognize that such failures are the norm, not the exception and will be built to deal with unexpected failures rather than be surprised by them.
- Classical programming is like call and response music. One asks a question (makes a subprogram call) and gets a response. Post-modern software may explore other options, including event-based systems, conversational communication, and context-sensitive evaluations.
- Early analysts of programming languages put a lot of stock in language syntax, such as the choice of keywords for particular operations. Modern programming language analysis rushes to dispense with syntactic sugar to get to operational, semantic meat. A post-modern world may find a different balance between universal, common ways of expressing programs and notations that are specific to particular domains or eccentric for particular programmers. Matching such *domain-specific* syntax will be domain-specific semantics—software languages with inherent facilities for particular problem domains.

Post-object Languages

New movements in the art of software are often heralded by new programming languages (for example, "object-oriented languages"). When the linguistic idea takes hold, the support structure for that language emerges (for example, "object-oriented software analysis and design"). In a future column, I plan to examine some of the language and environment trends that are trying to overcome object limitations.

And speaking of promises, last January I promised to report on how the Mars Exploration Rover/Collaborative Information Portal system worked out. I've postponed that reporting because, well, the damn things are still working, even though they're out of warranty. This issue includes a paper (not by me) with a more complete description of that system and its use, which I figure is dispensation from that prior promise.

1. Donald E. Knuth, *The Art of Computer Programming*, Reading-Massachusetts: Addison-Wesley, 1968–20xx.

2. Donald E. Knuth, "Computer programming as an art," *Communications of the ACM, 17* , 12 (December 1974) pp 667 – 673.