

## The Accidental Architecture

**Grady Booch**

**E**very interesting software-intensive system has an architecture. While some of these architectures are intentional, most appear to be accidental.

Since *accidental* is an emotionally explosive word, let's tease apart the elements of my statement. First, the terms *interesting* and *software-intensive*. For my purposes, an interesting system is one that has significant

economic value; a software-intensive system is one that involves some degree of software/hardware interplay, such as that found not only in large distributed systems but also in smaller embedded systems or even captive uniprocessor or multicore systems. Second, the term *architecture* itself.

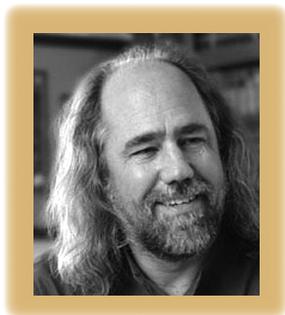
Here, I'm not so much concerned about the definition (the terminology in *IEEE Standard 1471* is quite sufficient for my needs) but rather the naming of particular architectural patterns. I'll say more about this later, but the fact that we cannot yet meaningfully enumerate a comprehensive set of architectural patterns or styles across domains is a gap in our understanding of software architecture. (Filling that gap is one desired outcome of my *Handbook* project.) Finally, the terms *intentional* and *accidental*. An intentional architecture is explicitly identified and then implemented; an accidental architecture emerges from the multitude of individual design deci-

sions that occur during development, only after which can we name that architecture.

Philippe Kruchten has observed that "the life of a software architect is a long and rapid succession of suboptimal design decisions taken partly in the dark." The journey between vision and ultimate executable system is complex. For every interesting software-intensive system, that path is marked by myriad decisions, some large and some small, some of which advance progress while others represent vestigial dead ends or trigger points for scrap and rework. As Philippe also observes, the architecture of classical systems comes primarily from theft, whereas the architecture of unprecedented systems comes largely from intuition carried out in the context of a controlled exploratory process. The fact that this is so for software-intensive systems shouldn't come as a surprise, for as Henry Petroski explains in his book *To Engineer Is Human* (Vintage, 1992), all sound engineering disciplines advance by building on past successes while simultaneously mitigating causes of observable failure.

Thus, having an accidental architecture is not necessarily a bad thing, as long as

- the decisions that make up that architecture are made manifest and
- the essential ones are made visible as soon as they are instituted and then are allowed to remain visible throughout the meaningful life of that system.



Insofar as we can then name these architectures after they're formed, we can use these names and their associated semantics to communicate decisions using a common language, just as we can do now with design patterns, and perhaps even reuse these architectural patterns in future systems. In other words, by naming these accidental architectures, we again raise the level of abstraction by which we can describe and reason about a system.

## Architectural patterns

In her book *The Grammar of Architecture* (Bulfinch, 2002), Emily Cole enumerates a set of 18 distinct architectural styles in civil engineering, ranging from Babylonian to Grecian and Islamic to Baroque, then on to Neoclassical and Picturesque. Her set isn't exactly complete or of suitable granularity—one could argue that Victorian, Craftsman, or even Gehrian are equally valid styles—but the fact that such things are even nameable reflects that field's maturity.

Gehrian? Well, therein lies a conundrum. The contemporary architect Frank Gehry, just like Daniel Libeskind, I.M. Pei, and many others, has a uniquely identifiable style that we don't quite know how to classify and thus name. Indeed, if you consider all the styles from Cole's book, you'll realize we can name them and distinguish one from another only because of the distance of time, having had the opportunity to reflect back on numerous instances of specific architectures and harvest their reoccurring patterns. The problem of naming software architectural patterns, therefore, is that we don't have a similar luxury of time or of as many identifiable reoccurring instances from which to harvest.

As part of my software archeological digs, I collect a lot of metadata, including the history of the system under study. In every case (taking into account that my work is self-selecting, for I'm only studying successful systems), I've uncovered a steady growth of architectural maturity within each domain. This maturity is represented by what evolutionists would call a punctuated rhythm consisting of architectural dis-

covery, architectural stability, architectural collapse, and then a repeated cycle initiated by a harvesting of architectural patterns that survived the collapse. In other words, accidental architectures emerge but then become intentional only after they've proven themselves through time.

## Web-centric architectural patterns

As an example, every contemporary dot-com system I've studied—particularly those that survived the near-death experience of the late 1990s dot-bomb era—have gone through several distinct architectural periods, periods that track the history of the Web itself. Many others have documented the Web's social, economic, and business history (and writing a history of something that's only 38 years old is disturbing enough, given that I'm only 614 months old myself), so I've taken a stab at naming the various technical generations of Web-centric systems.

In the beginning, simple documents dominated the Web. You can easily identify sites of this era: they were stylistically distinct, consisting primarily of modestly formatted text, basic hyperlinks, and a few crudely placed static graphics, reflecting the simplicity of the HTML standards of the time. In the next identifiable movement, we saw the rise of colorful clients. The first generation of Web-centric systems had suffi-

cient value that they attracted graphic designers (many from the print media) as well as serious developers. However, many of the designers who could build attractive sites couldn't create good software, and many of the developers who could build great software couldn't create approachable user experiences. In this generation, we saw the rise of eye candy (and the now-amusing but the then-ever-annoying HTML `<blink>` tag) but also more precise formatting and more engaging content.

It took a period of time coupled with natural market forces, but best practices began to emerge that yielded a balance of power between the stakeholders responsible for presentation and infrastructure. This led to a third generation focused on simple scripting. The rise of scripting languages began, and thus Perl, PHP, and others started to take hold. In this generation, we saw the first great schism surrounding the theology of dynamic HTML, with Microsoft's Active Server Pages on one side and, well, not Microsoft (including Java Server Pages) on the other.

Consistent with this model of punctuated equilibrium, the fourth generation was best identified by the rise of middleware, which continued the previous schism (leading to Microsoft's .NET and the alternative products supporting J2EE) and which represented the codification of many of the mechanisms commonly found in such systems.

In the fifth generation, the focus of innovation turned back again to the client layer above this middleware. With by-then-considerable experience in scripting languages and with the still-oscillating pendulum swinging between pure Web thin clients and traditional thick clients, the mark of this generation was the rise of simple frameworks (such as Struts). These served to codify best practices at the client layer, such as reflected in the Model-View-Controller pattern.

## And here we are

In the sixth (and contemporary) generation, an interesting bifurcation has taken place. On the edges of Web-centric systems are considerable demand pull

**Accidental  
architectures emerge  
but then become  
intentional only  
after they've proven  
themselves  
through time.**

and technology push toward rich clients, so technologies such as Ajax have gained traction. At the core of Web-centric systems, service-oriented architectures have taken hold because they provide a solution to the problem of building systems of systems. This is particularly true when some (but not necessarily all) of those systems are already wrapped up in the collateral implications of their Webification (namely, investment in the infrastructure elements of security, universal accessibility, and transparent location).

Were you to take a snapshot of an interesting Web-centric system during any one of these generations, you'd find a distinctly unique architectural style at play. Each generation was complex in its time, but the technology available at that time shaped the accidental architectures that emerged. Only now—in reflection of that time—can we begin to attempt to name and thus make intentional these architectural patterns.

A final point I need to make is that, depending on where you're looking, one person's system is another's subsystem. Thus, you might see the use of intentional architectures at some levels (only because we know how to build them and have built them before) but

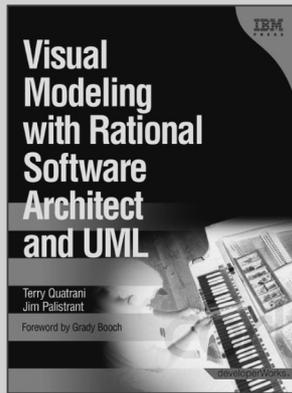
then accidental on others (because we're striking out on new ground or are assembling systems of systems in novel, heretofore untried ways).

**A**ccidental architectures are not evil things; indeed, they are inevitable in the growth of systems. It's only when we begin to turn these accidental archi-

tectures into intentional ones that we advance our understanding of software architecture. 

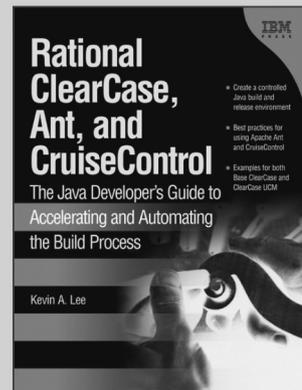
**Grady Booch** is an IBM Fellow. He's one of the Unified Modeling Language's original authors. He also developed the Booch method of software development, which he presents in *Object Oriented Analysis and Design*. Contact him at [architecture@booch.com](mailto:architecture@booch.com).

## SOFTWARE DEVELOPMENT EXPERTISE DELIVERED...



TERRY QUATRANI and  
JIM PALISTRANT  
ISBN: 0-321-23808-7

TURN TO THESE IBM PRESS PUBLICATIONS for software development learning from top experts in the field...



KEVIN A. LEE  
ISBN: 0-321-35699-3

For details on special discounts and to sample a free chapter of one of these IBM Press titles, visit our web site today!

Also newly available from Addison-Wesley: 

AGILITY AND DISCIPLINE MADE EASY:  
Best Practices from the Rational Unified Process  
PER KROLL  
ISBN 0-321-32130-8

<http://ibmpressbooks.com/ieeesoftware>  
AVAILABLE WHEREVER TECHNICAL BOOKS ARE SOLD.

**IBM**  
Press™

**IEEE**  
**Software**

**UPCOMING ISSUES:**  
Software Testing  
Global Software Development  
Software Engineering  
Curriculum Development



VISIT US AT  
[www.computer.org/software](http://www.computer.org/software)