Past, Present, and Future Trends in Software Patterns

Frank Buschmann, *Siemens*
Kevlin Henney, *Curbralan*
Douglas C. Schmidt, *Vanderbilt University*

Software patterns influence how developers design and implement computing systems. Examining software patterns' past, present, and future trends can help developers improve their projects.

For more than a decade, patterns have influenced how software architects and developers create computing systems. Design-focused patterns provide a vocabulary for expressing architectural visions and clear, concise representative designs and detailed implementations. Presenting software pieces in terms of their constituent patterns also lets developers communicate more effectively, with greater conciseness and less ambiguity.

Since the mid-'90s, many software systems—including major parts of the Java and .NET libraries and many middleware platforms—have been developed with the conscious awareness of patterns. Sometimes developers applied these patterns selectively to address specific challenges and problems. Other times, they used patterns holistically to help construct software systems, from initially defining baseline architectures to finally realizing fine-grained details. Knowledge and conscious application of patterns has become a valuable commodity for software professionals.

Much has changed since Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides ("the Gang of Four" or "GoF") published *Design Patterns*, the most popular book on patterns.[1] The technology landscape has shifted, software design approaches have evolved and expanded, our understanding of development processes has matured, and we know more about documenting and applying patterns to software development. Ironically, *Design Patterns* is still so popular and influential that many software developers are unaware how much the field has matured or where to find pattern publications that cover a broader range of domains and technologies.

The pattern community has long aimed to document and promote good software engineering practices. This article summarizes the breadth and depth of the patterns in practice to help software developers and managers understand where the field has been and where it's headed, so that you can use patterns in your own projects.

A brief history of software patterns

Patterns haven't always been as popular or pervasive as they are now. Although the late '80s and early '90s saw isolated research on software patterns, patterns didn't enter the mainstream

0740-7459/07/$25.00 © 2007 IEEE

Patterns are generally gregarious, in that they form relationships with other patterns.

until *Design Patterns* was published. The book described 23 patterns derived largely from the authors' experiences developing single-threaded, object-oriented, user interface frameworks in Smalltalk and C++.

Design Patterns remains the most popular and influential pattern work. Numerous books and articles have addressed GoF patterns in various ways. Some publications discuss implementing the patterns in other programming languages, such as C# and Java. Other publications rework and integrate GoF patterns for specific application contexts, such as distributed computing,[2–4] security,[5,6] and real-time embedded systems.[7–9]

At the 1999 OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications) conference, James Burke noted that history rarely happens in the right order or at the right time—the historian's job is to make it appear as if it did. Likewise, although we summarize key trends from two decades of patterns research, our list is far from exhaustive. For more comprehensive references, see http://hillside.net/patterns.

Stand-alone patterns and pattern collections

Inspired by the GoF's success, much of the popular patterns research in the mid-to-late '90s focused on *stand-alone patterns* and *pattern collections*. Stand-alone patterns are "point solutions," which address relatively bounded problems that arise in specific contexts. Examples of popular stand-alone patterns include these:

- Iterator offers aggregate traversal without exposing aggregate representation details to callers.
- Strategy captures pluggable behavior.
- Wrapper Facade encapsulates an existing procedural API's functions and data within more concise, robust, portable, and cohesive object-oriented interfaces.

Any significant software design inevitably includes many patterns, however, which makes stand-alone patterns unusual in practice. A pattern collection is the most obvious presentation of multiple patterns. The most ambitious pattern collection to date is Grady Booch's ongoing work on the *Handbook of Software Architecture* (www.booch.com/architecture), which references approximately 2,000 patterns. Most pattern collections are much more modest in size and ambition, often numbering tens of patterns and focusing on a particular kind of problem, context, or system.

Many stand-alone patterns were initially created for—and honed in—writers' workshops at various Pattern Language of Programming (PLoP) conferences. In a writers' workshop, authors read each others' patterns and discuss their strengths and weaknesses to help improve content and style. You can find the most mature patterns from these conferences in Addison-Wesley's *Pattern Languages of Program Design* books or in the online PLoP conference proceedings.

Pattern relationships

Patterns that represent the foci for discussion, point solutions, or localized design ideas can be used in isolation with some success. Patterns are generally gregarious, however, in that they form relationships with other patterns. Any given application or library will thus use many related patterns. The most common types of pattern relationships include complements, compounds, and sequences.

Pattern complements. In these, one pattern provides either the missing ingredient for another pattern or an alternative solution to a related problem. Cooperative complements aim to make the resulting design more complete and balanced. For example, Disposal Method complements Factory Method by addressing object destruction and creation in the same design. This complementary combination supports designs that encapsulate resource life-cycle policies, such as pooling. Patterns can also compete with each other. For example, Batch Method, an alternative to Iterator, can access an aggregate object's elements without exposing its underlying implementation. Batch Method is more suitable for distributed environments (where remote access makes using a conventional Iterator cost-prohibitive) because it accesses an aggregate's elements in bulk, reducing round-trip network costs.

Pattern compounds. These capture recurring subcommunities of patterns that are common and identifiable enough that software developers can treat them as a single decision in response to a recurring problem. For example, applying two patterns together, such as a Command implemented as a Composite, is so

common that you can name the patterns as a single entity, such as Composite Command. Another example is Batch Iterator, which joins two complementary patterns, Iterator and Batch Method, to remotely access the elements of aggregates with large numbers of elements. A Batch Iterator refines an Iterator's position-based traversal with a Batch Method for accessing many, but not all, elements.

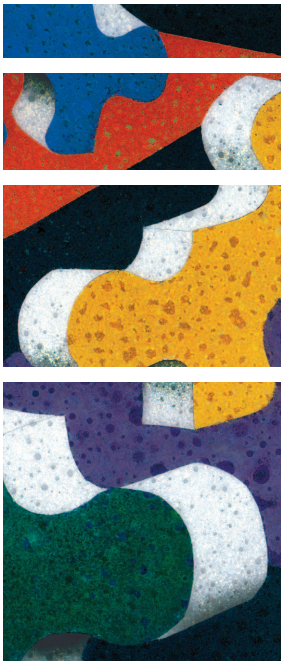
Pattern sequences. These generalize the progression of patterns and show how to establish a design by joining predecessor patterns to form part of each successive pattern's context. For example, in our previous work, we presented a communication-middleware pattern sequence that joins the Broker, Layers, Wrapper Facade, Reactor, Acceptor-Connector, Half-Sync/Half-Async, Monitor Object, Strategy, Abstract Factory, and Component Configurator patterns.[2] A pattern sequence captures how a design or situation unfolds pattern by pattern. You can also illustrate such a progression with a *pattern story*, which is a concrete example of applying a pattern sequence.

Pattern languages

As early as 1995, leading authors in the pattern community began documenting groups of patterns for specific software development domains, particularly telecommunications systems. As the first *Pattern Languages of Program Design* books showed, these patterns were more closely related than the earlier stand-alone patterns and pattern collections. In fact, some were so closely related that they didn't exist in isolation, so authors organized them as *pattern languages* in which each pattern built on and wove together other patterns in the language. We can view pattern languages as the logical extrapolation of the pattern relationships described previously. Likewise, we view pattern relationships as constrained and simplified aspects and subsets of pattern languages.

Pattern languages aim to provide holistic support for using patterns to develop software for specific technical or application domains, such as e-commerce or communication middleware. Accordingly, they enlist multiple patterns for each potential problem and weave them together to define a generative, domain-specific, and pattern-oriented software development process.

Many pattern languages are elaborations and decompositions of stand-alone patterns and pattern sequences. It's only natural that these languages build on—and recursively unfold and strengthen—the core qualities of their constituent patterns. For an example, see *Remoting Patterns*,[3] which decomposes the Broker pattern[10] into a fully fledged description of modern communication middleware architectures. Broker couldn't be that expressive when it was described as a stand-alone pattern, but the Remoting pattern language revealed this generativity.



Domains and technologies documented by patterns

From the start, patterns and frameworks have been intimately connected, demonstrated by the many framework examples *Design Patterns* used to motivate patterns. A framework is an integrated set of components that collaborate to provide a commoditized software architecture for a family of related applications. Mature frameworks exhibit high pattern density, making patterns an ideal descriptive tool for developing, evolving, and understanding frameworks. Beyond the interest in frameworks, the following domains and technologies are popular foci for pattern authors.

Distributed computing. Our previous work presents an extensive pattern language for building distributed software systems.[2] The pattern language connects more than 250 patterns that address topics ranging from defining and selecting an appropriate baseline architecture and communication infrastructure to specifying component interfaces and their implementations and interactions. The pattern language also addresses key technical aspects such as adaptation and extension, concurrency, database access, event handling, synchronization, and resource management.

Language- and domain-specific idioms. Several programming styles have evolved and emerged over the last decade, including aspect-oriented software development, domain-driven design, model-driven software development, and generative programming. Each has its own patterns and best practices that distinguish programming in it from programming in other styles. Patterns and pattern collections exist for these styles. Programming-language idioms have been a common focus for published patterns, from Smalltalk to Python and from C++ to C#.

The patterns community has met its goal of documenting and promoting good software engineering practices.

Fault tolerance and management. As we increasingly integrate software into mission- and safety-critical systems, we need robust techniques to meet user dependability requirements. Fault tolerance and management patterns have therefore been an active focus for the past decade. Several recent books contain patterns and pattern languages that address fault tolerance and fault management for systems with stringent operational requirements.[4,7]

Security. This has become a popular topic for patterns—and software systems in general. For example, *Security Patterns* documents a range of patterns and pattern languages for security-related areas such as authentication, authorization, integrity, and confidentiality.[5] Many of those patterns were documented in earlier pattern publications such as PLoP conference proceedings. Another recent text, *Core Security Patterns*, covers networked software security.[6]

Embedded systems. Increasingly, applications such as pacemakers, power-plant controllers, and flight-critical avionics systems embed intelligence in physical devices and systems. Because these applications are inextricably connected to the physical environment, developers must design them to satisfy physical demands and limitations—such as dynamics, noise, power consumption, and physical size—in a timely manner. Research in this area documents patterns for addressing these constraints without losing the benefits of abstraction.[7,8]

Process and organizational structure. Much pattern language research has focused on improving software development processes and organizations. Some recent books address certain types of software development processes, such as distributed, agile, test-driven, and domain-driven software development, as well as software refactoring and reengineering.[11,12] *Organizational Patterns of Agile Software Development* integrates existing patterns on software development processes and organizations into an interconnected pattern language.[13]

Education. Patterns and pattern languages are popular vehicles for learning the art of teaching programming and software engineering. A PLoPD book compiles several publications in this area,[14] as does the Pedagogical Patterns Project (www.pedagogicalpatterns.org), which provides a forum for disseminating and discussing teaching patterns, with a focus on teaching effective software practice.

Where patterns are now

In one sense, the patterns community has met its goal of documenting and promoting good software engineering practices. After more than a decade of experience in mining, documenting, and applying patterns, the community has established patterns in mainstream software development practice. Many production software projects and university curricula consciously use patterns. Moreover, there's much more experience with the format used to document patterns, such as a deeper appreciation of the importance of capturing the forces that shape designs,[15] compared with the format used in *Design Patterns*. We expect these trends to continue as developers increasingly understand the core patterns and the wealth of patterns and pattern languages in the broader literature.

Although many software systems have succeeded by intentionally applying patterns, failures have also occurred due to common misunderstandings about patterns—including what they are and are not to their properties, purpose, target audience, benefits, and drawbacks. The pattern community has long sought to understand the underlying theories, forms, and methodologies of patterns and pattern languages (and their associated concepts) to help codify knowledge about and effective application of software patterns. The largest work in this area, *Pattern-Oriented Software Architecture Volume 5*, integrates many facets of the pattern concept into a coherent whole.[15]

Compared to the initial wave of pattern users after *Design Patterns* was published, software developers seem to better understand patterns than before, in terms of their experience using patterns effectively on software projects and their understanding of various aspects of the pattern concept. The published patterns and pattern languages have also generally increased in quality. Most patterns and pattern languages published in the past several years are more expressive, comprehensive, precise, and readable than those published earlier.

Where patterns may go

Many software developers are now familiar with patterns and applying them in their domains, begging the question, "What is left to

say about patterns?" Since 1996, we've been observing trends that shape the pattern literature and community, as well as forecasting the future of patterns as part of our work on the *Pattern-Oriented Software Architecture* books. So, we've made conjectures on future trends given the benefit of hindsight and our experiences discovering, documenting, and applying patterns in practice.

We expect to see software developers document more technology- and domain-specific patterns and pattern languages. Software development's technologies and domains aren't all addressed by patterns yet—and it might take decades to cover them all. In particular, patterns capture experience, and for newer domains and technologies, developers must first gain, evaluate, and codify this experience before they can document effective patterns. Here, we predict which domains and technologies developers will address first.

Service-oriented architecture

SOA is a style of organizing and using distributed capabilities that different organizations or groups control and own. SOA isn't a new concept, but it has become a buzzword in recent years. The approach builds on principles and technologies from distributed computing and enterprise system integration, drawing from a spectrum of existing patterns and pattern languages.[16,17] However, some SOA technologies—such as business process modeling, service orchestration, and ultra-large-scale systems—are still unexplored and not covered by the pattern literature.

Generative software technologies

Most patterns documented since the early '90s have been mined from object-oriented software written in third-generation programming languages. Patterns and pattern languages, however, have already started to influence and support other software development approaches, particularly aspect-oriented and model-driven software development. We measure an approach's maturity and acceptance in terms of the degree to which authors publish its patterns and best practices. Developers have tried to capture model-driven software development patterns associated with process and organization, domain modeling, tool architecture, and application platform development. We expect that patterns and pattern languages will have the same influence on aspect-oriented software development.

Distributed real-time and embedded systems

Developing high-quality distributed real-time and embedded systems is hard—harder in fact than developing traditional real-time and embedded systems—and somewhat of a "black art." A forthcoming book documents patterns for developing distributed real-time and embedded software based on material from pattern-writing workshops.[9] We expect to see more patterns for such systems because progress in this domain is essential for developing viable mission- and safety-critical applications.

Quality of service for COTS-based distributed systems

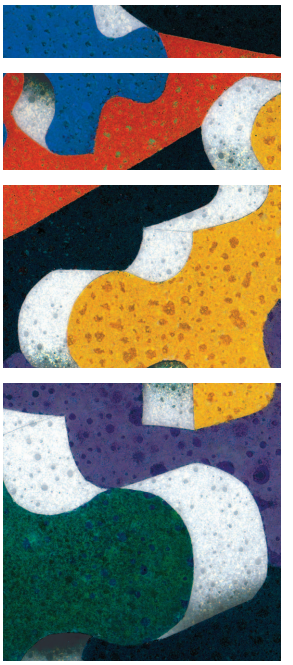
To reduce development cycle time and cost, projects are increasingly developing distributed systems using multiple layers of COTS hardware, operating systems, and middleware components. It's hard, however, to configure COTS-based systems that can simultaneously satisfy multiple QoS properties, such as security, timeliness, and fault tolerance. As developers and integrators continue to master the complexities of providing end-to-end QoS guarantees, we expect to see an increase in documented patterns that help others configure, monitor, and control COTS-based distributed systems that possess a range of interdependent QoS properties.

Mobile systems

Wireless networks have become pervasive, and embedded computing devices are becoming ever smaller, lighter, and more capable. Likewise, you can now access Internet services, from Web browsing to online banking, from mobile systems. Mobile systems present many challenges, however, such as managing low and variable bandwidth and power, adapting to frequent disruptions in connectivity and service quality, diverging protocols, and maintaining cache consistency across disconnected network nodes. We expect that experienced mobile-systems developers will document their expertise in pattern form to help meet the growing demand for best practices in this area.

Software architecture

Despite an increase in documented pattern languages, the software industry has no parallel to other design disciplines' comprehensive



Given the growing demand for electronic collaboration support, we expect to see more patterns in HCI in the near future.

handbooks. Although the existing patterns literature has steadily progressed toward creating handbooks for software engineers, the final goal hasn't been reached. As we mentioned earlier, the *Handbook of Software Architecture* contains thousands of patterns, exposing their essential roles and relationships and allowing comparisons across domains and architectural styles.

Group interaction

Although many developers have published human-computer interaction patterns in recent years, we don't have an integrated view of these patterns yet. Moreover, patterns don't yet cover all areas of group interaction in an electronic environment—for example, virtual worlds and massively multiplayer games. Given the growing demand for electronic collaboration support, we expect to see more patterns and pattern languages in HCI in the near future.

Web 2.0

The Web increasingly provides the context for more dynamic and open business models, where harnessing collective intelligence becomes the means of business and the next-generation Web (that is, Web 2.0) becomes the medium. Tim O'Reilly's initial set of Web 2.0 patterns (www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html) captures some key emerging and proven practices.

Business transaction and e-commerce systems

Many business information systems—such as those for accounting, payroll, inventory, and billing—are based on transactions. The rules for processing transactions are complex and must be flexible to reflect new business practices and mergers. Business systems must also handle increasingly large volumes of transactions online. The meteoric growth of e-commerce on the Web has exposed many business-to-business systems directly to consumers. Despite these systems' importance, relatively little has been written about their robust and secure analysis, architecture, or patterns.

Process and organizational structure

The growing adoption of agile development processes suggests that the corresponding pattern literature will continue to grow. Some work will focus on macroprocess aspects, such as overall life-cycle and business interaction, and some will focus on microprocess aspects, such as test-driven development, refactoring, and tool use.

The Gang of Four

Many books and articles address the GoF patterns, with no end in sight. Authors will continue meeting the demand for ruminations on the GoF work—from essays on missing ingredients, alternative solutions, and the patterns' proper scope to discussions about whether a certain pattern is outdated or isn't a pattern at all. There's also talk of a second edition of *Design Patterns*, although a publication date hasn't been finalized.

Pattern theory

Work on the theory underlying pattern concepts will continue, focusing on better understanding the known facets, such as pattern sequences, and exploring new views, such as problem frames (which capture, describe, and name recurring types of problems). Although there's still much scope for consolidating, clarifying, and communicating theoretical concepts, we believe that the most interesting trends will be associated with the domains in which patterns are documented, rather than in theory.

Certainly, authors will publish patterns and pattern languages for areas other than those we've mentioned—we just consider these topics the most promising. Naturally, our predictions include a healthy dose of uncertainty and are based on our knowledge of the pattern community, its ongoing and planned interests and research activities (insofar as we're aware of them), and the future directions we're involved with.

References

1. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994.
2. F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture Vol. 4: A Pattern Language for Distributed Computing*, John Wiley & Sons, 2007.
3. M. Voelter, M. Kircher, and U. Zdun, *Remoting Patterns*, John Wiley & Sons, 2004.
4. G. Utas, *Robust Communications Software: Extreme Availability, Reliability, and Scalability for Carrier-Grade Systems*, John Wiley & Sons, 2005.
5. M. Schumacher et al., *Security Patterns: Integrating Security and Systems Engineering*, John Wiley & Sons, 2006.
6. C. Steel et al., *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2006.
7. M.J. Pont, *Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051*

Family of Microcontrollers, Addison-Wesley, 2001.
8. J. Noble and C. Weir, *Small Memory Software: Patterns for Systems with Limited Memory*, Addison-Wesley, 2000.
9. L. DiPippo and C.D. Gill, *Design Patterns for Distributed Real-Time Embedded Systems*, Springer, to be published in 2007.
10. F. Buschman et al., *Pattern-Oriented Software Architecture Vol. 1: A System of Patterns*, John Wiley & Sons, 1996.
11. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2004.
12. G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007.
13. J. Coplien and N. Harrison, *Organizational Patterns of Agile Software Development*, Prentice Hall, 2005.
14. D. Manolescu, M. Voelter, and J. Noble, *Pattern Languages of Program Design 5*, Addison-Wesley, 2006.
15. F. Buschmann, K. Henney, and D.C. Schmidt, *Pattern-Oriented Software Architecture Vol. 5: On Patterns and Pattern Languages*, John Wiley & Sons, 2007.
16. M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
17. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, 2004.

About the Authors

Frank Buschmann is a principal engineer at Siemens Corporate Technology in Munich. His research interests include object technology, software platforms and product lines, application frameworks, and specifically patterns. He received a diploma in computer science from the Universität Dortmund, Germany. He's coauthored four volumes in Wiley's *Pattern-Oriented Software Architecture* book series. Contact him at Siemens AG, Corporate Technology, Software and Eng., CT SE 2, Otto-Hahn Ring 6, 81739 Munich, Germany; frank.buschmann@siemens.com.

Kevlin Henney is a UK-based independent consultant and trainer for telecommunications, information services, logistics, and finance organizations. His research interests include programming languages, object orientation, patterns, software architecture, and the development process. He received his MSc in parallel computer systems from the University of the West of England (formerly Bristol Polytechnic). He's coauthored two volumes in Wiley's *Pattern-Oriented Software Architecture* book series. He's a member of the ACM. Contact him at Curbralan Ltd., 17 Tyne Rd., Bishopston, Bristol, BS7 8EE, UK; kevlin@curbralan.com.

Douglas C. Schmidt is a computer science professor and the associate chair of Vanderbilt University's Computer Science and Engineering program. He's also chief technology officer for PrismTech and a visiting scientist at the Software Engineering Institute. He received his PhD in computer science from the University of California, Irvine. He's coauthored three volumes of Wiley's *Pattern-Oriented Software Architecture* book series. He's a member of the IEEE. Contact him at the Inst. for Software Integrated Systems, 2015 Terrace Pl., Vanderbilt Univ., Nashville, TN 37064; d.schmidt@vanderbilt.edu.