Using Patterns to Improve Our Architectural Vision

NORMAN L. KERTH, Elite Systems
WARD CUNNINGHAM, Cunningham & Cunningham

Pattern languages can play an important role in furthering the use of architecture and objects in software design. But first we must understand what these terms mean. The authors use the work of Christopher Alexander to illuminate the problems and shed light on future directions in our use of pattern languages in design.

Despite their repeated use in software development, the terms "objects," "architecture," and, most recently, "patterns" are not generally understood. The situation is well characterized by an exchange in Lewis Carroll's *Through the Looking Glass* when Humpty Dumpty tells Alice, "When I use a word . . . it means just what I choose it to mean, neither more nor less."

Objects may pose less of a problem than the other terms, but even here misunderstanding is possible. For example, we have both used objects for more than a decade, and in our discussions we generally assume that "object" means a single unit that

- encapsulates routines and data,
- can inherit behaviors from a parent, and
- can exist as multiple instances.

Still, this simple definition does not seem to encompass the term "object" fully. When we say "object" we also imply a problem-solving strategy associated with object use. This strategy is ethereal and thus very difficult to talk about. Before you have experienced it, no words will help you understand; after you experience it, words don't exist to explain it.

0740-7459/97/$10.00 © 1997 IEEE

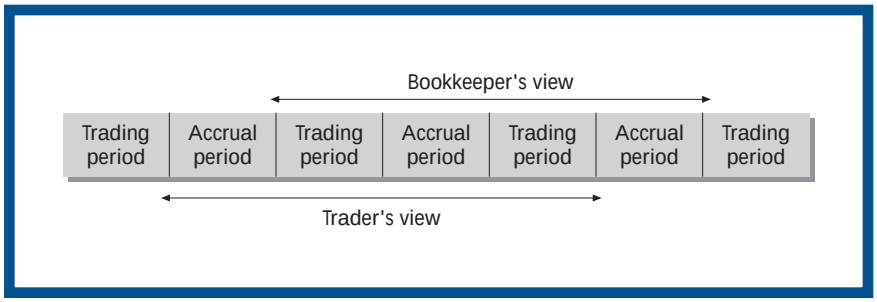

Figure 1. *Two views of a trading day based on different time intervals.*

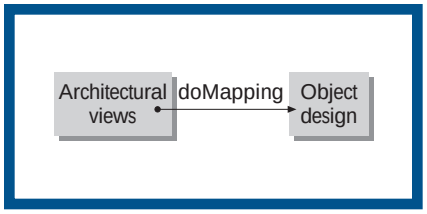

Figure 2*. A simple example of the mapping between architecture and objects.*

Christopher Alexander writes about this phenomenon, calling it a *quality-without-a-name*. Alexander explores synonyms that come close to describing the phenomenon—including alive, whole, comfortable, free, exact, egoless, and eternal—but concludes that "No word can ever catch the quality-without-a-name because the quality is too particular, and words too broad. And yet it is the most important quality there is, in anyone, or anything."[1]

Although we generally think of an object as being concrete, it is just this elusive and ethereal quality that makes the object concept special and important. It is a quality that exists in architecture as well, but in a more complicated way.

The notion of quality-without-a-name is deeply based in each of us. It helps us agree on creative greatness, whether it be in music, art, architecture, or software. Quality-without-a-name also drives us to develop a philosophy that underlies our own creative acts. The better we understand quality-without-a-name, and how our personal philosophy is developed, the more we will know about creating fine software architecture.

UNDERSTANDING ARCHITECTURE

In the November 1995 *IEEE Software* special issue on architecture,[2] there are six theme articles. In each article, authors define the term *architecture* differently, ranging from rules for assembling modules to multiple, cooperating models. Others have attempted to define the term as well.[3,4]

Defining views. Faced with this variety of definitions, we began to explore what they had in common. We found instead a range of meanings including

- a synonym for design,
- prototyping or early implementation,
- a high-level system overview, and
- an explanation of how a particular technology will be incorporated into a particular system.

We also found many different ways to represent architecture, including

- quick sketches on a white board, which are erased before the system enters maintenance;
- a generic, tiered model with pertinent names for different levels;
- elaborate English prose, usually understood only by the author; and
- standard notations defined by popular texts on analysis or design methodology.

Why does the term *architecture* have so many different meanings? Why do architectural representations vary so much in both their value and the approach to construction they imply? The answer is simple: Arch-itectures vary according to a project's size, timing, and goals; its level of risk and innovation; and the number of people involved, their proximity, and their ability to communicate and resolve conflict.

However, understanding the reasons for diverse definitions is not enough; there are other issues that must be explored if we are to understand architecture.

Varying views. When used with objects, the architecture drives their development. But architecture is more that just a high-level design of how objects might be used. The architecture includes complementary architectural viewpoints that can illuminate nuances of the problem at hand or explain concepts that are factored into the design of numerous objects.

To illustrate this point, let's examine a subtle architecture issue that surfaced in a bond-trading application. In this application, there are examples of abstract objects for bonds, trades, dates, and payments, and two kinds of users: traders and bookkeepers. The users measure time in whole days and recognize these days as periods in which securities are traded and bonds accrue interest.

However, as Figure 1 shows, traders start and end their intervals at the close of the trading day, before interest has accrued, whereas bookkeepers start and end their intervals as trading opens and include the accrued interest. The application is further complicated because most bonds accrue interest uniformly, making the time distinction undetectable. Still, if the odd bond is to fit into the architecture, the timing

distinction must be made at all levels of the program.

In this example, understanding how dates are interpreted is key to mastering the system. The interpretation is not the responsibility of a single object, but is a concept that drives the design decisions for a variety of objects. Thus, that dates are interpreted from multiple viewpoints is an architectural viewpoint of a bond-trading application.

An architectural viewpoint is often application-specific and varies widely based on the application domain. In reviewing our own software design experiences, we have seen architectural viewpoints that address a variety of issues, including

- temporal issues,
- state and control approaches,
- data representation,
- transaction life cycle,
- security safeguards, and
- peak demand and graceful degradation.

No doubt there are many more possible viewpoints. Every system has a set of architectural viewpoints that must be resolved before a design can be developed. These viewpoints, mapped onto the solution, drive the design decisions that create systems of objects. Figure 2 shows a simple mapping between architecture and objects.

View from the top. To make sense of the multiple architectural views of a single system, we looked to the architect's role in the centuries-old field of building design. An after-the-fact study of a great architect's work often talks of the architect's *style*. But it is not style that leads great architects to design buildings a certain way—it is their internal philosophy.

If you study a great architect such as Frank Lloyd Wright, you discover that his architectures did not come from just random ideas in his head, but from blending customer requirements with deep convictions about how a building should be made. These convictions are about how people should interact with buildings, how buildings relate to their surroundings, how materials should be used, and so on. Wright believed buildings should fit into their surroundings, such as the rectilinear ranch house he designed to fit within the flat plains of the Southwest. These convictions combine to form a philosophy—a personal philosophy for a great architect.

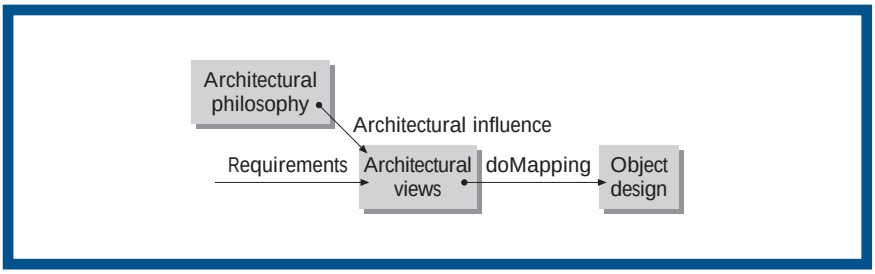

Figure 3. *An architect's personal philosophy determines the mapping between architecture and objects.*

In the software field, there are also great architects whose work is worthy of study. But we don't know many of these people because software architectures are rarely written about, discussed, or appreciated. As a profession, we do not seek out and learn from our great architects, who often work in obscurity. The architects for Apple Lisa, MacApp, and Hypercard, for example, developed fine systems that greatly advanced our field, but few people know their names: Eric Harslem, Larry Tesler, and Bill Atkinson, respectively.

As the box on page 58 explains and Figure 3 shows, architectural philosophy determines software system design. If we are to discover, learn from, and emulate the work of our best architects, we must not only recognize them, but seek to understand their philosophy as well.

ENTER PATTERN LANGUAGE

In the software development context, a *pattern* is an important and recurring system construct and a *pattern language* is a system of patterns

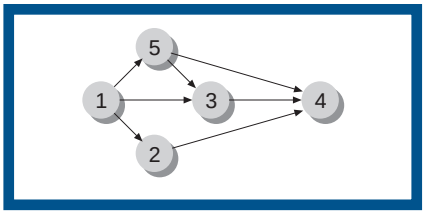

Figure 4*. A simple pattern language for human-interface design. Satisfying Pattern 1 allows the next three patterns, which in turn make Pattern 4 possible.*

organized in a structure that guides the patterns' application. A pattern language can be represented by a directed graph or network of patterns.

Although a software architectural philosophy is ethereal and hard to explain, a pattern language can embody the philosophy in a form that can be written, discussed, and evaluated. Patterns assembled into a pattern language order our thought processes, much as a natural language's grammar orders our sentence structure. The combination of patterns and their ordering makes a system rich enough to carry a philosophy.

Beginnings. In the mid-1980s, Ward Cunningham and Kent Beck created a simple pattern language to help users communicate with system developers about the type of human interface they wanted in their system. Five patterns were identified by number and were used to guide the human-interface design for a Smalltalk Model-View-

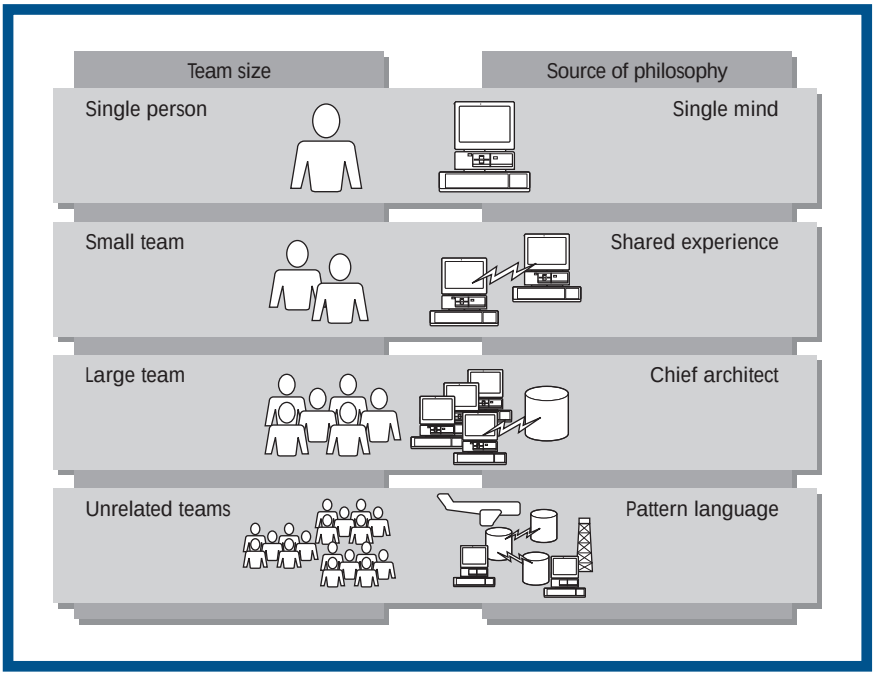

Figure 5. *The relationship between project size and project philosophy.*

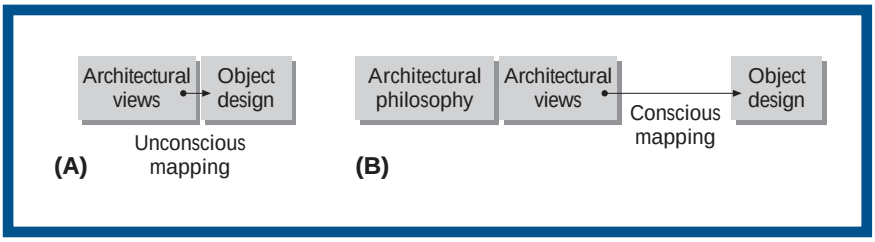

Figure 6. *On a single-person project [A], the mapping from architectural view to object design is unconscious. With a small team [B], the mapping becomes overt.*

Controller environment:

1. *Window per task*: Each task a user needs to perform is handled by a different window.
2. *Tiled panes*: Logical subtasks are handled within their own pane. Users can switch between panes.
3. *Pallet of panes*: Only a few classes of simple panes can be used. The pane classes are text, list, table, and waveform.
4. *Short menus per pane*: Users can invoke only a few commands while in a particular pane. Hierarchical or scrolling menus are not allowed.
5. *Nouns in lists, verbs in menus*: This differentiates between a command and an object being operated upon.

Figure 4 shows the five patterns. Pattern 1 (window per task) is a prerequisite to patterns 5, 3, and 2. That is, satisfying Pattern 1 makes satisfying the next three patterns possible. These in turn make Pattern 4 (short menus) possible. Although these patterns don't explicitly cite a philosophy, they are consistent with and even embody the particular human-interface philosophy the Smalltalk components were meant to realize.

Although most real-life systems would use many more than five patterns, even this simple pattern language effectively resolves much discussion about what a system can and cannot do. It also moves the design power from developer to user, and assures that the user designs a workable system.

Patterns today. The notion that there are patterns of objects that recur in programs and are worthy of study is now a very important topic in our field. The germ of this idea stems from the recognition that Christopher Alexander's work has parallels in software development. As with many new ideas in our field, many bright people have seized the idea and tried to apply it in different ways.

Fundamental to all patterns investigation is the attempt to recognize recurring situations in design so we can learn from other people's experience. The processes that investigators use to find patterns varies widely. We will consider three general categories.

- The *introspective* approach is when people reflect on the systems that they have built and find patterns relating to their experience. This approach can be described as a search for individual architectural style.
- The *artifactual* approach studies systems built by different teams working on similar problems. The pattern investigator is not involved with system development and seeks a more objective perspective. This approach can be described as a study of the software artifacts.
- The *sociological* approach studies the people building similar systems to discover the recurring problems in the systems and in developer interactions. This technique can best be described as an investigation through interview.

Patterns thinking is new; so, too, is patterns investigation. No doubt there are other approaches, and some researchers are using a combination of methods.

PHILOSOPHY AND PROJECT SIZE

As Figure 5 shows, the architectural philosophy of a software project becomes more concrete as the project size increases. In a single-person project, you unconsciously map the architectural view to the object design, as Figure 6a shows. The philosophy is internal and ethereal, and a natural way to solve problems; it is best described by this statement: "All I have to do is get my mind around the problem and the objects will pop out."

A one-person system has little need for an expressed architecture. The way the system will meet the architectural goals is kept in the designer's head and deployed automatically, often without

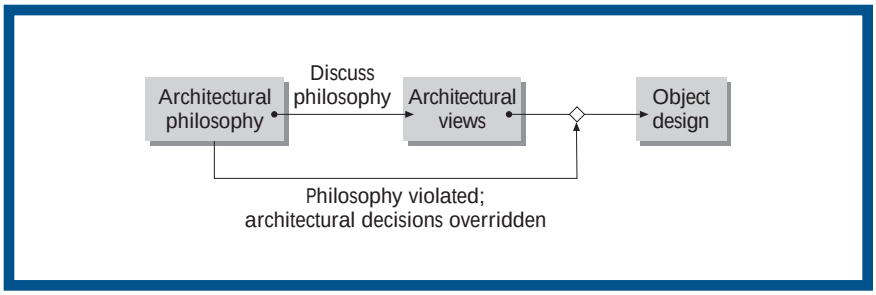

Figure 7. *As the size of the project grows, so too does the complexity of mapping between architecture and object.*

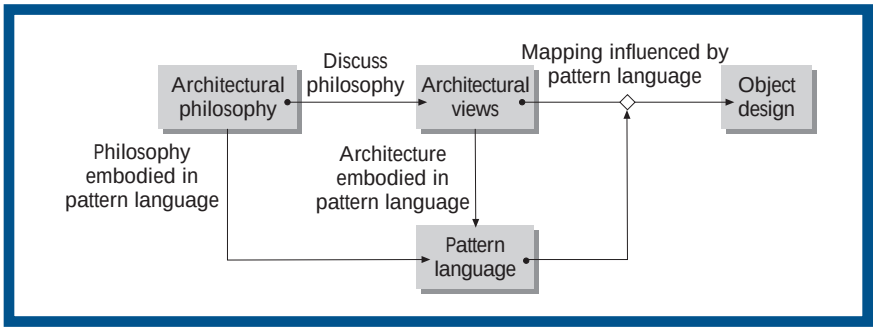

Figure 8. *A system's architecture and the philosophy underlying it are embodied in a pattern language.*

awareness. But when we move to a small team, ideas must be expressed to others and thus the mapping of architectural view to object design must be overt, as Figure 6b shows. This expression is usually informal, through discussion, storytelling, ad hoc pictures, and refinement. The architectural philosophy remains in the minds of team members and is assumed but not named.

As the team grows beyond a few people, we must make the architecture explicit and formalize and articulate the philosophy driving the architecture. As Figure 7 shows, our mapping model is now more complicated.

On large teams, a chief architect develops and communicates the architectural philosophy. Typically, the architect articulates the philosophy in an ad hoc fashion and focuses on documenting the architecture. The mapping between architectural views and object design must be quite disciplined or the architecture might be improperly implemented or simply inadequate. The result of a failed architecture can be a system that doesn't work or one that is buggy, fragile, and difficult to maintain. Patterns are useful in this environment because they suggest common solutions to common design problems.

When we look at unrelated teams working in different locations, the need for conscious application of the philosophy is clear. (Think, for example, of all the Macintosh developers building software according to Apple's philosophy.) Somehow, the ethereal architectural philosophy must be shared among the different groups, along with a careful and detailed description of the architecture.

Because the architecture generally evolves as the design progresses and as various groups raise questions that impact other groups, you must make good decisions about object design to minimize rework and unnecessary change. Although using objects contributes to the possibility of rework and change, a pattern language can significantly minimize the chaos that results when many people work independently on a single system.

A pattern language embodies both the philosophy and the architecture, as Figure 8 shows. It acts as a bridge between the architecture and the object design, and assures that the philosophy is communicated, taught, and followed. What was once ad hoc and informal is now explicit, but without the developer constraints a design standard can impose.

TOWARD THE FUTURE

In *The Timeless Way of Building*,[1] Alexander applies a disciplined approach to studying patterns. For our purposes, there are three significant components of his work.

♦ *Problem solving.* Alexander developed a directed graph of patterns—which he called a pattern language—to help solve large problems such as the design and construction of buildings and towns. The idea evolved from his strong mathematical foundation and a mastery of systems theory combined with his belief that the human brain follows certain paths to understand patterns, just as it follows certain paths to understand natural language.[5] Alexander's approach reduces a seemingly massive problem into several smaller problems connected by a grammar. This grammar allows developers to intentionally ignore aspects of the large problem early in the design, knowing that they will be addressed at the appropriate time.

♦ *Philosophical base.* Alexander develops a process that includes identifying, accepting, valuing, and discarding particular patterns in his pattern language. His architectural philosophy is embodied in this process. Although the philosophy is ethereal and abstract—as we might expect—it is consistent and provides the basis for developing concrete pattern languages.

♦ *Application.* Alexander applies his pattern language to real-life projects, such as building a cafe in Vienna,[6] developing a master plan for the University of Oregon,[7] and creating a town in Baja, California.[8] He has reported on several of these projects in a series of books. In them, he discusses what works and, just as importantly, what does not.

Present tense. Despite appearances, little of Alexander's work has been mirrored in the software community. We have used the word "pattern" and, in some instances, decided what must be captured when documenting a pattern.

A PHILOSOPHY CREATES A SYSTEM

The early work of Apple Macintosh designers shows the influence of architectural philosophy on a software system. The architecture for Macintosh applications stands apart from the requirements, but influences every good application designed for the machine. The system's architectural elements include

- the notion of users in control of their computers (event-driven programming),
- users freely moving snippets of information (cut and paste), and
- users sharing a network (distributed desktops).

But the philosophy is more than the sum of these architectural elements.

The philosophy behind the Mac design predates Apple. "Computers augmenting the human intellect" was first articulated by Douglas Engelbart, who credits Vannavar Bush and his imaginary device, the Memex, as his inspiration.[1, 2] The work of these innovators was initially very difficult to understand. They were often discussing concepts that were yet to be implemented and the practicality of their work was not obvious.

Nevertheless, their philosophy slowly grew in several laboratories until it was finally used in a product by the Macintosh developers. For the most part, the developers' adoption of this philosophy involved ethereal discussions of what the machine could do for people and how people would work with the machine. It was not a precise doctrine that could be written. Rather, it was the personal acceptance of values that defined the philosophy.

This philosophy motivated most of the discussion in *Inside Macintosh*.[3] It was embodied in the MacApp framework and was the foundation for Apple's evangelism work. But because of its ethereal nature, the philosophy was never successfully documented and was often missed by uninitiated developers. Even Macintosh users and software developers who understood the Macintosh system couldn't explain the philosophy. Telling a new programmer that the "computers must augment the human intellect" provided no tanglible guidelines for development. Nevertheless, the philosophy existed and was clearly apparent in the product. As Alexander predicted with his quality-without-a-name, once the philosophy was mastered, it could not be explained.

REFERENCES

1. J. Norton and R.W. Watson, "The Augment Knowledge Workshop," *National Computer Conf. Proc.*, American Federation of Information Processing, June 1973, pp. 9-21.
2. D. Engelbart, "As We May Think," *Atlantic Magazine*, July 1945.
3. C. Rose with B. Hacker, et al., *Inside Macintosh*, Addison-Wesley, Reading, Mass., 1985.

However, the most popular pattern texts have been merely collections of isolated patterns. If you use these works, you will not be guided by a grammar—you have to simply study the patterns and hope you recognize one that applies to your project.

In our own work with pattern languages, we have begun to embody philosophical ideas.[9,10] However, these ideas are a side effect of introspection rather than something we intended. Basically, we allowed our strong internal philosophies to guide our discovery process, not unlike taking a position in a philosophical debate. Had we consciously set out to understand our philosophies, we may have found a deeper set of patterns.

A great architect has a deep, consistent philosophy that is likely to grow and change over time. But each software developer need not have an individual philosophy. Developers can learn the philosophy of a master architect and choose to emulate it. In fact, as we see from the building profession, most good young architects will strive to emulate a master, and over time and with experience will develop their own philosophy.

Future perfect? To effectively build large object systems that realize a philosophy, we must use a pattern language. First, however, we must develop effective pattern languages. Using Alexander's work as a guide, we can formulate a methodical approach:

- Develop a consistent philosophy for building systems.
- Construct a pattern language that reflects this philosophy. We must then assure that the patterns in the language are effectively discovered, evaluated, and woven into a system of patterns in such a way that later design decisions do not force us to rework earlier ones; and earlier design decisions do not limit decisions later in the process.
- Carefully test the pattern language and report our findings in an objective manner.

Currently there are people exploring and finding software patterns using the three different approaches to patterns investigation we described earlier. Based on our own work, we've formulated recommendations for each approach so it can best benefit future designers.

- *Introspective.* Although their work embodies a philosophy, introspective explorers work without awareness of its role. They become aware of philosophical aspects when a pattern violates a philosophical value; the questions they then ask can lead to the discovery of new patterns. In the future, Alexander's third component—reporting on real-life projects and noting successes and failures—must be addressed at this level. Introspectively derived pattern languages are a valuable resource for forward-looking architectural solutions. We need independent evaluation and feedback to help acknowledge their value.
- *Artifactual.* Although the artifactual researchers most closely parallel Alexander's work, they only look at project artifacts and draw conclusions. They do not work from an architectural philosophy. In the future, these researchers must work on articulating philosophies and developing an evaluation process.
- *Sociological.* This research area has been tapped only slightly.[11] We need to develop skills that support investigation through interviewing. At this point, we have only our natural listening and observation skills. Cultural anthropologists know a great deal about how to study a cultural system.

We must learn their skills and experiment with applying them to the pattern-discovery process.

To date, the study of patterns has been limited to a small community of pattern-language developers and a few academics. Participants in a recent OOPSLA session expressed concern that because industry work on patterns draws heavily on practical experience, pattern language use would increase the distance between industrial and academic researchers. However, a closer look suggests something quite different.

Pattern language anthropology and archeology are disciplines waiting to be developed. The professionals who pioneer these fields will chart the methodologies of software architecture development and bring to light the genius and mastery that have remained too long in obscurity. They will discover new material and structures that enable us to understand, discuss, and teach the philosophies, architectural elements, and pattern languages of our finest practitioners. Their success—and the ultimate importance and sophistication of this new information—will depend upon academia and industry working together more closely than ever before. ◆

REFERENCES

1. C. Alexander, *The Timeless Way of Building*, Oxford Univ. Press, New York, 1979. p. 39.
2. M. Boasson, "The Artistry of Software Architecture," *IEEE Software*, Nov. 1995, pp. 13-16.
3. M. Moriconi, X. Qian, and R.A. Riemenschneider, "Correct Architecture Refinement," *Trans. Software Eng.*, Apr. 1995, pp. 356-372.
4. D. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM Software Eng. Notes*, Oct. 1992, pp. 40-52.
5. S. Pinker, *The Language Instinct*, William Morrow, New York, 1994.
6 C. Alexander, *The Linz Cafe/Das Lins Cafe*, Oxford Univ. Press, New York, 1981.
7. C. Alexander, *The Oregon Experiment*, Oxford Univ. Press, New York, 1975.
8. C. Alexander with Howard Davis et al., *The Production of Houses*, Oxford Univ. Press, New York, 1985.
9. N.L. Kerth, "Caterpillar's Fate: A Pattern Language for the Transformation from Analysis to Design," *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schmidt, eds., Addison-Wesley, Reading, Mass., 1995, pp. 293-324; also see http://c2.com/ppr/catsfate.html.
10. W. Cunningham, "The Checks Pattern Language for Information Integrity," *Pattern Languages Program Design*, J.O. Coplien and D.C. Schmidt, eds., Addison-Wesley, Reading, Mass., 1995, pp. 145-156; also see http://c2.com/ppr/checks.html.
11. J.O. Coplien, "A Generative Development-Process Pattern Language," *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schmidt, eds., Addison-Wesley, Reading, Mass., 1995, pp.183-238.

Norm Kerth is a consultant working with companies to ensure they make a successful transition to using object-oriented technologies, postmorta, and pattern languages. He also consults on specification and design activities, quality assurance, continuous process improvement, project management, and effective team building. He is particularly interested in building large, distributed object-oriented systems and growing system architects. Kerth has been studying pattern languages for more than a decade, authored the Caterpillar's Fate pattern language, and served as co-editor of *Pattern Languages of Program Design 2*. Prior to starting his company, Elite Systems, Kerth was a professor and researcher at the University of Portland.

He is a member of the IEEE Computer Society and ACM.

Ward Cunningham is cofounder of Cunningham & Cunningham. He has served as a principal in the IBM Consulting Group and as director of R&D at Wyatt Software, where he designed the WyCash portfolio management system. He also developed the Foundation graphical framework while with Knowledge Systems. As a principal engineer at Tektronix Computer Research Lab, he did research in object-oriented programming, computer-aided design, and human-computer interfaces. He has studied and authored pattern languages for more than a decade and served as program chair of the Pattern Languages of Programs conference.

Cunningham is a member of the ACM and the American Association for the Advancement of Science (AAAS).

Address questions about this article to Kerth at Elite Systems, P.O. Box 2205, Beaverton, OR 97075; nkerth@teleport.com; or Cunningham at Cunningham & Cunningham, 7830 SW 40th Avenue, Suite 4, Portland, OR 97219; ward@c2.com.