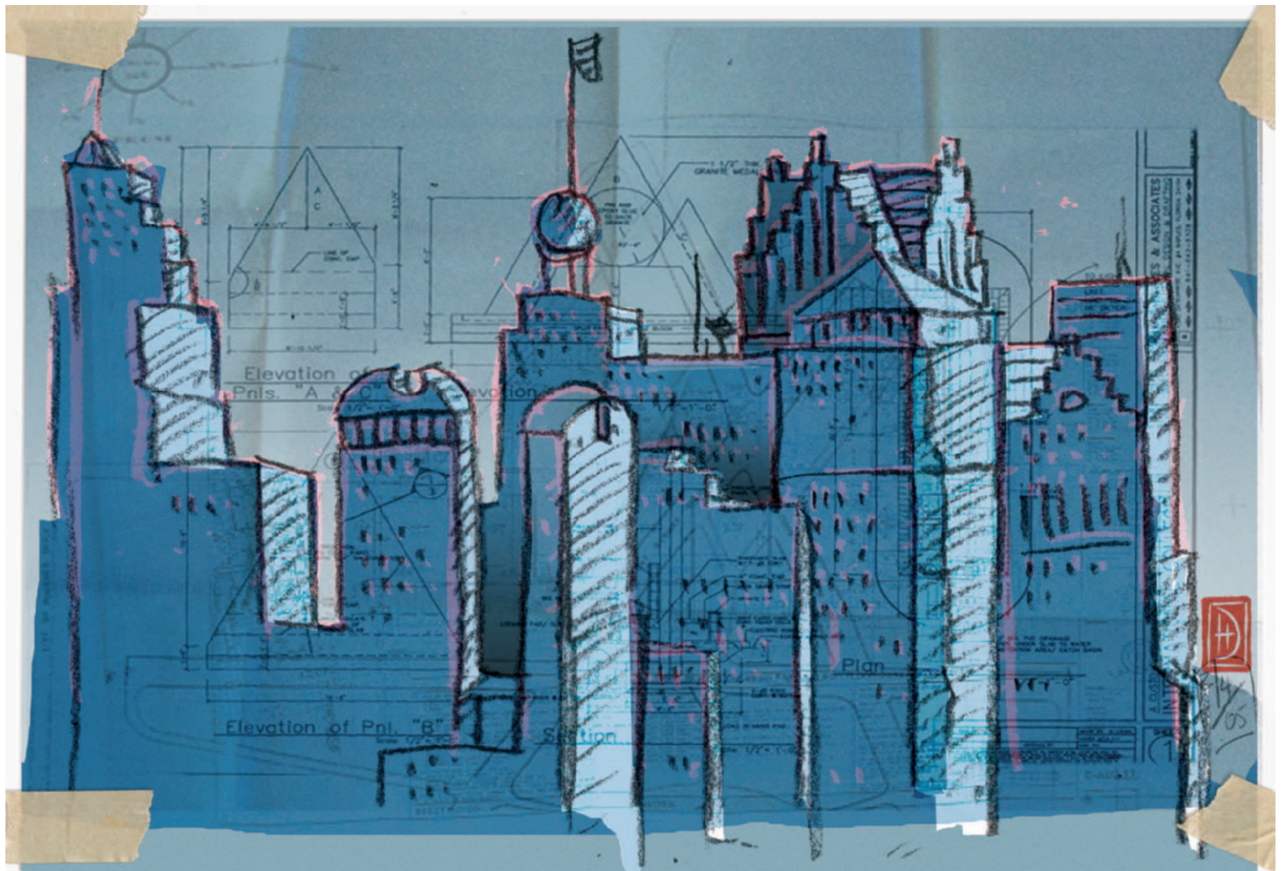
Software Design in a Postmodern Era

Philippe Kruchten, *University of British Columbia*

Over the last 30 years, software design has made tremendous progress. But this progress hasn't been continuous: it proceeded by jumps and leaps, with some plateaus in between. For example, after the plateau of structured methods and functional decompo-



 0740-7459/05/$20.00 © 2005 IEEE

sition, object-oriented design approaches surfaced, then bounced and leaped through the late '80s and early '90s. It has now reached a plateau. It's gradually become mainstream, codified, and partly standardized in the form of UML.

Reaching a plateau isn't at all negative; it's a necessary step for every discipline to have time to integrate good practice, to reflect, and to produce a critique that will launch further progress. Any bold advance needs some time to mature—to "cross the chasm," as Geoffrey Moore eloquently described, to get a critical mass of practitioners across our industry beyond the eager early-adopter stage.[1] Techniques, practices, and methods must be taught in schools and must be supported by tools. They must prove their value beyond any reasonable doubt and sometimes even be enshrined in some industry standard. All this takes time and effort—hence, the value of having these plateaus.

Over time, as we developed larger and more complex systems, we realized that software design has several nested levels, from the module or class level up to the system or enterprise architecture level. It was exactly 10 years ago that *IEEE Software* had its first special issue on software architecture,[2] then still described as an "emerging discipline" by Mary Shaw and David Garlan.[3] Software architecture also seems to have reached a plateau today, while its concepts and techniques continue to percolate in our software processes.

Postmodern programming

Perhaps we've reached another, more fundamental plateau, wittily called the era of "postmodern programming" by James Noble and Robert Biddle.[4] Computer science hasn't achieved the grand narrative that explains it all, the *big picture*—we haven't found the fundamental laws of software that would play the role that the fundamental laws of physics play for other engineering disciplines. We still live with the bitter aftertaste of the Internet bubble burst and the Y2K doomsday. So, in this postmodern era, where it seems that everything matters a bit yet not much really matters, what are the next directions for software design? Where will be the next plateau?

We should ask ourselves two questions regarding software design: Where are we? And where do we want to go from here? And maybe even, what exactly is software design?

Where are we?

Answering "Where are we?" was one of the intentions of the *Guide to the Software Engineering Body of Knowledge* project. SWEBOK took a broad and shallow perspective to answer this for design. In this issue of *IEEE Software*, Javier Garzás and Mario Piattini go narrower and deeper on the topic of design knowledge. They show how we can systematically harvest and organize our collective knowledge, experience, and wisdom—variously captured in design principles and heuristics, patterns, best practices, and bad smells—into a coherent whole. To do this, they offer us an ontology for microarchitectural design knowledge.

In another article in this issue, six pioneers of the concept of software architecture and architectural reviews share some 17 years of experience and practice in more than 800 projects in their four companies. Clearly, the practice of architectural review has matured and proven its benefits.

Where do we go from here?

Noble and Biddle see "scrap-heap software development" ahead of us.[4] But this seems very ad hoc, and limited to small software development efforts. We need a better, grander vision for software design. The process of designing software must be made to fit better with the surrounding engineering processes, both upstream and downstream. Our processes are not seamless. Upstream, there's still a wide gap between users' needs and the way we express requirements on one hand, and our designs and the way we design on the other. The Standish Group reports make this clear: the primary cause of failure in our software endeavors is our inability to deal correctly with users and their changing needs.[5,6] We try to alleviate this with various means—XP's onsite customer, for example. Downstream, we still struggle to analyze our designs, to demonstrate that they're correct and that they fulfill the requirements. And finally, there's still a gap between our designs and the code that the programmer fills manually. All these gaps have become narrower in the last 15 years, but they're still a major obstacle to our industry consistently producing great software products.

Aspect-oriented software development might be one way to reduce the gaps, both upstream and downstream. Upstream, AOSD provides a more natural way to express some of the non-

Reaching a plateau is a necessary step for a discipline to have time to integrate good practices.

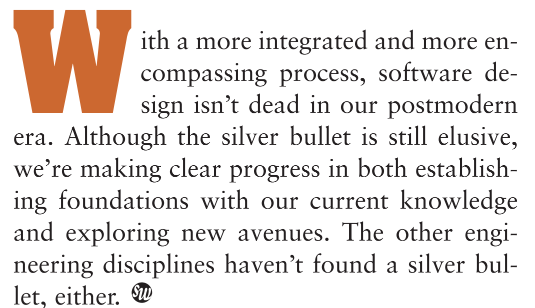
functional requirements. At the same time, downstream, it weaves the appropriate code more or less automatically, reducing the need for programmers to translate (typically an error-prone and tedious process). We're planning a special issue of *IEEE Software* on AOSD for early next year (see the call for articles in this issue on page 95).

Model-driven development resolutely attempts to reduce the gap downstream. It aims to give software designers the means to express a large spectrum of semantics in their designs at the model level rather than at the code level, and then to provide automated ways to produce a compliant program. In their article in this issue, Guy Caplat and Jean-Louis Sourrouille introduce us to the concept of model mapping: how we translate concepts and entities from one model to another—in particular, from a platform-independent model to a platform-specific one. While the jury is still out on the best way to produce domain-specific languages, these authors are strong advocates for using language extensions (as opposed to language modifications).

Do we have our noses too close to our blueprints? Many researchers and practitioners want to shift our focus from design elements (classes, subsystems, interfaces, and so on) to the design decision itself. This more central concept would become a first-class citizen in the process of designing software-intensive systems and, in particular, architectural decisions. Jeff Tyree and Art Ackerman offer a convincing problem analysis of existing architectural approaches that fail to treat design decisions as first-class entities. The authors recognize the importance of decision making in the architecting process, which is critical to system development and maintenance, and they argue that we can make decisions systematically and document them in a useful way. Managing design decisions might be the key to end-to-end traceability—the solution to capturing design rationale, analyzing the impact of projected changes, or harvesting reusable know-how when simply reusing the code isn't feasible.

The boundaries of software design

If we are to further narrow the gaps between requirements engineering (upstream) and programming (downstream), we might ask, What are the boundaries of software design? In my article in this issue, I suggest that we are designing even when we don't call it that. In other words, when we elicit and capture requirements and when we program and test, we're making decisions about the system under construction: this is doing design. Software design is therefore a wider concept than we usually think. To convince you, I cast our concept of software development in a more general framework of engineering design, the Function-Behavior-Structure framework developed by architect John Gero.



With a more integrated and more encompassing process, software design isn't dead in our postmodern era. Although the silver bullet is still elusive, we're making clear progress in both establishing foundations with our current knowledge and exploring new avenues. The other engineering disciplines haven't found a silver bullet, either.

About the Author

Philippe Kruchten is a professor of software engineering at the University of British Columbia. (His full biography appears on p. 58.) Contact him at kruchten@ieee.org.

References

1. G. Moore and R. McKenna, *Crossing the Chasm: Marketing and Selling Technology Products to Mainstream Customers*, Harper Business, 1991.
2. *IEEE Software*, special issue on software architecture, vol. 12, no. 6, 1995.
3. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
4. J. Noble and R. Biddle, *Notes on Postmodern Programming*, tech. report CS-TR-02/9, School of Mathematical and Computing Sciences, Victoria Univ., New Zealand, 2002.
5. *The Chaos Report*, Standish Group, 1995.
6. *Extreme Chaos*, Standish Group, 2001.