REMOTING PATTERNS

Foundations of Enterprise, Internet and Realtime Distributed Object Middleware

Markus Völter
Michael Kircher
Uwe Zdun

WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Contents

Remoting Patterns

Remoting Patterns

Foundations of Enterprise, Internet and Realtime Distributed Object Middleware

Markus Völter, voelter - ingenieurbüro für softwaretechnologie, Heidenheim, Germany

Michael Kircher, Siemens AG Corporate Technology, Munich, Germany

Uwe Zdun, Vienna University of Economics and Business Administration, Vienna, Austria

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.wileyeurope.com or 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, 33 Park Road, 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.

Library of Congress Cataloging-in-Publication Data

Völter, Markus.
Remoting patterns: foundations of enterprise, internet and realtime distributed object middleware / Markus Völter, Michael Kircher, Uwe Zdun.
p. cm.
Includes bibliographical references and index.
ISBN 0-470-85662-9 (cloth : alk. paper)
1. Computer software—Development. 2. Software patterns. 3. Electronic data processing— Distributed processing. 4. Middleware. I. Kircher, Michael. II. Zdun, Uwe. III. Title.
QA76.76.D47V65 2004
005.1—dc22

2004018713

British Library Cataloguing in Publication Data

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

ISBN 0-470-85662-9

Foreword

Many of today's enterprise computing systems are powered by distributed object middleware. Such systems, which are common in industries such as telecommunications, finance, manufacturing, and government, often support applications that are critical to particular business operations. Because of this, distributed object middleware is often held to stringent performance, reliability, and availability requirements. Fortunately, modern approaches have no problem meeting or exceeding these requirements. Today, successful distributed object systems are essentially taken for granted.

There was a time, however, when making such claims about the possibilities of distributed objects would have met with disbelief and derision. In their early days, distributed object approaches were often viewed as mere academic fluff with no practical utility. Fortunately, the creators of visionary distributed objects systems such as Eden, Argus, Emerald, COMANDOS, and others were undeterred by such opinion. Despite the fact that the experimental distributed object systems of the 1980s were generally impractical – too big, too slow, or based on features available only from particular specialized platforms or programming languages – the exploration and experimentation required to put them together collectively paved the way for the practical distributed objects systems that followed.

The 1990s saw the rise of several commercially successful and popular distributed object approaches, notably the Common Object Request Broker Architecture (CORBA) promoted by the Object Management Group (OMG) and Microsoft's Common Object Model (COM). CORBA was specifically designed to address the inherent heterogeneity

of business computing networks, where mixtures of machine types, operating systems, programming languages, and application styles are the norm and must co-exist and cooperate. COM, on the other hand, was built specifically to support component-oriented applications running on the Microsoft Windows operating system.

Today, COM has been largely subsumed by its successor, .NET, while CORBA remains in wide use as a well-proven architecture for building and deploying significant enterprise-scale heterogeneous systems, as well as real-time and embedded systems.

As this book so lucidly explains, despite the fact that CORBA and COM were designed for fundamentally different purposes, they share a number of similarities. These similarities range from basic notions, including remote objects, client and server applications, proxies, marshalers, synchronous and asynchronous communications, and interface descriptions, to more advanced areas, including object identification and lookup, infrastructure extension, and lifecycle management. Not surprisingly, though, these similarities do not end at CORBA and COM. They can also be found in newer technologies and approaches, including .NET, the Java 2 Enterprise Edition (J2EE), and even in Web Services (which, strictly speaking, is not a pure distributed object technology, but nevertheless has inherited many of its characteristics).

Such similarities are of course better known as ‘patterns’. Patterns are generally not so much created as discovered, much as a miner finds a diamond or a gold nugget buried in the earth. Successful patterns result from the study of successful systems, and the remoting patterns presented here are no exception. Our authors, Markus, Michael, and Uwe, who are each well versed in both the theory and practice of distributed objects, have worked extensively with each of the technologies I’ve mentioned. Applying their

pattern-mining talents and efforts, they have captured for the rest of us the critical essence of a number of successful solutions and approaches found in a number of similar distributed objects technologies.

Given my own long history with CORBA, I am not surprised to find that several of the patterns that Markus, Michael, and Uwe document here are among my personal favorites. For example, topping my list is the Invocation Interceptor pattern, which I have found to be critical for creating distributed objects middleware that provides extensibility and modularity without sacrificing performance. Another favorite of mine is the Leasing pattern, which can be extremely effective for managing object lifecycles.

This book does not just describe a few remoting patterns, however. While many patterns books comprise only a loose collection of patterns, this book also provides a series of technology projections that tie the patterns directly back to the technologies that employ them. These projections clearly show how the patterns are used within .NET, CORBA, and Web Services, effectively recreating these architectures from the patterns mined from within them. With technology projections like these, it has never been easier to see the relationships and roles of different patterns with respect to each other within an entire architecture. These technology projections clearly link the patterns, which are already invaluable by themselves, into a comprehensive, harmonious, and rich distributed objects pattern language. In doing so, they conspicuously reveal the similarities among these different distributed object technologies. Indeed, we might have avoided the overzealous and tiresome ‘CORBA vs. COM’ arguments of the mid-1990s had we had these technology projections and patterns at the time.

Distributed objects technologies continue to evolve and grow. These patterns have essentially provided the building

blocks for the experimental systems of the 1980s, for the continued commercial success and wide deployment of distributed objects that began in the 1990s, and for today's Web Services integration approaches. Due to the never-ending march of technology, you can be sure that before too long new technologies will appear to displace Web Services. You can also be sure that the remoting patterns that Markus, Michael, and Uwe have so expertly provided for us here will be at the heart of those new technologies as well.

Steve Vinoski
Chief Engineer, Product Innovation
IONA Technologies
March 2004

Series Foreword

At first glance writing and publishing a remoting pattern language book might appear surprising. Who is its audience? From a naïve perspective, it could only be distributed object middleware developers – a rather small community. Application developers merely *use* such middleware – why should they bother with the details of how it is designed? We see confirmation of this view from the sales personnel and product 'blurbs' of middleware vendors: remote communication should be transparent to application developers, and it is the job of the middleware to deal with it. So why spend so much time on writing – and reading – a pattern language that only a few software developers actually need?

From a realistic perspective, however, the world looks rather different. Despite all advances in distributed object middleware, building distributed systems and applications is still a challenging, non-trivial task. This applies not only to application-specific concerns, such as how to split and distribute an application's functionality across a computer network. Surprisingly, many challenges in building distributed software relate to an appropriate use of the underlying middleware. I do not mean issues such as using APIs correctly, but fundamental concerns. For example, the type of communication between remote objects has a direct impact on the performance of the system, its scalability, its reliability, and so on and so forth. It has an even stronger impact on how remote objects must be designed, and how their functionality must be decomposed, to really benefit from a specific communication style.

It is therefore a myth to believe that remote communication is transparent to a distributed application. The many failures

and problems of software development projects that did so speak very clearly! Failures occur due to the misconception that ‘fire and forget’ invocations are reliable, that remote objects are always readily available at their clients’ fingertips, or problems due to a lack of awareness that message-based remote communication decouples operation invocation from operation execution not only in space, but also in time, and so on.

But how do I know what is ‘right’ for my distributed system? How do I know what the critical issues are in remote communication and what options exist to deal with them? How do I know what design guidelines I must follow in my application to be able to use a specific middleware or remote communication style correctly and effectively? The answer is simple: understanding both how it works, and why it works the way it works. Speaking pictorially, we must open the black box called ‘middleware’, sweeping the ‘shade’ of communication-transparency aside, and take a look inside. Fundamental concepts of remoting and modern distributed object middleware must be known to, and understood by, application developers if they are to build distributed systems that work! There is no way around this.

But how can we gain this important knowledge and understanding? Correct: by reading and digesting a pattern language that describes remoting, and mapping its concepts onto the middleware used in our own distributed systems! So in reality the audience for a remoting pattern language is quite large, as it comprises every developer of distributed software.

This book contributes to the understanding of distributed object middleware in two ways. First it presents a comprehensive pattern language that addresses all the important aspects in distributed object middleware – from remoting fundamentals, through object identification and lifecycle management, to advanced aspects such as

application-specific extensions and asynchronous communication. Second, and of immense value for practical work, this book provides three technology projections that illustrate how the patterns that make up the language are applied in popular object-oriented middleware technologies: .NET, Web Services, and CORBA. Together, these two parts form a powerful package that provides you with all the conceptual knowledge and various viewpoints necessary to understand and use modern communication environments correctly and effectively. This book thus complements and completes books that describe the ‘nuts and bolts’ – such as the APIs – of specific distributed object middlewares by adding the ‘big picture’ and architectural framework in which they live.

Accept what this book offers and explore the secrets of distributed object middleware. I am sure you will enjoy the journey as much as I did.

Frank Buschmann
Siemens AG, Corporate Technology

Preface

Today distributed object middleware belongs among the basic elements in the toolbox of software developer, designers, and architects who are developing distributed systems. Popular examples of such distributed object middleware systems are CORBA, Web Services, DCOM, Java RMI, and .NET Remoting. There are many other books that explain how a particular distributed object middleware works. If you just want to use one specific distributed object middleware, many of these books are highly valuable. However, as a professional software developer, designer, or architect working with distributed systems, you will also experience situations in which just understanding how to use one particular middleware is not enough. You are required to gain a deeper understanding of the inner workings of the middleware, so that you can customize or extend it to meet your needs. Or you might be forced to migrate your system to a new kind of middleware as a consequence of business requirements, or to integrate systems that use different middleware products.

This book is intended to help you in these and similar situations: it explains the inner workings of successful approaches and technologies in the field of distributed object middleware in a practical manner. To achieve this we use a pattern language that describes the essential building blocks of distributed object middleware, based on a number of compact, Alexandrian-style [AIS+77] patterns. We supplement the pattern language with three technology projections that explain how the patterns are realized in different real-world examples of distributed object middleware systems: .NET Remoting, Web Services, and CORBA.

How to read this book

This book is aimed primarily at software developers, designers, and architects who have at least a basic understanding of software development and design concepts.

For readers who are new to patterns, we introduce patterns and pattern languages to some extend in this section. Readers familiar with patterns might want to skip this. We also briefly explain the pattern form and the diagrams used in this book. You might find it useful to scan this information and use it as a reference when reading the later chapters of the book.

In the pattern chapters and the technology projections we assume some knowledge of distributed system development. In Chapter 1, *Introduction To Distributed Systems*, we introduce the basic terminology and concepts used in this book. Readers who are familiar with the terminology and concepts may skip that chapter. If you are completely new to this field, you might want to read a more detailed introduction such as Tanenbaum and van Steen's *Distributed Systems: Principles and Paradigms* [TS02].

For all readers, we recommend reading the pattern language chapters as a whole. This should give you a fairly good picture of how distributed object middleware systems work. When working with the pattern language, you can usually go directly to particular patterns of interest, and use the pattern relationships described in the pattern descriptions to find related patterns.

Details of the interactions between the patterns can be found at the end of each pattern chapter, depicted in a number of sequence diagrams. We have not included these interactions in the individual pattern descriptions for two reasons. First, it would make the pattern chapters less readable. Second, the patterns in each chapter have strong

interactions, so it makes sense to illustrate them with integrated examples, instead of scattering the examples across the individual pattern descriptions.

We recommend that you look closely at the sequence diagram examples, especially if you want to implement your own distributed object middleware system or extend an existing one. This will give you further insight into how the pattern language can be implemented. As the next step, you might want to read the technology projections to see a couple of well-established real-world examples of how the pattern language is implemented by vendors.

If you want to understand the commonalities and differences between some of the mainstream distributed object middleware systems, you should read the technology projections. You can do this in any order you prefer. They are completely independent of each other.

Goals of the book

Numerous projects use, extend, integrate, customize, and build distributed object middleware. The major goal of the pattern language in this book is to provide knowledge about the general, recurring architecture of successful distributed object middleware, as well as more concrete design and implementation strategies. You can benefit from reading and understanding this pattern language in several ways:

- If you want to *use* distributed object middleware, you will benefit from better understanding the concepts of your middleware implementation. This in turn helps you to make better use of the middleware. If you know how to use one middleware system and need to switch to another, understanding the patterns of distributed object middleware helps you to see the commonalities, in spite

of different remoting abstractions, terminologies, implementation language concepts, and so forth.

- Sometimes you need to *extend* the middleware with additional functionality. For example, suppose you are developing a Web Services application. Because Web Services are relatively new, your chosen Web Services framework might not implement specific security or transaction features that you need for your application. You must then implement these features on your own. Our patterns help you to find the best hooks for extending the Web Services framework. The patterns show you several alternative successful implementations of such extensions. The book also helps you to you find similar solutions in other middleware implementations, so that you avoid reinventing the wheel.
  Another typical extension is the introduction of ‘new’ remoting styles, implemented on top of existing middleware. Consider server-side component architectures, such as CORBA Components, COM+, or Enterprise Java Beans (EJB). These use distributed object middleware implementations as a foundation for remote communication [VSW02]. They extend the middleware with new concepts. Again, as a developer of a component architecture, you have to understand the patterns of the distributed object middleware, for example to integrate the lifecycle models of the components and remote objects.
- While distributed object middleware is used to integrate heterogeneous systems, you might encounter situations in which you need to *integrate* the various middleware systems themselves. Consider a situation in which your employer takes over another company that uses a different middleware product from that used in your company. You need to integrate the two middleware solutions to let the information systems of the two

companies work in concert. Our patterns can help you find integration points and identify promising solutions.
- In rarer cases you might need to *customize* distributed object middleware, or even *build* it from scratch. Consider for example an embedded system with tight constraints on memory consumption, performance, and real-time communication [Aut04]. If no suitable middleware product exists, or all available products turn out to be inappropriate and/or have a footprint that is to large, the developers must develop their own solution. As an alternative, you could look at existing open-source solutions and try to customize them for your needs. Here our patterns can help you to identify critical components of the middleware and assess the effort required in customizing them. If customizing an existing middleware does not seem to be feasible, you can use the patterns to build a new distributed object middleware for your application.

The list above consists of only a few examples. We hope they illustrate the broad variety of situations in which you might want to get a deeper understanding of distributed object middleware. As these situations occur repeatedly, we hope these examples illustrate why we think the time is ready for a book that explains such issues in a way that is accessible to practitioners.

About the authors

Markus Völter

Markus Völter works as an independent consultant on software technology and engineering based in Heidenheim, Germany. His primary focus is software architecture and patterns, middleware and model-driven software

development. Markus has consulted and coached in many different domains, such as banking, health care, e-business, telematics, astronomy, and automotive embedded systems, in projects ranging from 5 to 150 developers.

Markus is also a regular speaker at international conferences on software technology and object orientation. Among others, he has given talks and tutorials at ECOOP, OOPSLA, OOP, OT, JAOO and GPCE. Markus has published patterns at various PLoP conferences and writes articles for various magazines on topics that he finds interesting. He is also co-author of the book *Server Component Patterns*, which is - just like the book you are currently reading - part of the Wiley series in Software Design Patterns.

When not dealing with software, Markus enjoys cross-country flying in the skies over southern Germany in his glider.

Markus can be reached at voelter@acm.org or via www.voelter.de

Michael Kircher

Michael Kircher is working currently as Senior Software Engineer at Siemens AG Corporate Technology in Munich, Germany. His main fields of interest include distributed object computing, software architecture, patterns, agile methodologies, and management of knowledge workers in innovative environments. He has been involved in many projects as a consultant and developer within various Siemens business areas, building software for distributed systems. Among these were the development of software for UMTS base stations, toll systems, postal automation systems, and operation and maintenance software for industry and telecommunication systems.

In recent years Michael has published papers at numerous conferences on topics such as patterns, software

architecture for distributed systems, and eXtreme Programming, and has organized several workshops at conferences such as OOPSLA and EuroPLoP. He is also co-author of the book *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*.

In his spare time Michael likes to combine family life with enjoying nature, engaging in sports, or just watching wildlife.

Michael can be reached at michael@kircher-schwanninger.de or via www.kircher-schwanninger.de

Uwe Zdun

Uwe Zdun is working currently as an assistant professor in the Department of Information Systems at the Vienna University of Economics and Business Administration. He received his Doctoral degree from the University of Essen in 2002, where he worked from 1999 to 2002 as research assistant in the software specification group. His research interests include software patterns, scripting, object-orientation, software architecture, and Web engineering. Uwe has been involved as a consultant and developer in many software projects. He is author of a number of open-source software systems, including Extended Object Tcl (XOTcl), ActiWeb, Frag, and Leela, as well as many other open-source and industrial software systems.

In recent years he has published in numerous conferences and journals, and co-organized a number of workshops at conferences such as EuroPLoP, CHI, and OOPSLA.

He enjoys hiking, biking, pool, and guitar playing.

Uwe can be reached at zdun@acm.org or via wi.wu-wien.ac.at/~uzdun

Acknowledgments

A book such as this would be impossible without the support of many other people. For their support in discussing the contents of the book and for providing their feedback, we express our gratitude.

First of all, we want to thank our shepherd, Steve Vinoski, and the pattern series editor, Frank Buschmann. They have read the book several times and provided in-depth comments on technical content, as well as on the structure and coherence of the pattern language.

We also want to thank the following people who have provided comments on various versions of the manuscript, as well as on extracted papers that have been workshopped at VikingPLoP 2002 and EuroPLoP 2003: Mikio Aoyama, Steve Berczuk, Valter Cazzalo, Anniruddha Gokhale, Lars Grunske, Klaus Jank, Kevlin Henney, Wolfgang Herzner, Don Hinton, Klaus Marquardt, Jan Mendling, Roy Oberhauser, Joe Oberleitner, Juha Pärsinen, Michael Pont, Alexander Schmid, Kristijan Elof Sorenson (thanks for playing shepherd and proxy), Michael Stal, Mark Strembeck, Oliver Vogel, Johnny Willemsen, and Eberhard Wolff.

Finally we thank those that have been involved with the production of the book: our copy-editor Steve Rickaby and editors Gaynor Redvers-Mutton and Juliet Booker. It is a pleasure working with such proficient people.

Patterns and Pattern Languages

Over the past couple of years patterns have become part of the mainstream of software development. They appear in different types and forms.

The most popular patterns are those for software design, pioneered by the Gang-of-Four (GoF) book [GHJV95] and

continued by many other pattern authors. Design patterns can be applied very broadly, because they focus on everyday design problems. In addition to design patterns, the patterns community has created patterns for software architecture [BMR+96, SSRB00], analysis [Fow96], and even non-IT topics such as organizational or pedagogical patterns [Ped04, FV00]. There are many other kinds of patterns, and some are specific for a particular domain.

What is a Pattern?

A pattern, according to the original definition of Alexander[1] [AIS+77], is:

> *... a three-part rule, which expresses a relation between a certain context, a problem, and a solution.*

This is a very general definition of a pattern. It is probably a bad idea to cite Alexander in this way, because he explains this definition extensively. In particular, how can we distinguish a pattern from a simple recipe? Consider the following example:

Context	You are driving a car.
Problem	The traffic lights in front of you are red. You must not run over them. What should you do?
Solution	Brake.

Is this a pattern? Certainly not. It is just a simple, plain if-then rule. So, again, what is a pattern? Jim Coplien, on the Hillside Web site [Cop04], proposes another, slightly longer definition that summarizes the discussion in Alexander's book:

> *Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves*.

Coplien mentions *forces*. Forces are considerations that somehow constrain or influence the solution proposed by the pattern. The set of forces builds up *tension*, usually formulated concisely as a problem statement. A solution for the given problem has to balance the forces somehow, because the forces cannot usually all be resolved optimally – a compromise has to be found.

To be understandable by the reader, a pattern should describe *how* the forces are balanced in the proposed solution, and *why* they have been balanced in the proposed way. In addition, the advantages and disadvantages of such a solution should be explained, to allow the reader to understand the *consequences* of using the pattern.

Patterns are solutions to recurring problems. They therefore need to be quite general, so that they can be applied to more than one concrete problem. However, the solution should be sufficiently concrete to be practically useful, and it should include a description of a specific software configuration. Such a configuration consists of the participants of the pattern, their responsibilities, and their interactions. The level of detail of this description can vary, but after reading the pattern, the reader should know what he has to do to implement the pattern's solution. As the above discussion highlights, a pattern is not merely a set of UML diagrams or code fragments.

Patterns are never 'new ideas'. Patterns are *proven* solutions to recurring problems. So *known uses* for a pattern must always exist. A good rule of thumb is that something that does not have at least three known uses is not a pattern. In software patterns, this means that systems must exist that are implemented according to the pattern. The usual approach to writing patterns is not to invent them from scratch – instead they are discovered in, and then extracted from, real-life systems. These systems then serve as known uses for the pattern. To find patterns in software systems,

the pattern author has to abstract the problem/solution pair from the concrete instances found in the systems at hand. Abstracting the pattern while preserving comprehensibility and practicality is the major challenge of pattern writing.

There is another aspect to what makes a good pattern, the *quality without a name* (QWAN) [AIS+77]. The quality without a name cannot easily be described: the best approximation is *universally-recognizable aesthetic beauty and order*. So a pattern's solution must somehow appeal to the aesthetic sense of the pattern reader – in our case, to the software developer, designer, or architect. While there is no universal definition of beauty, there certainly are some guidelines as to what is a good solution and what is not. For example, a software system, while addressing a complex problem, should be efficient, flexible and easily understandable. The principle of beauty is an important – and often underestimated – guide for judging whether a technological design is good or bad. David Gelernter details this in his book *Machine Beauty* [Gel99].

Classifications of Patterns in this book

The patterns in this book are *software patterns*. They can further be seen as *architectural* patterns or *design* patterns. It is not easy to draw the line between architecture and design, and often the distinction depends on your situation and viewpoint. For a rough distinction, let's refer to the definition of software architecture from Bass, Clements, and Kazman [BCK03]:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally-visible properties of those components and the relationships among them*.

What we can see here is that whether a specific pattern is categorized as an architectural pattern or a design pattern depends heavily on the viewpoint of the designer or architect. Consider for example the *Interpreter* pattern [GHJV95]. The description in the Gang-of-Four book describes it as a concrete design guideline. Yet according to the software architecture definition above, instances of the pattern are often seen as a central elements in the architecture of software systems, because an *Interpreter* is a central component of the system that is externally visible.

Most of the patterns in this book can be seen as falling into both categories – design patterns and architectural patterns. From the viewpoint of the designer, they provide concrete guidelines for the design of a part of the distributed object middleware. Yet they also comprise larger, visible structures of the distributed object middleware and focus on the most important components and their relationships. Thus, according to the above definition, they are architectural foundations of the distributed object middleware as well.

From patterns to pattern languages

A single pattern describes one solution to a particular, recurring problem. However, 'really big problems' usually cannot be described in one pattern without compromising readability.

The pattern community has therefore come up with several ways to combine patterns to solve a more complex problem or a set of related problems:

- *Compound patterns* are patterns that are assembled from other, smaller patterns. These smaller patterns are usually already well known in the community. Often, for a number of related smaller patterns, known uses exist in which these patterns are always used together in the

same software configuration. Such situations are good candidates for description as a compound pattern. It is essential that the compound pattern actually solves a distinct problem, and not just a combination of the problems of its contained patterns. A compound pattern also resolves its own set of forces. An example of a compound pattern is *Bureaucracy* by Dirk Riehle [Rie97], which combines *Composite, Mediator, Chain of Responsibility*, and *Observer* (all from the GoF book, [GHJV95]).

- A *family of patterns* is a collection of patterns that solves the same general problem. Each pattern either defines the problem more specifically, or resolves the common forces in a different way. For example, different solutions could focus on flexibility, performance or simplicity. Usually each of the patterns has different consequences. A family therefore describes a problem and *several* proven solutions. It is up to the reader to select the appropriate solution, taking into account how he wants to resolve the common forces in his particular context. A good example is James Noble's *Basic Relationship Patterns* [Nob97], which describes several alternatives ways in which logical relationships between objects can be realized in software.
- A *collection*, or *system of patterns* comprises several patterns from the same domain or problem area. Each pattern stands on its own, sometimes referring to other patterns in the collection in its implementation. The patterns form a system because they can be used by a developer working in a specific domain, each pattern resolving a distinct problem the developer might come across during his work. A good example is *Pattern Oriented Software Architecture* by Buschmann, Meunier, Rohnert, Sommerlad, and Stal (also known as POSA 1 [BMR+96]).