

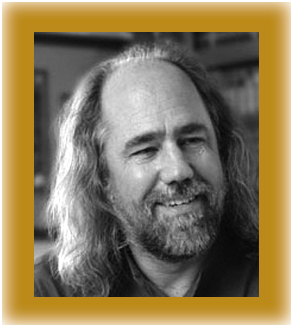
On Architecture

Grady Booch

For centuries, Ptolemaic theory reflected civilization's understanding of the earth and its place in the universe. To the pre-scientific mind, this geocentric viewpoint was just common sense: it was consistent with local observations and generated no dissonance with contemporary religious beliefs.

Unfortunately, Ptolemy's theory and most of the conclusions drawn from it were false.

It wasn't until the early 1500s that Copernicus rediscovered the reality of the earth turning on its axis daily and orbiting around the sun yearly, observations he published in *De Revolutionibus Orbium Coelestium* in 1543. Galileo's unabashed acceptance of Copernican theory landed him in trouble but, fortunately for us, hard scientific evidence tends to prevail. Later that century, the gentleman-scientist Tycho Brahe assembled a detailed,



precise, and comprehensive catalog of the motion of more than 1,000 stars and planets. After Brahe's death, Johann Kepler studied this data and then formulated his three laws of planetary motion, which he explained in *Astronomia Nova* and *Harmonices Mundi*.

This story's lesson is that classical science advances via the marvelous dance between quantitative observation and theoretical construction, and, in turn, the engineering disciplines apply those advances to creating new things. In stark contrast to the classical sciences, software engineering and its subdiscipline of software architecture are still in their infancy relative to observation and theory.

Software architecture's growth

Ten years ago, *IEEE Software* celebrated software architecture as an identifiable discipline, and the first International Software Architecture Workshop was held. Since then, the number of people who call themselves software architects has steadily increased (although few agree on what "software architect" means or what skills and activities it entails). Similarly, organizations have increasingly placed value on treating a system's software architecture as an artifact that they can create, grow, measure, and manage.

An engineering discipline shows signs of maturity when we can name, study, and apply the patterns relevant to that domain. In civil engineering, we can study the fundamental elements of architecture in works that expose and compare common architectural styles. Similarly, in chemical engineering, mechanical engineering, electrical engineering, and now even genomic engineering, libraries of common patterns have proven useful in practice.

Unfortunately, no such architectural reference yet exists for software-intensive systems. Although the patterns community has pioneered the vocabulary of design patterns through the work of the Hillside Group (<http://hillside.net>) and the Gang of Four (*Design Patterns*, Addison-Wesley, 1995), our industry has no parallel to more mature design disciplines' handbooks (see the "Other Disciplines' Handbooks" sidebar for some examples).

A handbook of software architecture

For the past two years, I've been working to create a handbook of software architecture (www.booch.com/architecture). I don't expect

to complete a critical mass of this work for another two to three years, simply because I'm collecting a broad set of data that's largely locked up in certain developers' heads and in internal documents that were rarely intended to see the light of day. In this ongoing column, I'll share some of my experiences as I continue my research.

The handbook's primary goal is to fill this empirical void in software engineering by codifying the architecture of 100 interesting software-intensive systems, presenting them in a manner that exposes their essential patterns and permits comparisons across domains and architectural styles. Second, the project aims to study these architectural patterns in the context of the engineering forces that shaped them in order to expose a set of proven architectural patterns that developers can use to construct new systems or reason about legacy ones. The third goal of this research is selfish—namely, to feed my insatiable curiosity. Whenever I encounter an interesting or useful software-intensive system, I ask myself, how did they do that? By studying their architectural patterns and thus exposing these systems' inner beauty, I hope to inspire developers who want to build on the experience of other well-engineered systems.

My motivation for this work comes from Bruce Anderson, who over a decade ago conducted a series of workshops at OOPSLA (ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications) to create *The Complete Handbook of Object-Oriented Software Architecture & Designs* in 1991. Since that time, the patterns community has grown quite vibrant, but most of its work has focused on design patterns. As such, around three years ago, I began sketching out an approach to continue Bruce's work, starting with a broad industry survey.

Software-intensive systems written several decades ago were complex in their own time; the systems we build today are equally complex. As intricacy has risen over the years, each new generation of developers has found

Other Disciplines' Handbooks

- *Building Embedded Linux Systems*, by Karim Yaghmour, O'Reilly, 2003
- *The Elements of Style: A Practical Encyclopedia of Interior Architectural Details from 1485 to the Present*, Simon and Schuster, 1997
- *The Phaidon Atlas of Contemporary World Architecture*, Phaidon Press, 2004
- *Perry's Chemical Engineers' Handbook*, McGraw Hill, 1997
- *Mechanism and Mechanical Devices Sourcebook*, McGraw-Hill, 2001
- *Electrical Engineers' Handbook*, McGraw-Hill, 1996

new ways to attack it. The nearly one trillion lines of code created over the decades provide a humbling reminder of our industry's breadth, depth, and reach; this body of work is so large that identifying a representative set of interesting systems for architectural study is difficult. Far more interesting systems are worthy of study than any one person could cover in a lifetime.

One way to prune the problem is to consider only contemporary systems, systems in production use at the time of investigation. Thus, I decided to leave the study of classic software for another project. (The Computer History Museum is engaged in a major effort to preserve a number of classic software systems' artifacts; www.computerhistory.org.) This still leaves an enormous number of systems to con-

sider, but I further pruned the problem by selecting a set of interesting systems balanced across different genres—specifically, artificial intelligence, commercial nonprofit, communications, content authoring, devices, entertainment and sports, financial, games, government, industrial, legal, medical, military, operating systems, platforms, scientific, tools, transportation, and utilities.

Selecting specific systems was another problem, one that was guided by my desire to present a set of interesting patterns. What you might consider interesting is subjective, but only a few unique architectural patterns appear to exist. For example, the basics of accounting have become reasonably stable over the centuries, and although every enterprise has its own set of schemas, rules, and policies, the architecture of such systems is remarkably similar. So, I'd like to present widely differing architectures across domains. Although practitioners in any given domain might consider their architectures obvious, few outside that domain would find them obvious at all.

Selecting the final systems for study within each genre still required a somewhat arbitrary and capricious decision process. When faced with a family of systems with a common architecture, I selected the one whose assets were more easily available. In a few cases, I made a specific selection simply because of the beauty of its architecture or because it was the seminal exemplar of its domain. Finally, I've intentionally balanced my selection globally: clearly, software innovation isn't constrained by political or geographical boundaries.

By exposing these systems' inner beauty, I hope to inspire developers who want to build on the experience of other well-engineered systems.

IEEE Software

EDITORIAL CALENDAR

2006

JANUARY/FEBRUARY

Aspect-Oriented Programming

MARCH/APRIL

Past, Present, and Future
of Software Architecture

MAY/JUNE

Requirements Engineering
Update: Best Papers of
IEEE RE '05

JULY/AUGUST

Software Testing

SEPTEMBER/OCTOBER

Global Software Development


NOVEMBER/DECEMBER

Software Engineering
Curriculum Development

To frame my research and my data collection, I've chosen to build on *IEEE Standard 1471, Recommended Practice for Architectural Description of Software-Intensive Systems*, along with Philippe Kruchten's work ("The 4+1 View Model of Architecture," *IEEE Software*, Nov. 1995). Essentially, the data set for each system under study is one instance of the IEEE metamodel augmented in three ways: by including information about

- the environment in which the system lives;
- the development team, tools, and processes; and
- a specific set of views.

The handbook is a work in progress. In some ways, there's no end because new systems worthy of study constantly emerge. As I said earlier, I plan to force closure to the present effort in two to three years.

Perhaps the most fascinating element of this research is that I'm expecting the unexpected: it's too early to draw conclusions about the commonality of patterns that I might find across such a wide set of domains. I hope you'll enjoy encountering the unexpected in my future columns. 

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.

