Networked: Postmodern Software Development

Robert E. Filman
RIACS/NASA Ames Center

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,[1] and one of the starting points for this column is his Turing award, a discussion of art in computer science.[2]

Art is a word with many meanings. It originally referred to the skill of joining or fitting. (Much—too much, for us 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 we get engineering, like the computer science holy grail of "software engineering." Doing something following a well-defined, low-skill plan yields manufacturing. Choosing among equivalent possibilities is fashion. Designing a computer is an art. Designing one that people can afford 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 aesthetically or intellectually pleasing. Fine arts show an intellectual progression through the ages, shaped by new technology (for example, casting, cameras, and computers), shifting economic forces (including the decline of the church, the rise of the merchant class, and ultimately, the emergence of mass media with the entire population as customers for art), new understandings (such as perspective and the physics of light and sound) and evolving response to the previous generation's ideas (baroque, realism, impressionism, and modernism, for example). 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 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's dual role in software systems plays out in both software creation and use.

The history of software development includes elements of art, science, engineering, and fashion (though very little manufacturing). Such intellectual fields have eras: in the fine arts, the baroque gave way to rococo, romanticism, modernism, postmodernism, and so forth. In software, the early emphasis on functional development yielded to structured programming, and, over the past 20 years, the rise of computer science's modernism: object-oriented technology. Along the way, we've seen offshoots such as functional, logic, and rule-based systems. Artistic development has been characterized first by the improved ability to render concrete realism and, later, by attempts to express more in a work of art than the literal interpretation of its content—conveying richer relationships and tying the work into the context of its developer and environment.

Software development shows a similar progression. The 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: you 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, new, more complex technologies came to dominate. Today we have objects. Programmers are instructed to think of the elements of their domain and their implementation as "things" with "state" and "behavior," and to code that state and behavior. Linearity and planarity have decreased. Inheritance allows statements asserted in distant ancestors to intrude in program execution; dynamic object binding draws bridges over the program surface.

Postmodern Programming

In all domains, old ideas give way or evolve to new ones. What is the postmodern 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 a system that aren't about how the system executes. Comments, Unified Modeling Language (UML) diagrams, and similar documents are, of course, exceptions. I'm not suggesting skipping comments in code, but such elements are notoriously unreliable and nonautomatable. On 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. This information can be used in ways beyond code generation. In the future, we will likely see richer uses of annotation in programming, tied not only to program execution but also to program analysis, understanding, and tuning. Such annotations might range from simple propositional elements to descriptions of invariants that the system ought to maintain, including constraints about how program elements can be composed and extended.

Novel Modularizations

The great wisdom of objects was to bring together everything about something—an object's code 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 developers to scatter these concerns throughout a system. Programmers must develop code that tangles such crosscutting concerns. Future programming environments might provide mechanisms for discretely expressing crosscutting concerns while nevertheless assembling working systems. This can be understood as an instance of the more general idea of being able to make and enforce statements about a program's behavior without having to tie such statements to singular or particular decompositions.

Complex Representations

Object orientation encourages us to think of systems as discrete, weakly related atoms. On the other hand, the real world (and real data structures) exhibit complex collections of elements. For example, the wheels are part of a car—objects in themselves, but also maintaining a special relationship with the car and each other. Similarly, both the world as a whole and software systems exhibit a variety of semantic collections and masses. Additionally, real-world and software elements often need to have lifetimes beyond the execution of a single program (persistence). Postmodern programming will have ways to express and maintain such complex relationships and durations.

Software Doesn't Work

My browser crashes periodically. Even the Mars Rovers, with their inherent difficulty of on-site repairs, have software bugs. (I won't begin to get into my experiences with a certain popular desktop operating system.) Postmodern software systems might come to recognize that such failures are the norm, rather than the exception, and that developers need tools for building systems that can cope with unexpected failures rather than be surprised by them.

Conversations

Classical programming is like call-and-response music. Programming languages let us ask a question (make a subprogram call) and get a response. Postmodern software might explore other options, including event-based systems, conversational communication, and context-sensitive evaluations.

Adaptable Language

Early programming-language analysts put a lot of stock in language syntax, such as keyword choice for particular operations. Modern programming-language analysis rushes to dispense with syntactic sugar to get to operational-semantic meat. A postmodern world might find a different balance between universal, common ways of expressing programs and notations that are specific to certain domains or to particular programmers' eccentricities. Matching such domain-specific syntax will be domain-specific semantics—software languages with inherent facilities for problem domains.

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 will examine some of the language and environment trends that are trying to overcome object limitations.

And speaking of promises, last January, I said I'd report on how the Mars Exploration Rover/Collaborative Information Portal system worked out. I've postponed that reporting because, well, the darned things are still working—even though they're out of warranty. This issue of IC includes an article (not by me) with a more complete description of that system and its use ("The Collaborative Information Portal and NASA's Mars Rover Mission," pp. 20–26), which I figure is enough to earn dispensation from my prior promise.

References

1. D.E. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1968.
2. D.E. Knuth, "Computer Programming as an Art," *Comm. ACM*, vol. 17, no. 12, 1974, pp. 667–673.