PATTERN-ORIENTED SOFTWARE ARCHITECTURE

On Patterns and Pattern Languages

Volume 5

Frank Buschmann,
Siemens, Munich, Germany

Kevlin Henney,
Curbralan, Bristol, UK

Douglas C. Schmidt,
Vanderbilt University, Tennessee, USA

John Wiley & Sons, Ltd

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

On Patterns and Pattern Languages

Volume 5

Frank Buschmann,
Siemens, Munich, Germany

Kevlin Henney,
Curbralan, Bristol, UK

Douglas C. Schmidt,
Vanderbilt University, Tennessee, USA

John Wiley & Sons, Ltd

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wiley.com

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada M9W 1L1

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN-13: 978-0-471-48648-0

Typeset in 10/13 Bookman-Light by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire
This book is printed on acid-free paper responsibly manufactured from sustainable forestry
in which at least two trees are planted for each one used for paper production.

For Anna, Bebé†, and Martina

Frank Buschmann

For Carolyn, Stefan, and Yannick

Kevlin Henney

For Lori, Bronson, Mom, and Dad

Douglas C. Schmidt

For John

Frank Buschmann, Kevlin Henney, Douglas C. Schmidt

† Bebé, July 3, 1999

Table of Contents

Foreword by Richard P. Gabriel

†'Software patterns have significantly changed the way we design…' is how POSA5 starts out—its preface, its self-explanation, maybe its justification. But design: what happens when we design? Is design about problems or about beauty? Does resolving forces while solving a problem force beauty into the design? Or can beauty and ideal problem solving emerge only after a (pattern) language has been tooled over the raw material? Someone once told me that any establishment where the entrance is obvious is not worth visiting.

The Oxford English Dictionary, second edition, informs us about design as follows:

> 1. A mental plan.
>
> 1.a. a plan or scheme conceived in the mind and intended for subsequent execution;…the preliminary conception of an idea that is to be carried into effect by action; a project;

In Guy Steele's 1998 talk at OOPSLA, 'Growing a Language,' he said:

> A design is a plan for how to build a thing. To design is to build a thing in one's mind but not yet in the real world—or, better yet, to plan how the real thing can be built.

I once wrote:

> Design is the thinking one does before building.

Carliss Baldwin and Kim B. Clark define design like this:

> Designs are the instructions based on knowledge that turn resources into things that people use and value.
>
> *—Between 'Knowledge' and 'the Economy': Notes on the Scientific Study of Designs*

Each of these definitions and characterizations revolve around ideas, plans, knowledge, problems, values and purposes, and hint at a before-building time. Very rational-sounding and the way we would like design to go—reliable, the trusty well-educated mind

working things out: you can feel safe walking across the real thing because it will not sway and fall down. But let's think a little about who *we*† is.

People have been 'designing and building' cities for millennia, using, perhaps, tools and concepts like Christopher Alexander's pattern languages. Such artifacts are huge and hugely complex. It's hard to say where the mental plan is, because the imagining that might take place is surely scattered across the years, decades, centuries, and sometimes millennia it takes to fashion a proper city—a city like Athens, Rome, Berlin, London, Istanbul, or even New York or San Francisco and surely Boston. Thus the '*we*' in the up-noted sentence† must refer to a more recent set of *us*-es—namely software designers, who, one could argue, have gravitated toward the more conservative or possibly less realistic ideas of design reflected in the quotes above. Even in my quote—where I was trying to leave room for the interweaving of design, building, and reflection that cities, for example, enjoy as part of their design journey—it is hard to not see the before-building thrust into the forefront: *the thinking one does before building*.

That design has a landscape of meanings and nuances brings to mind a set of characterizations of design by Adrian Thompson at the University of Sussex. I call these characterizations *design metaheuristics*, meaning they are approaches to how to think about design, like the approach of using patterns and pattern languages. There are three design metaheuristics. Here's the first:

> **Inverse model is tractable**: If there is a tractable 'inverse model' of the system, then there is a way of working out in advance a sequence of variations that brings about a desired set of objective values.

Here a design is a plan, like a blueprint, and the inverse model is this: once the thing to be designed is imagined, there is a way to work out the blueprint or a plan for how to build the thing. This is the essential up-front design situation, which works best when designing something that has been built at least once before, and typically dozens or thousands of times. Alexander knows of this style of working: the ideas behind patterns and pattern languages evolved from the notion that people who built, for example, Swiss barns knew Swiss-barn

patterns and pattern languages naturally—as a result of seeing them every day, working in them, and helping build them as children. In this sense the inverse model is well established: the image in the head is not unlike the barn next door, and the steps to build it are known with perhaps only modest modification. A pattern language (implicit or explicit) can guide the designer/builder along, and the little adjustments that are always needed are just building as usual. This leads to Thompson's second design metaheuristic:

> **Inverse model is not tractable, but forward model is**: In this case, we can predict the influence of variations upon the objective values, but the system is not tractably invertible so we cannot derive in advance a sequence of variations to bring about a desired set of objective values. This implies an iterative approach, where variations carefully selected according to the forward model are applied in sequence. This kind of iterative design-and-test is a common component of traditional approaches.

Design a little, build a little, reflect a little—this is how I explain this heuristic. It is the essence behind the agile methodologies, I think, evolutionary design/programming, and Alexander's 'Fundamental Process.' From the Agile Manifesto:

> Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
>
> Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to [sic] the shorter timescale.
>
> …
>
> The best architectures, requirements, and designs emerge from self-organizing teams.
>
> —http://agilemanifesto.org/principles.html

Alexander's 'Fundamental Process' is how he describes designing and building in *The Nature of Order*. In this massive four-volume book, he talks about how **centers**—places of interest and attention in space—are structured to create **wholeness**. Here are steps 1, 2, 4, 6, and 8:

> 1. At every step of the process—whether conceiving, designing, making, maintaining, or repairing—we must always be concerned with the whole within which we are making anything. We look at this wholeness, absorb it, try to feel its deep structure.

> 2. We ask which kind of thing we can do next that will do the most to give this wholeness the most positive increase of life.
>
> 4. As we work to enhance this new living center, we do it in such a way as also to create or intensify (by the same action) the life of some larger center.
>
> 6. We check to see if what we have done has truly increased the life and feeling of the whole. If the feeling of the whole has not been deepened by the step we have just taken, we wipe it out. Otherwise we go on.
>
> 8. We stop altogether when there is no further step we can take that intensifies the feeling of the whole.

Pattern languages are used like this, both by Alexander and by software developers. Such usage reflects, merely, the realization that the first design metaheuristic applies only to cases in which designs are being (mostly) repeated. In such cases, there are often models that can be used to simulate the design-a-little / build-a-little pattern the 'Fundamental Process' describes. Either there is literally a model—physics and mathematics and science—that is the boss of the design, and pretend designs can be tried out, or else we fool ourselves into thinking that large steps in the fundamental process—releases of version 1, then version 2, then version 3—are each actually full-blown designs. We are fooled because we don't want to zoom out to the larger scale. Alexander's 'Fundamental Process' is also a good description of the various agile methodologies, although one would expect refactoring to be more explicitly mentioned.

The book you either are about to read, are reading, or have finished reading—the one in which this foreword is embedded—is about pattern languages largely in this context. *The thinking one does before building* can be described by steps 1, 2, & 4. In this context, design really is a sort of problem solving. As Jim Coplien says:

> Design is a process of intentional acts to the end of moving from some conception of a problem to the reduction of that problem. A problem is the difference between a current state and desired state.

This seems to miss two aspects that seem (to some) central to design: beauty and newness. A very fine wine glass is both functional and beautiful, it is made from an exquisite and exotic material, its

shape—each type of wine has a preferred wineglass shape to concentrate the aroma best, to direct the wine to the best part of the mouth for its flavor components, to prevent too much warming from the hands—presents the wine to its best effect, while also pleasing the eye with its elegance and line. Instead of 'newness' I almost wrote 'novelty,' but 'novelty' implies an intellectual or academic newness that misses the point. It isn't that the design is an invention full of brand new workings, it's the pleasure and surprise of seeing something you've never seen before. I'm thinking about the Cord Model 810 automobile introduced in 1936:

> The body design of the Cord 810 was the work of designer Gordon M. Buehrig and his team of stylists that included young Vince Gardner. The new car caused a sensation at the 1936 New York Auto Show in November. The crowds around the 810 were so dense, attendees stood on the bumpers of nearby cars to get a look. Many orders were taken at the show…
>
> —Wikipedia

People had never seen a car like this before—coffin-shaped nose, headlights hidden, exhaustpipes (one per cylinder) chromesnakes jutting out the sides, the smooth curved shelf at the front—it even was designed by stylists not designers. Notable about both its appearance, engineering, and user experience were the recessed headlights and the fact that the transmission came out the front of the engine, so that the floor of the passenger compartment could be lower and flatter than was usual at the time. Some say the design of the Cord 810 remains the most distinctive of the twentieth century. In 1996, American Heritage magazine proclaimed the Cord 810 sedan 'The Single Most Beautiful American Car.'

For example.

I find it hard to say that the Cord is the result of the reduction of a problem to a final outcome: something else is happening. Rebecca Rikner names this 'something else':

> Design is not about solving a problem. Design is about seeing the beauty.

Many would guess I'm done with Thompson's metaheuristics, but there's a third. And it's the third which is the topic of his research.

> **Neither forward nor inverse models are tractable**: There is neither a way of discerning which variations will give improvements in the objective values, nor a way of predicting what will be the effects of variations upon the objective values. Without evolution all is lost.
>
> —*Notes on Design Through Artificial Evolution: Opportunities and Algorithms*

Evolution!

Thompson's work includes the design of an analog circuit that can discriminate between 1kHz and 10kHz square waves using a field-programmable gate array designed using a genetic algorithm.[1] The result is described as 'probably the most bizarre, mysterious, and unconventional unconstrained evolved circuit yet reported,' in large part because the researchers were unable, in the end, to understand how the circuit works. It's not important to understand the details of the following remarks from their paper—just take a look at the expressions of incredulity:

> Yet somehow, within 200ns of the end of the pulse, the circuit 'knows' how long it was, despite being completely inactive during it.
>
> This is hard to believe, so we have reinforced this finding through many separate types of observation, and all agree that the circuit is inactive during the pulse.

Research in genetic algorithms, genetic programming, neuro-evolution, neural nets, and statistical/aleatoric methods all demonstrate surprising results—some of them displaying a puzzling illusion of design, like the antenna designed and tested by NASA and designed by artificial evolution. The size of an infant's hand, the elements bend off in a crazy cryptosymmetry like a small black bush half-frozen after a frost. It doesn't look like something a person would design, but it has better antenna-crucial characteristics in some ways, and so it is a good design—in terms of *reducing a problem*.

---

1. Adrian Thompson et al, *Explorations in Design Space: Unconventional Electronics Design Through Artificial Evolution*.

But do we *feel* that it is a good design? What do we make of this evolution stuff? How does artificial evolution work? In short, a population is subjected to artificial reproduction in which some characteristics from one parent are combined with others from another. The resulting characteristics are subjected to a small amount of random mutation, and the resulting population—which can include the original parents in some algorithms—is tested for fitness against a fitness function. Assuming the characteristics are related to aspects of design, this reduces to taking some design elements from one not-so-bad solution and combining them with design elements from another not-so-bad solution, making some small changes, and seeing how that goes. Aside from the random part, this doesn't sound so random.

If you take it from me that some of the things 'designed' this way have interesting characteristics, we need to ask whether this approach is a form of design. Some people would say that without a human designer, there can be no design, that the 'intentional' in Coplien's characterization and the 'mind' in the OED definition are essential features. Where are the 'instructions based on knowledge'? If Pierre Menard[2]

---

2. *Pierre Menard, Author of The Quixote*, by Jorge Luis Borges. In this story, a twentieth century writer (Menard) puts himself in a frame of mind to rewrite, word for word, part of Cervantes' *Don Quixote de la Mancha*, but not as a transcription, but as a coincidence. Enjoy:

> *It is a revelation to compare Menard's Don Quixote with Cervantes'. The latter, for example, wrote (part one, chapter nine):*
>
> > *…truth, whose mother is history, rival of time, depository of deeds, witness of the past, exemplar and adviser to the present, and the future's counselor.*
>
> *Written in the seventeenth century, written by the 'lay genius' Cervantes, this enumeration is a mere rhetorical praise of history.*
>
> *Menard, on the other hand, writes:*
>
> > *….truth, whose mother is history, rival of time, depository of deeds, witness of the past, exemplar and adviser to the present, and the future's counselor.*
>
> *History, the mother of truth: the idea is astounding. Menard, a contemporary of William James, does not define history as an inquiry into reality but as its origin. Historical truth, for him, is not what has happened, it is what we judge to have happened. The final phrases—exemplar and adviser to the present, and the future's counselor—are brazenly pragmatic.*
>
> *The contrast in style is also vivid. The archaic style of Menard—quite foreign, after all—suffers from a certain affectation. Not so that of his forerunner, who handles with ease the current Spanish of his time.*

were to dream up the FPGA tone discriminator or the weird antenna, such doubters might declare the result a design—or perhaps further, human-sensible explanations of the design are required to make that leap.

This brings to my mind the idea of canalization, described like this:

> A developmental outcome is canalized when the developmental process is bound to produce a particular end-state despite environmental fluctuations both in the development's initial state and during the course of development.
>
> —(adapted from) Andre Ariew, *Innateness is Canalization: In Defense of a Developmental Account of Innateness*

A thing produced is canalized when once it falls into a tributary of a river, by the process of its production it ends up in the river. 'Canalized' comes from the word 'canal'—once in the canal, you cannot escape. Strange attractors in complexity science are canalizations. Artificial evolution doesn't care whether a design—or at least a thing it produces—is beautiful, elegant, understandable, parsimonious, maintainable, extendable, sleek, slick, simple, superfluous, silly, or human. All it cares about is that it fits what the fitness function dictates. The design is not canalized by those things—those human things—as just about all human designs are and most likely must be.

Facing it in another direction, the design of, say, the antenna looks strange because it is not what a person would ever design. We are incapable of exploring a full design space, only those parts that look pretty to us.

This book is about design viewed through the lens of patterns and pattern languages. As you read it, permit your mind to drift closer to ideas of design and perhaps later on, further away. Permit yourself to wonder about design in all its glories and incarnations. The three design metaheuristics might very well represent points on a coherent continuum, and if they do, there also is a point where the human becomes non-human or, maybe worse, post-human.

Pattern languages can encompass lots of design space. They are part of our software design and construction toolboxes. They work by helping people design better. Reflection on good design is how pattern languages are written down. Design, design, design, design.

Richard P. Gabriel

Foreword by Wayne Cool

Way back I fell in with a bunch hot on the trail of a revolution in programming, hefting on their backs the unwieldy and orthodoxless ideas of a mad architect himself bent on revving the future by reviving the past in the splendor of its design sense[1] but not its design [Ale79]. This cool group holed up in retreats and self-concocted workshops, never in a mainstream meet-up but in contrast to official miracles, in a far-off locale, some backwater or Podunk which although unverifiable is visitable.[2] Unlike the madman who crowed 'beauty beauty beauty,' this crew worked the nuts, oiled the bolts, screwed together a practice and a program, a practicality and a precision straight-aimed at the practitioner and around the corner from theory.

Along the way I hopped from PLoP to PLoP, café to bar, beer to Starbucks; I was there when Beck proclaimed, when the workshops wound up, when the rainstorm first came up on the parquet Allerton floor, when the sun rose at Irsee, when the trail rides ended and the dust settled in Wickenberg; I was there when the books were written, reviewed, printed, and praised. Through all this I watched the interest grow in design and the structure of man-made things, in the strength of the written word aimed at the creation of a built-up world made of ideas and abstractions but felt like the built world of wood and shellac, stone and metal rods.

And this is what I want to come to: the realm of software patterns converts the world of surfaces—a world where mechanism is hidden beneath impenetrable covers—to one of exposed innards and plain working; the hidden becomes the object of a craftsmanship that can be defined as the desire to do something well—for its own sake, for no reward, for the pride of skill applied with discernment and in reflection, taking a long time and going deeply into it. For those who build

1. Or '*scents,*' as the agile would proclaim.

2. Apologies to Bill Knott.

this way, it matters little that the craft is buried under a user interface—because one day someone must peek underneath and see. The surfaces matter but they don't hide.

Art. Craft. Engineering. Science. These are the swirling muses of design patterns. Art and science are stories; craft and engineering are actions.

Craft is midway between art and science; art and craft stand over against engineering and science. Art is the unique example, the first thing, the story as artifact condensing out of talent and desire. Craft is reliable production of quality. A craftsman[3] might be disappointed but rarely fails. A work of craft is the product of a person and materials. Engineering is reliable and efficient production of things for the use and convenience of people. Science is a process of making a story that can be used for engineering. A book called 'The Art of X' is about the mystery of the rare individual who can make an X. 'The Craft of X' is about the sweat and training endured by those who make Xs one at a time. 'The Engineering of X' is about discipline and long days planning and making Xs for institutional customers. 'The Science of X' is a maths book.

But the roles of science and craft have been misdiagnosed. Most believe (deeply) that science precedes, as they believe it must, craft and engineering; that building requires abstract knowing. As in: to design a steam engine you need a (correct) theory of thermodynamics. But when the steam engine was developed, scientists believed the 'caloric theory:' that there is a 'subtle fluid' called *caloric* that is the substance of heat. The quantity of this substance is fixed in the universe; it flows from warmer bodies to colder. Good story. The caloric theory explained a lot, but the mechanics who built the first steam engines didn't know the theory or didn't care about it. Working with boilers, they noticed relations between volume, pressure, and temperature and they built steam engines. Maybe the scientists even used observations of steam engines to come up with 'modern' thermodynamics.

3. Today I choose the pretty word.

Craftsmen know the ways of the substances they use. They watch. Perception and systematic thinking combine to formulate understanding. A well-developed craft gives rise to technological developments. And science. Sure: there are feedback loops between science and engineering, but neither is queen—and hands-on dominates sit-and-think.

This is about building, and all building requires planning and execution. Planning is the thinking one does before building. Execution is understanding the plan and producing something from it. Since the early twentieth century management science has pushed for the separation of planning from execution. Frederick Winslow Taylor said it with art and poetry this way:

> All possible brain work should be removed from the shop and centered in the planning or lay-out department.
>
> —Principles of Scientific Management

Sweet.

Planning and execution merge in art but stand wide apart in engineering; somewhere in between for craft. Science is in the realm of thought and ideas.

Separation of planning from execution is not about efficiency. Not about getting the most value per hour. Maybe it will; maybe it won't. Planning takes thinking; thinking costs money. If you can think a little, build a lot, you can make money. Thinking while executing is replaced by process; education is replaced by training—the way Dancing Links and Algorithm X^4 can solve Sudoku. With Dancing Links and Algorithm X, Sudoku needs a computer, not a person. Separating planning from execution is about cost.

And this is not about quality. The best quality cars are not built on blind robotic assembly lines. Cost.

If cost is pressure toward engineering, patterns is the push-back toward craft and art.

---

4. Dancing Links is a technique suggested by Donald Knuth to efficiently implement Algorithm X, a recursive, nondeterministic, depth-first, brute-force algorithm that finds all solutions to the exact cover problem.

Patterns. Craftsmanship: we think of it arising from the texture of the built world. A world where quality and attention to detail is visible on the surface of materials, in the gaps between things, in methods of joining without friction but with shape and convolution. A world where talent sits besides knowledge and intelligence. Read this description of the problem of making *felloes*—sections of rim on a wooden carriage wheel—in *The Wheelwright's Shop* by George Sturt:[5]

> Yet it is in vain to go into details at this point; for when the simple apparatus had all been gotten together for one simple-looking process, a never-ending series of variations was introduced by the material. What though two felloes might seem much alike when finished? It was the wheelwright himself who had to make them so. He it was who hewed out that resemblance from quite dissimilar blocks, for no two felloe-blocks were ever alike. Knots here, shakes[6] there, rind-galls[7], waney[8] edges, thicknesses, thinnesses, were for ever affording new chances or forbidding previous solutions, whereby a fresh problem confronted the workman's ingenuity every few minutes. He had no band-saw (as now [1923]) to drive, with ruthless unintelligence, through every resistance. The timber was far from being prey, a helpless victim, to a machine. Rather it would lend its own special virtues to the man who knew how to humour it.
>
> —The Wheelwright's Shop

You can feel the wood as equal partner to craftsman.

The work of patterns is the work of people who have systematic encounters with code, going deeply into it, dwelling for long periods of time on the tasks of design and coding to get them right, people who don't believe planning can be separated from execution. People who work this way are having the sort of encounter that gives rise to science. This is not mere trafficking in abstractions; it is thinking. Patterns don't push

5. First published in 1923.

6. A crack in timber caused by wind or frost.

7. A damage the tree received when young, so that the bark or rind grows in the inner substance of the tree.

8. A sharp or uneven edge on a board that is cut from a log not perfectly squared, or that is made in the process of squaring.

toward art and craft but are the tools of people who do. To be resisted is the automation of patterns which is the separation of planning and execution.

Computer scientists stand at the chalkboard—write and erase, squint and stare, remark and declaim. Patterns people, like mechanics, bend at the waist and peer into the code, their arms are drenched in code up to the elbows. It's what they work on: knots here, shakes there, rind-galls, waney edges, thicknesses, thinnesses forever affording new chances or forbidding previous solutions. They see programming almost as manual work—muscled arms, sleeves rolled tight against biceps, thought bright behind the eye linking mind and hand.

♦ ♦ ♦

I was there when the patterns community started. I sat the hillside; I huddled in the redwoods; I ran the beach; I hiked Mt Hood; I kayaked, did the ropes course, gazed on the Sun Singer at sunset; I sweated in the sauna; I rode the big horse, Bigfoot; I sang with the cowboy over steaks and corn. But I didn't add a single idea, not even one sentence, nor one dot of punctuation to what the patterns people thought, wrote, and built. I'll bet, though, you know exactly what I think about it.

Wayne Cool, Venice Beach

About the Authors

Frank Buschmann

Frank Buschmann is Senior Principal Engineer at Siemens Corporate Technology in Munich, Germany. His research interests include object technology, software architecture, product-lines, model-driven software development, and patterns. He has published widely in all these areas, most visibly in his co-authorship of three POSA volumes [POSA1] [POSA2] [POSA4]. Frank was a member of the ANSI C++ standardization committee X3J16 from 1992 to 1996, initiated the first EuroPLoP conference in 1996, co-edited several books on patterns [PLoPD3] [SFHBS06], and serves as an editor of the Wiley Series in Software Design Patterns. In his development work at Siemens, Frank has led architecture and implementation efforts for several large-scale industrial software projects, including business information, industrial automation, and telecommunication systems.

When not at work Frank spends most of his time enjoying life with his wife Martina and daughter Anna, having fun riding his horse Eddi, watching the time go by in Munich beer gardens, getting excited when watching his favorite soccer team, Borussia Dortmund, dreaming when listening to a performance at the Munich opera, and relaxing with rare Scotch single malts before bedtime.

Kevlin Henney

Kevlin Henney is an independent consultant based in Bristol, UK. His work involves teaching, mentoring, and practicing across his areas of interest, which include programming languages and techniques, software architecture, patterns, and agile development. His clients range from global firms to smaller startups, involved in the worlds of systems software, telecommunications, embedded systems, middleware development, business information, and finance.

Kevlin is a regular speaker at software conferences, and has also been involved with the organization of many conferences, including EuroPLoP. He has participated in the C++ standardization process, through the BSI and ISO, as well other language standardization efforts. Kevlin is also known for his writing, which has included a POSA volume [POSA4], conference papers, and regular (and irregular) columns for many publications, including *C++ Report*, *C/C++ Users Journal*, *Java Report*, *JavaSpektrum*, *Application Development Advisor*, *The Register*, *EXE*, and *Overload*.

In what passes for spare time, Kevlin enjoys spending time with Carolyn, his wife, and Stefan and Yannick, their two sons. This time takes in Lego, toy fixing, reading, and the odd beer or glass of wine.

Douglas C. Schmidt

Doug Schmidt is a Professor of Computer Science and Associate Chair of the Computer Science and Engineering program at Vanderbilt University, Nashville, Tennessee, USA. His research focuses on patterns and pattern languages, optimization principles, and empirical analysis of techniques that facilitate the development of quality of service (QoS)-enabled component middleware and model-driven engineering tools that support distributed real-time and embedded systems.

Doug is an internationally-recognized expert on patterns, object-oriented frameworks, real-time middleware, modeling tools, and open-source development. He has published over 300 papers in top technical journals and conferences, has co-authored books on patterns [POSA2] [POSA4] and C++ network programming [SH02] [SH03], and has also co-edited several popular books on patterns [PLoPD1] and frameworks [FJS99a] [FJS99b]. In addition to his academic research, Doug has led the development of ACE, TAO, CIAO, and CoSMIC, which are widely used open-source middleware frameworks and model-driven engineering tools that contain a rich set of reusable components implemented using the patterns presented in this book.

In his rare spare time Doug enjoys spending time with his wife Lori and their son Bronson, as well as weight-lifting, guitar playing, debating world history and politics, and driving Chevy Corvettes.

About this Book

Software patterns have significantly changed the way we design, implement, and think about computing systems. Patterns provide us with a vocabulary to express architectural visions, as well as examples of representative designs and detailed implementations that are clear and to the point. Presenting pieces of software in terms of their constituent patterns also allows us to communicate more effectively, with fewer words and less ambiguity.

Since the mid-1990s many software systems, including major parts of the Java and C# programming languages and libraries, were developed with the help of patterns. Sometimes these patterns were applied selectively to address specific challenges and problems. At other times they were used holistically to support the construction of software systems from the definition of their baseline architectures to the realization of their fine-grained details. Today the use of patterns has become a valuable commodity for software professionals.

Over the past decade and a half, a large body of literature has been created to document known patterns in a wide range of areas related to software development, including organization and process, application and technical domains, and best programming practices. This literature provides concrete guidance for practicing software engineers and increasingly influences the education of students. Each year new books and conference proceedings are published with yet more patterns, increasing the depth and breadth of software development knowledge codified in pattern form.

In the same way, the knowledge of, and experience with, *applying* patterns has also grown steadily, along with our knowledge about the pattern concept itself: its inherent properties, different flavors, and relationships with other technologies. In contrast to the ever-growing number of documented and refactored *concrete* patterns, however, publications *about* patterns and the pattern concept have been

updated only sparsely and in selected areas since the mid-1990s, despite the enormous increase of conceptual knowledge in the software patterns community. The introduction to software patterns in *A System of Patterns* [POSA1] and *Design Patterns* [GoF95], and the white paper on *Software Patterns* [Cope96], remain the most relevant sources about the pattern concept. Furthermore, only relatively recently have publications started to discuss and document pattern sequences explicitly [CoHa04] [PCW05] [Hen05b].

To summarize the current state of affairs, no complete and up-to-date work on the pattern concept is available. Moreover, knowledge of the latest advances in the conceptual foundations of patterns remains locked in the heads of a few experts and thought leaders in the patterns community. Mining this knowledge and documenting it for consumption by the broader software development community is the intention of this book, *On Patterns and Pattern Languages*, which is the fifth and final volume in the *Pattern-Oriented Software Architecture* series.

In this book we present, discuss, contrast, and relate the many known flavors and applications of the pattern concept: stand-alone patterns, pattern complements, pattern compounds, pattern stories, pattern sequences, and—last but not least—pattern languages. For each concept flavor we investigate its fundamental and advanced properties, and explore insights that are well-accepted by the pattern community, as well as perspectives that are still the subject of discussion and dispute. We also discuss how patterns support and interact with other technologies commonly used in software development. In a nutshell, we provide an overview of the current state of knowledge and practice in software patterns.

Note, however, that while we are general and broad regarding the elaboration and discussion of the pattern concept itself, the concrete examples we use to illustrate or motivate different aspects of the concept focus mostly on software design patterns—as opposed to other types of patterns such as organizational patterns, configuration-management patterns, and patterns for specific application domains. The reason for this (self-)restriction is twofold. First, the majority of all documented software patterns are software design patterns, so we have a wealth of material for our examples. Second, the largest group of software pattern users are architects and developers—thus our

focus on software design patterns allows us to explain the 'theory' behind patterns using practical examples with which this group is most familiar.

Intended Audience

The main audience of the book are software professionals interested in the conceptual foundations of patterns. Our primary goal is to help such professionals broaden, deepen, and complete their knowledge and understanding of the pattern concept so that they know what and how patterns can contribute to their projects. Our other goals are to help them avoid common misconceptions about patterns, and apply concrete patterns more effectively in their daily software development work.

This book is also suitable for undergraduate or graduate students who have a solid grasp of software engineering, programming languages, runtime environments, and tools. For this audience, the book can help them to learn more about what patterns are and how they can help with the design and implementation of high-quality software.

Structure and Content

The book is structured into three main parts, which are surrounded and supported by several smaller chapters that motivate and complete its vision and content.

Chapter 0, Beyond the Hype, reflects on the original definitions of the pattern concept and discusses how these definitions are received and understood by the software community. Our analysis suggests that some adjustments and enhancements are useful to avoid misconceptions when understanding patterns, and to help prevent the misapplication of patterns in software projects. This introductory chapter provides the foundation for the three main parts of the book, which elaborate and discuss these adjustments and enhancements in greater detail to provide a more complete and consistent picture of the pattern concept.

Part I, *Inside Patterns*, reflects on the use of stand-alone patterns, and presents and discusses the insights into patterns we have collectively gained over the last decade. These insights complement existing pattern definitions, helping us to understand patterns at a deeper level.

Part II, *Between Patterns*, moves outside individual patterns to explore the relationships between patterns: sometimes a set of patterns represent alternatives to one another, sometimes they are adjuncts to one another, and sometimes they are bound together as a tightly-knit group. Beyond the common, passive notion of a collection, this part of the book also considers how patterns can be organized as a sequence, with patterns applied one after another in a narrative flow, thereby adding an active voice to the use of patterns in the design process.

Part III, *Into Pattern Languages*, builds on the concepts and conclusions of the first two parts by introducing pattern languages. Compared with individual patterns and pattern sequences, pattern languages provide more holistic support for using patterns in the design and implementation of software for specific technical or application domains. They achieve this goal by enlisting multiple patterns for each problem that arises in their respective domains, weaving them together to define a generative and domain-specific software development process.

Chapter 14, From Patterns To People, picks up the discussion about the concept of patterns from the first three parts of the book, to conclude that despite all technology within patterns and the support they provide for other software technologies, the prime audience of patterns is people.

Chapter 15, The Past, Presence, and Future of Patterns, revisits our 2004 forecast on where we expected patterns to go that was published in the third volume of the *Pattern-Oriented Software Architecture* series. We discuss the directions that patterns have actually taken during the past three years and analyze where patterns and the patterns community are now. Based on this retrospection, we revise our vision about future research and the application of patterns and pattern languages.

This book is the last volume we plan to publish within the POSA series—at least for now. *Chapter 16, All Good Things…*, therefore wraps up and concludes our more than fifteen years of work on, and experience with, patterns, and examines the five volumes of the POSA series that we have written during this time.

The book ends with a summary of all the pattern concepts we discuss, a chapter with thumbnail descriptions of all the patterns we reference in the book, an extensive list of references to work in the field, a pattern index, a general subject index, and an index of names that lists everyone who helped us shape this book.

There are undoubtedly properties and aspects of the pattern concept that we have omitted, or which will emerge over time with even greater understanding of patterns and their use in practical software development. If you have comments, constructive criticism, or suggestions for improving the style and content of this book, please send them to us via e-mail to `siemens-patterns@cs.uiuc.edu`. Guidelines for subscription can be found on the patterns home page at `http://hillside.net/patterns/`. This link also provides an important source of information on many aspects of patterns, such as available and forthcoming books, conferences on patterns, papers on patterns, and so on.

Acknowledgments

It is a pleasure for us to thank the many people who supported us in creating this book, either by sharing their knowledge with us or by reviewing earlier drafts of its parts and providing useful feedback.

First and foremost, we want to thank John Vlissides, to whom we also dedicate this book. John was one of the most brilliant minds in the software patterns community—as ground-breaking thought leader and co-author of the legendary and seminal Gang-of-Four book [GoF95], and as 'discoverer' and mentor of many now well-known and world-class pattern experts. The inspirations and foundations from his work have significantly influenced and helped shape the pattern concept we elaborate and discuss in this book.

Champion review honors go to Wayne Cool, Richard P. Gabriel, Michael Kircher, James Noble, and Linda Rising, who reviewed all our material in depth, focusing on its correctness, completeness, consistency, and quality. Their feedback significantly increased the quality of material in the book. Wayne Cool also contributed many ideas and thoughts that we explore in depth in the book.

In addition, we presented parts of the material in the book at four EuroPLoP pattern conferences and also to several pattern experts. Alan O'Callaghan, Lise Hvatum, Allan Kelly, Doug Lea, Klaus Marquardt,

Tim O'Reilly, Michael Stal, Simon St. Laurent, Steve Vinoski, Markus Völter, Uwe Zdun, and Liping Zhao provided us with extensive feedback, which led to many minor and also some major revisions of various aspects of the pattern concept and their presentation.

Many thanks also go to Mai Skou Nielsen, who permitted us to use photos from her collection in the book.

Special thanks go to Lothar Borrmann and Reinhold Achatz for their managerial support and backing at the software engineering labs of Corporate Technology of Siemens AG, Munich, Germany.

Very special thanks go to our editor, Sally Tickner, our former editor Gaynor Redvers-Mutton, and everyone else at John Wiley & Sons who made it possible to publish this book. On a sunny evening at EuroPLoP 2002, Gaynor convinced us to write this POSA volume, and she also accompanied the first two years of its creation. Sally, in turn, had an enormous amount of patience with us during the two additional and unplanned years we spent completing the manuscript. Very special thanks also go to Steve Rickaby, of WordMongers Ltd, our copy editor, for enhancing our written material. Steve accompanied all five volumes of the POSA series with his advice and support.

Last, but not least, we thank our families for their patience and support during the writing of this book!

Guide to the Reader

The White Rabbit put on his spectacles.
"Where shall I begin, please your Majesty?" he asked.

"Begin at the beginning," the King said, very gravely,
"and go on till you come to the end: then stop."

Lewis Carroll, Alice's Adventures in Wonderland

This book is structured and written so that the most convenient way to read it is from cover to cover. If you know where you want to go, however, you can choose your own route through the book. In this case, the following hints can help you decide which topics to focus on and the order in which to read them.

A Short Story about Patterns

This book provides an in-depth exploration of the pattern concept. Starting with a popular—yet brief and incomplete—pattern definition, we first motivate, examine, and develop the inherent properties of stand-alone patterns. A solid understanding of what a stand-alone pattern is—and what it is not—helps when applying individual patterns effectively in software development.

We next explore the space 'between' patterns. Patterns are fond of company and can connect to one another through a variety of relationships: they can form alternatives to one another, or natural complements, or define a specific arrangement that is applied wholesale. Patterns can also line up in specific sequences that, when applied, generate and inform the architectures of concrete software systems. Knowing about and understanding the multifaceted relationships that can exist between patterns supports the effective use of a set of patterns in software development.

Finally, we enrich the concept of stand-alone patterns with the various forms of relationships between patterns, to elaborate the notion of pattern languages. Pattern languages weave a set of patterns together to define a generative software development process for designing software for specific applications or technical domain. Pattern languages realize the vision and goal of pattern-based software development that we had in mind when we started the *Pattern-Oriented Software Architecture* series over ten years ago.

All the concepts we explore and develop build on one another. The various types of relationships between patterns take advantage of the properties of stand-alone patterns. Pattern languages further build on and take advantage of the relationships between patterns. Starting with an informal and intuitive characterization of what a pattern is, we progressively mine and elaborate the different properties and facets of the pattern concept, until we have developed a more complete and consistent picture of what patterns are, what they are not, how they can support you when developing software, and how they relate to other software technologies and techniques.