REMOTING PATTERNS

Foundations of Enterprise, Internet and Realtime Distributed Object Middleware

Markus Völter
Michael Kircher
Uwe Zdun

WILEY SERIES IN
SOFTWARE DESIGN 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

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

Typeset in 10.7/13.7 pt Book Antiqua by Laserwords Private Limited, Chennai, India
from files produced by the authors
Printed and bound in Great Britain by Biddles Ltd, King's Lynn
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.

Contents

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

1. In his book, *A Pattern Language – Towns • Buildings • Construction* [AIS+77] Christopher Alexander presents a pattern language consisting of 253 patterns about architecture. He describes patterns that guide the creation of space for people to live, including cities, houses, rooms, and so on. The notion of patterns in software builds on this early work by Alexander.

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.*

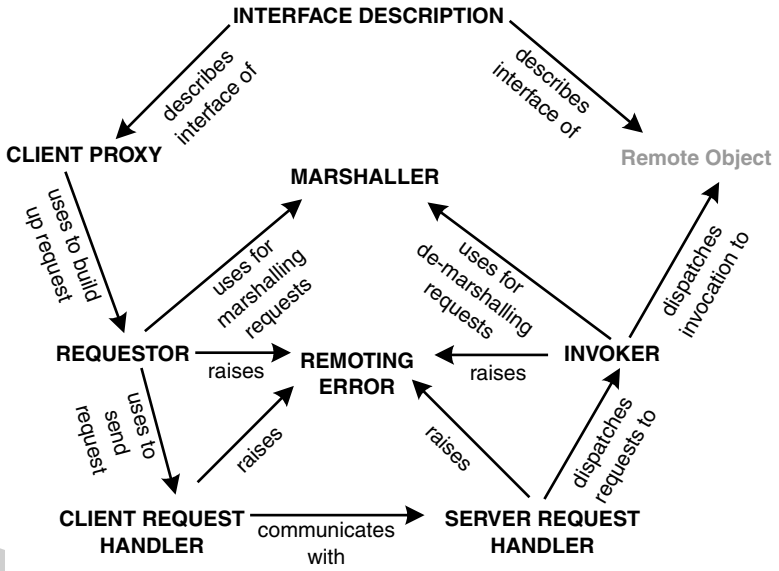
Coplien mentions *forces*. Forces are considerations that somehow constrain or influence the solution proposed by the pattern. The set of forces builds up *tension*, usually formulated concisely as a problem statement. A solution for the given problem has to balance the forces somehow, because the forces 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.

Overview of the Pattern chapters

The patterns mentioned so far detail the BROKER pattern and will be described in Chapter 3, *Basic Remoting Patterns*. The following illustration shows the typical dependencies among the patterns.



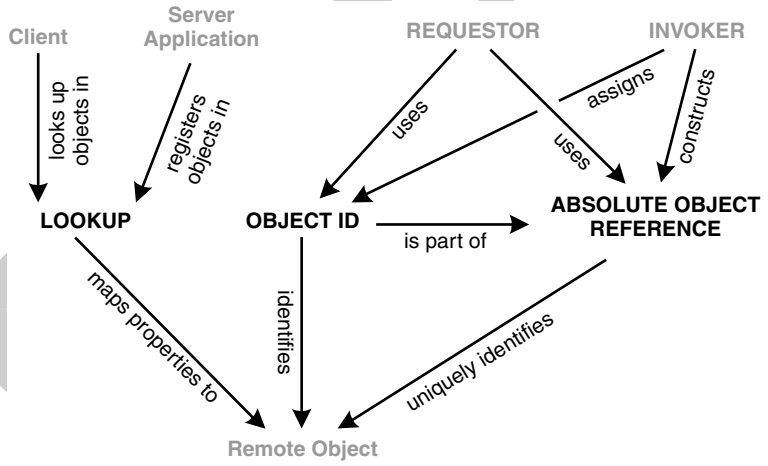
We use such dependency diagrams at the beginning of each pattern chapter. Pattern names are presented in small caps font. Participants, like remote object, are displayed in plain font. The patterns explained in a particular chapter are displayed in black font, while all other pattern and participant names are displayed in gray. The chapters after Chapter 3 present patterns that extend the elementary patterns. The remainder of this section provides an brief overview of them.

The patterns in Chapter 4, *Identification Patterns*, deal with issues of identification, addressing, and lookup of remote objects. It is important for clients to find the correct remote object within the server application. This is done by the assignment of logical OBJECT IDS for each remote object instance. The client embeds these OBJECT IDS in invocations, so that the INVOKER can find the correct remote object. However, this assumes that we are able to deliver the message to the correct server application – in two different server applications, two different

objects with the same OBJECT ID might exist. An ABSOLUTE OBJECT REFERENCE extends the concept of OBJECT IDS with location information. Typical elements of an ABSOLUTE OBJECT REFERENCE are, for example, the hostname, the port, and the OBJECT ID of a remote object.

LOOKUP is used to associate remote objects with human-readable names and other properties. The server application typically associates properties with the remote object on registration. The client only needs to know the ABSOLUTE OBJECT REFERENCE of the lookup service, instead of the potentially huge number of ABSOLUTE OBJECT REFERENCES of the remote objects it wants to communicate with. The LOOKUP pattern simplifies the management and configuration of distributed systems, as clients can easily find remote objects, while avoiding tight coupling between them.

The dependencies among the patterns are illustrated in the following diagram.



Chapter 5, *Lifecycle Management Patterns*, deals with the management of the lifecycle of remote objects. While some remote objects need to exist all the time, others need only be available for a limited period. The activation and deactivation of remote objects might also be coupled with additional tasks.

Lifecycle management strategies are used to adapt to the specifics of the lifecycle of remote objects and their use. These strategies have a strong influence on the overall resource consumption of the distributed application. Chapter 5 describes some of the most common strategies used in today's distributed object middleware.

The three basic lifecycle strategy patterns have the following focus:

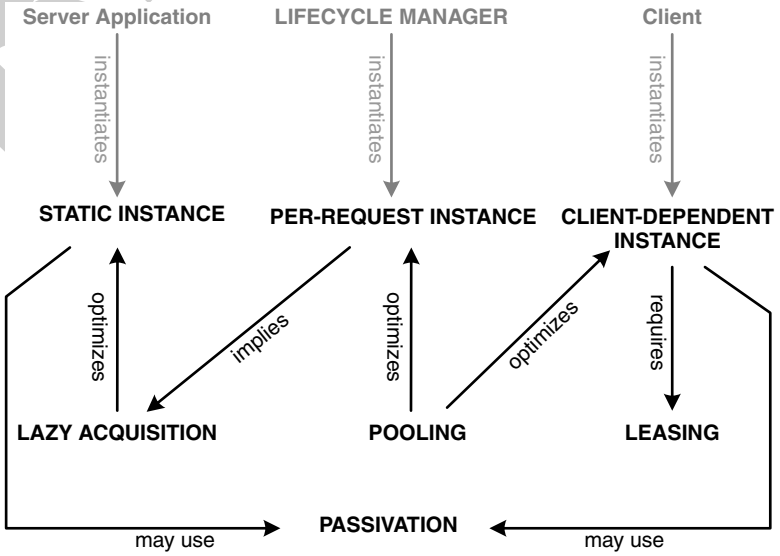
- STATIC INSTANCES are used to represent fixed functionality in the system. Their lifetime is typically identical to the lifetime of their server application.
- PER-REQUEST INSTANCES are used for highly concurrent environments. They are created for each new request and destroyed after the request.
- CLIENT-DEPENDENT INSTANCES are used to represent client state in the server. They rely on the client to instantiate them explicitly.

The lifecycle strategies patterns make use of a set of specific resource management patterns internally:

- LEASING is used to properly release CLIENT-DEPENDENT INSTANCES when they are no longer used.
- LAZY ACQUISITION describes how to activate remote objects on demand.
- POOLING manages unused remote object instances in a pool, to optimize reuse.

For state management, PASSIVATION takes care of removing unused instances temporarily from memory and storing them in persistent storage. Upon request, the instances are restored again.

The patterns and their relationships are shown in the following figure.



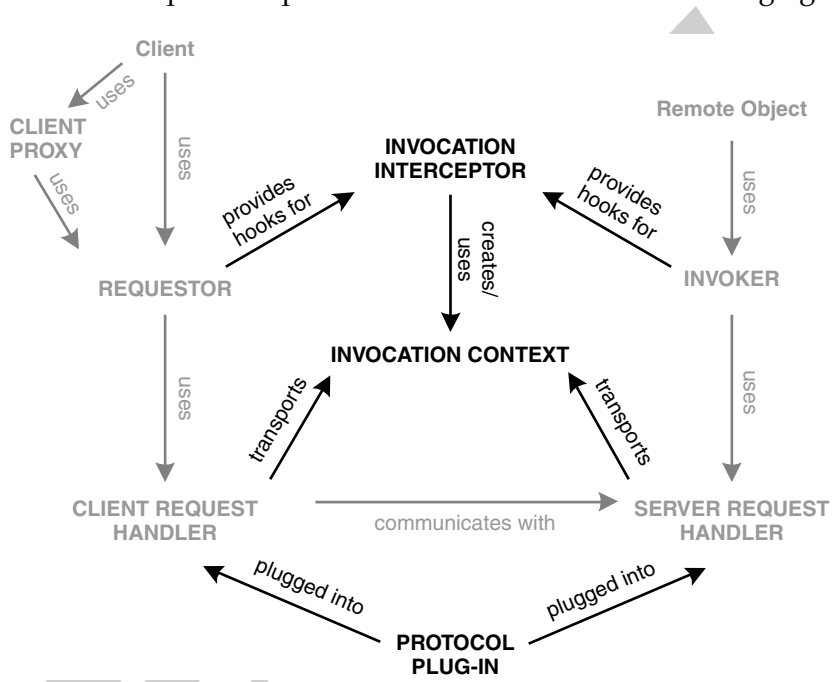
Chapter 6, *Extension Patterns*, deals with patterns that are used to extend distributed object middleware. Examples of such extensions are the support for security, transactions, or even the exchange of communication protocols.

To handle aspects such as security or transactions, remote invocations need to contain more information than just the operation name and its parameters – some kind of transaction ID or security credentials need to be transported between client and server. For that purpose INVOCATION CONTEXTS are used: these are typically added to the invocation on client side transparently and read on server side by the REQUESTOR, CLIENT/SERVER REQUEST HANDLERS, and INVOKER, respectively.

When the invocation process needs to be extended with behavior, for example when evaluating security credentials in the client and the server, INVOCATION INTERCEPTORS can be used. For passing information between clients and servers, INVOCATION INTERCEPTORS use the INVOCATION CONTEXTS we have already mentioned.

Another important extension is the introduction and exchange of different communication protocols. Consider again the example of a secure protocol that might be needed to encrypt the invocation data before it is sent and received using the CLIENT or SERVER REQUEST HANDLER respectively. While simple request handlers use a fixed communication protocol, PROTOCOL PLUG-INS make the request handlers extensible to support different, or even multiple, communication protocols.

The relationships of the patterns are illustrated in the following figure.



Chapter 7, *Extended Infrastructure Patterns*, deals with specific implementation aspects of the server-side BROKER architecture.

The LIFECYCLE MANAGER is responsible for managing activation and deactivation of remote objects by implementing the lifecycle management strategies described in Chapter 5, typically as part of the INVOKER.

To be able to configure groups of remote objects – instead of configuring each object separately – with regard to lifecycle, extensions, and other options, CONFIGURATION GROUPS are used.

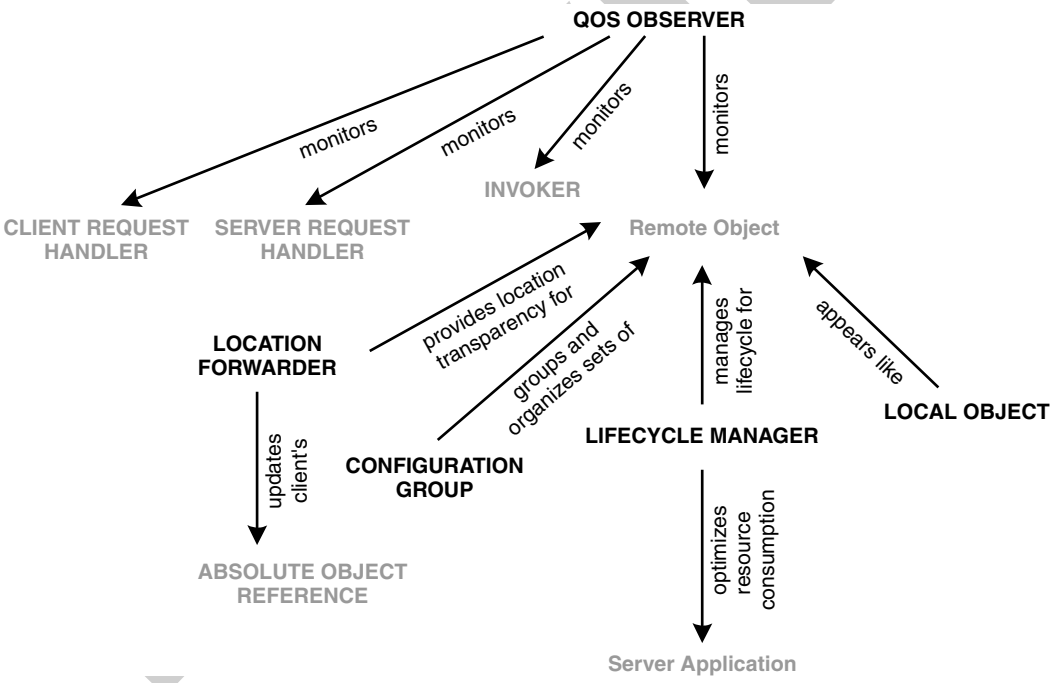
For monitoring the performance of various parts of the system, such as the INVOKER, the SERVER REQUEST HANDLER, or even the remote objects themselves, QOS OBSERVERS can be used. These help to ensure specific quality of service constraints of the system.

To make infrastructure objects in the distributed object middleware follow the same programming conventions as remote objects, while making them inaccessible from remote sites, LOCAL OBJECTS can be

used. Typical LOCAL OBJECTS are LIFECYCLE MANAGERS, PROTOCOL PLUG-INS, CONFIGURATION GROUPS, and INVOCATION INTERCEPTORS.

ABSOLUTE OBJECT REFERENCES identify a remote object in a server. If the remote object instances are to be decoupled from the ABSOLUTE OBJECT REFERENCE, an additional level of indirection is needed. LOCATION FORWARDERS allow this: they can forward invocations between different server applications. This allows load balancing, fault tolerance, and remote object migration to be implemented.

The following figure shows the Extended Infrastructure Patterns and their relationships.

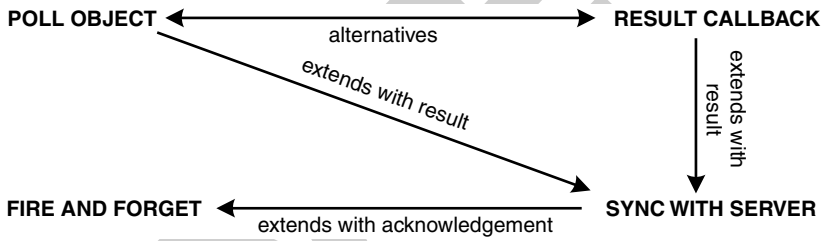


The last patterns chapter in the book, Chapter 8, describes *Invocation Asynchrony Patterns*, and deals with handling asynchronous invocations. It presents four alternative patterns that extend ordinary synchronous invocations:

- FIRE AND FORGET describes best-effort delivery semantics for asynchronous operations that have void return types.

- SYNC WITH SERVER sends an acknowledgement back to the client once the operation has arrived on the server-side, in addition to the semantics of FIRE AND FORGET.
- POLL OBJECTS allow clients to query the distributed object middleware for replies to asynchronous requests.
- RESULT CALLBACK actively notifies the requesting client of asynchronously-arriving replies.

The following figure illustrates the patterns and their interactions.



Lifecycle Manager

The server application has to manage different types of lifecycles for remote objects.

❊ ❊ ❊

The lifecycle of remote objects needs to be managed by server applications. Based on configuration, usage scenarios, and available resources, servants have to be instantiated, initialized, or destroyed. Most importantly, all this has to be coordinated.

The server application has to manage its resources efficiently. For example, it should ensure that only those servants of remote objects that are actually needed at a specific time are loaded into memory.

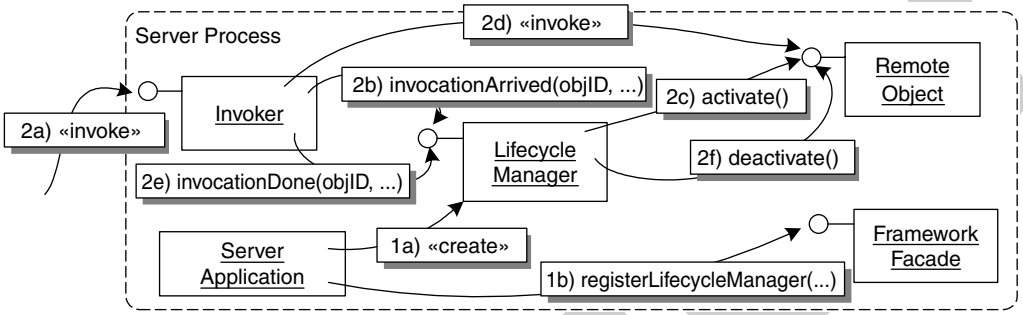
It is not only the creation and loading of servants that is expensive, but also their destruction and clean-up. Clean-up and destruction might involve invoking a destructor, releasing an ABSOLUTE OBJECT REFERENCE, invoking custom clean-up routines, and recycling servants using POOLING.

The lifecycle strategies should not be mixed with other parts of the distributed object middleware, as the strategies can become quite complex, but need to be highly configurable. Specifically, it should be possible for application developers to customize lifecycle strategies, or even implement their own lifecycle strategies.

Therefore:

Use a LIFECYCLE MANAGER to manage the lifecycle of remote objects and their servants. Let the LIFECYCLE MANAGER trigger lifecycle operations for servants of remote objects according to their configured lifecycle strategy. For servants that have to be aware of lifecycle events, provide *Lifecycle Callback* operations [VSW02]. The LIFECYCLE MANAGER will use these operations to notify the servant of

upcoming lifecycle events. This allows servants to prepare for the events accordingly.



A LIFECYCLE MANAGER is typically created by the server application during start-up, and is registered with the distributed object middleware's INVOKER. Before an invocation is dispatched, the INVOKER informs the lifecycle manager. If the servant is not active, the LIFECYCLE MANAGER activates it. The INVOKER dispatches the invocation. After the invocation returns, the LIFECYCLE MANAGER is informed again and can deactivate the servant if required.

❄ ❄ ❄

The LIFECYCLE MANAGER enables modularization of the lifecycle strategies, including activation, PASSIVATION, POOLING, LEASING and eviction, as explained in Chapter 5, *Lifecycle Management Patterns*. Such strategies are important for optimization of performance, stability, and scalability.

The LIFECYCLE MANAGER, with its strategies, is either implemented as part of the INVOKER, or closely collaborates with it. If multiple different lifecycle strategies have to be provided, different LIFECYCLE MANAGERS can be available in the same server application.

The INVOKER triggers the LIFECYCLE MANAGER before and after each invocation. This allows the LIFECYCLE MANAGER to manage the creation, initialization, and destruction of servants.

For complex remote objects, for example those that have non-trivial state, it can become necessary to involve the servants in lifecycle management by informing them about upcoming lifecycle events. For this, the LIFECYCLE MANAGER invokes *Lifecycle Callback* operations implemented by the servant. For example, when using PASSIVATION, a remote object's state has to be saved to persistent storage before the

servant is destroyed. After the servant has been resurrected, the servant has to reload its previously saved state. Both events are triggered by the LIFECYCLE MANAGER via *Lifecycle Callback* operations, just before the servant is passivated and just after resurrection.

Triggering the LIFECYCLE MANAGER can be hard-coded inside the INVOKER, but it can also be 'plugged in' using an INVOCATION INTERCEPTOR. Besides the synchronous involvement of the LIFECYCLE MANAGER when triggered by an INVOKER, a LIFECYCLE MANAGER implementation can also become active asynchronously, for example to scan for remote objects that should be destroyed because some lease has expired.

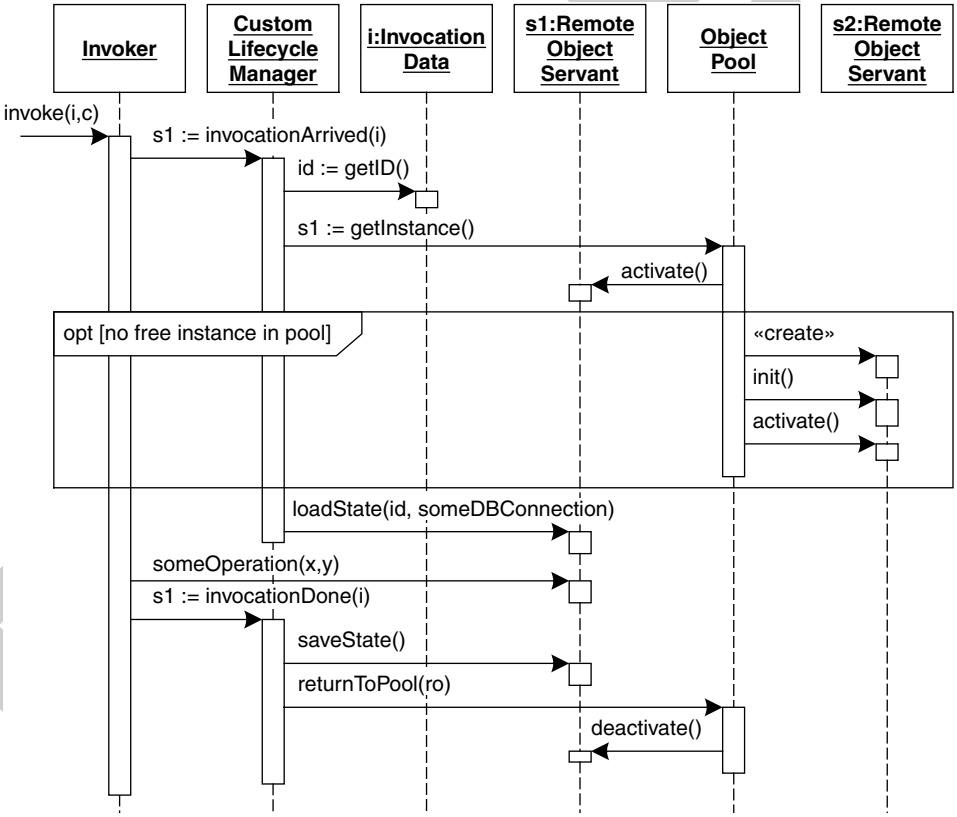
To decouple remote objects from servants, the LIFECYCLE MANAGER must maintain the association between OBJECT ID and servant. This mapping is either kept separately in the LIFECYCLE MANAGER or is reused from the INVOKER. Additionally, the LIFECYCLE MANAGER also has to store information about which lifecycle state each servant is in.

Besides all the advantages of decoupling lifecycle strategies from the INVOKER, the LIFECYCLE MANAGER also incurs a slight performance overhead, as it has to be invoked on every request of a remote object.

The LIFECYCLE MANAGER pattern applies the *Resource Lifecycle Manager* pattern [KJ04] to the management of remote objects. It integrates several existing patterns, such as *Activator* [Sta00] and *Evictor* [HV99].

Interactions among the patterns

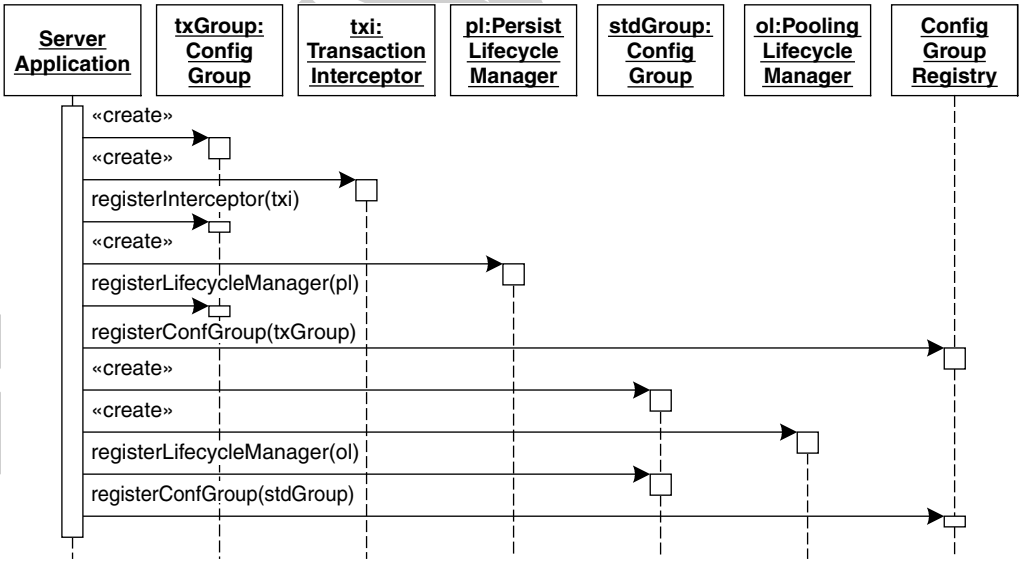
First, let us look at the LIFECYCLE MANAGER. For illustrative purposes, we will show how persistent state can be realized for remote objects in combination with POOLING. In the following example we assume that the developers of the server application have configured the distributed object middleware to use a custom developed LIFECYCLE MANAGER. Note that these interactions are just examples of how the patterns can be implemented – many other alternatives exist.



In the figure above the LIFECYCLE MANAGER obtains a servant from a pool of servants. After the servant is activated (or optionally created lazily), the state associated with the given OBJECT ID is loaded from persistent storage. The operation is executed, and finally the state of the remote object is passivated and the servant is put back into the pool.

Remote object servants that are to be used with this LIFECYCLE MANAGER need to implement up to five lifecycle operations [VSW02]: `init`, `activate`, and `deactivate` to be compatible with POOLING, and `loadState` and `saveState` to be compatible with the persistent state handling used by PASSIVATION.

Next we look at a possible scenario for CONFIGURATION GROUPS. Let us first look at how a server application sets up two CONFIGURATION GROUPS. One contains a LIFECYCLE MANAGER that uses pooling, whereas the other contains a persistency LIFECYCLE MANAGER, as well as a transaction INVOCATION INTERCEPTOR to make the persistency accesses transactional. For each of the CONFIGURATION GROUPS the group object has first to be instantiated. Next, INVOCATION INTERCEPTORS and LIFECYCLE MANAGERS for the group have to be created and registered. Finally, the group has itself to be registered with a group registry of the distributed object middleware.



9 Technology Projections

In the pattern language presented in earlier chapters we did not provide known uses for each of the patterns, just generic examples. We provide more substantial examples for the whole pattern language in the following chapters, in which we look at a specific technology, show how the patterns are applied, and how they relate to each other. In these chapters we emphasize the focus on the pattern language as a whole – instead of focusing on single patterns within the language. Instead of calling these chapters 'Examples of the pattern language' we call them *technology projections*[1].

Each subsequent technology projection is intended to emphasize specific features of the pattern language:

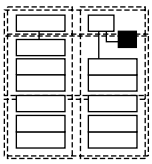
- *.NET Remoting*. .NET Remoting provides a generally usable, flexible and well-defined distributed object middleware. It has a nice and consistent API and can easily be understood, even by novices. It is already widely used and is thus an important remoting technology. We use C# as the example programming language.
- *Web Services*. Web Services are currently one of the hottest topics in the IT industry. Basically, they provide an HTTP/XML-based remoting infrastructure. Our technology projection here focuses especially on interoperability. Java, as a programming language, and Apache Axis, as a framework, are used for most of the examples, but we also discuss other Web Services frameworks, such as .NET Web Services, IONA's Artix, and GLUE.
- *CORBA and Real-time CORBA*. CORBA is certainly the most complex, but also the most powerful and sophisticated distributed object middleware technology available today. In addition to being language-independent and platform-interoperable, there are also implementations for embedded and real-time applications. Our

1. 'Technology projection' is a term we first heard from Ulrich Eisenecker.

technology projection for CORBA will focus especially on quality of service aspects. C++ is used in the examples.

Please note that these chapters cannot serve as complete and full tutorials for the respective technologies. They are really just meant to help you understand the patterns, as well as the commonalities and trade-offs of the technologies.

Reading the technology projections will of course give you a basic understanding of the respective technology, but to use one of the described technologies in practice, you should probably also read a dedicated tutorial.



At the beginning of each of the three technology projections, we present a *pattern map* that illustrates the relationships between the Remoting Patterns and the architecture of the respective technology. To provide a guide to where we are in this overall architecture, in the left-hand margin we display thumbprints of the full pattern map shown on pages 190, 242 and 293. The thumbprint on the left shows an example that denotes that we are in the server application. This example is taken from the .NET technology projection.

10 .NET Remoting Technology Projection

.NET provides an easy to use but yet powerful distributed object middleware that follows the patterns described in the first part of this book very closely. .NET Remoting is a powerful and flexible distributed object middleware. We use the C# programming language exclusively for our examples, although we could have also used other .NET languages such as VB.NET or C++.NET.

Note that we cannot go into every detail of .NET Remoting in this chapter. You can find more information for example in [Bar02], [Ram02], and [Sta03].

A brief history of .NET Remoting

.NET Remoting was introduced as part of Microsoft's .NET platform. From the developer's perspective, .NET replaces the older Windows programming APIs such as the Windows 32-bit API (Win32 API), the Microsoft Foundation Classes (MFC) and, with regard to remoting, DCOM – although DCOM will live on as part of COM+ [Mic04d]. We don't want to compare DCOM and .NET Remoting here, except to say that they have almost nothing in common, and that developing with .NET Remoting is much simpler and more straightforward than DCOM [Gri97].

Note that, technically, .NET does not *replace* the older APIs, but it is built on top of them. This is, however, invisible to developers.

.NET Remoting is an infrastructure for distributed *objects*. It is not ideally suited to building service-oriented systems. Microsoft is planning to release a new product in 2006 (currently named Indigo) that provides a new programming model for Web Services-based, service-oriented systems on top of .NET. It will be possible to migrate .NET Remoting applications, as well as applications using .NET Enterprise services, to Indigo. At the time of writing, further information on

Indigo can be found at [Mic04b] – we also summarize some key concepts in *Outlook for the next generation* on page 235.

.NET concepts – a brief introduction

This section explains some basics that should be understood in outline before reading on.

Just as Java, .NET is based on a virtual machine architecture. The virtual machine is called *Common Language Runtime* (CLR), also named *runtime* in this section. The runtime runs programs written in '.NET assembler', the *Microsoft Intermediate Language* (MSIL). Many source languages can be compiled into MSIL, including C#, C++, VB.NET, Eiffel. Since everything is ultimately represented as MSIL, a great deal of language interoperability is possible. For example, you can let a C++ class inherit from a C# class.

.NET supports *namespaces* to avoid name clashes. Just as in C++, a namespace is logically a prefix to a name. Multiple classes can exist in the same namespace. In contrast to Java, there is no relationship between namespaces and the file system location of a class.

Assemblies are completely independent of namespaces: assemblies are the packaging format, a kind of archive, for a number of .NET artifacts, such as types, resources, and metadata (see below). The elements of a namespace can be scattered across several assemblies, and an assembly can contain elements from any number of namespaces. So, namespaces are a means to structure names logically, whereas assemblies are used to combine things that should be deployed together. It is of course good practice to establish some kind of relationship between namespaces and assemblies to avoid confusion, for example to put all the elements of one namespace into the same assembly.

.NET provides many features that are known from scripting or interpreted languages. For example, it provides reflection: it is possible to query an assembly for its contained artifacts, or to introspect a type to find out its attributes, operations, supertypes, and so on. It is also possible to create .NET types and assemblies on the fly. `CodeDOM` and the `Reflection.Emit` namespace provides facilities to define types, as well as their complete implementation.

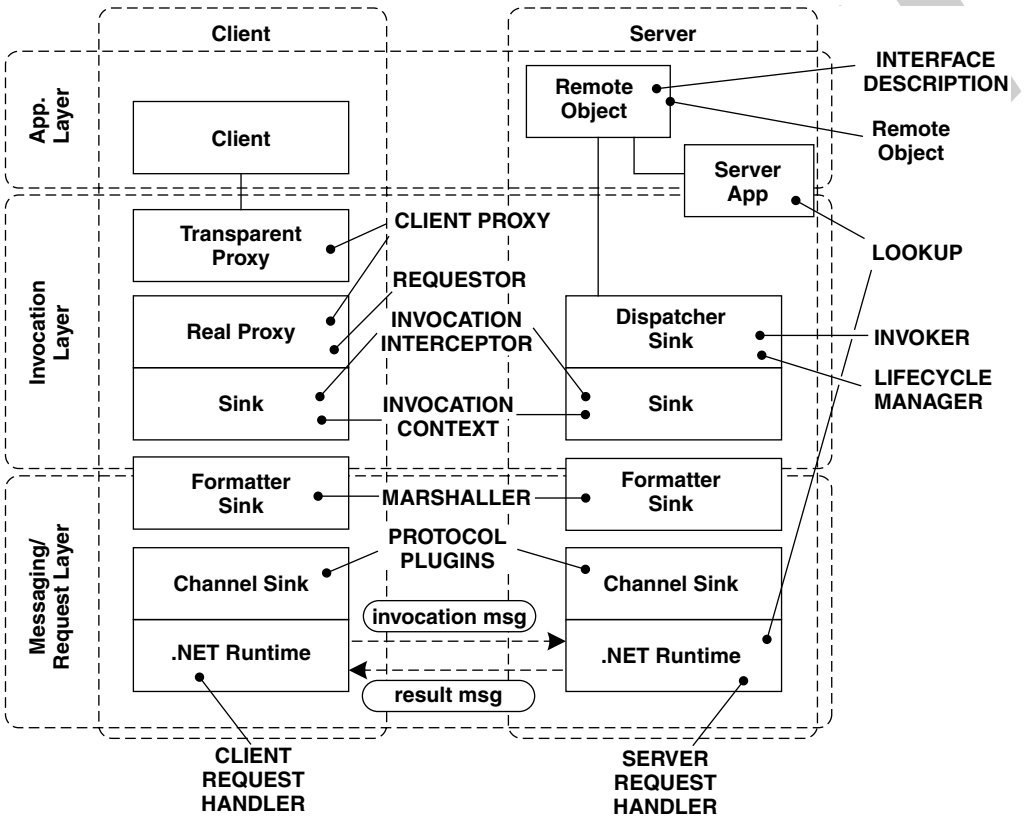
Attributes are another very interesting feature. Many .NET elements, such as types, member definitions, operations, and so on, can be annotated with attributes in the source code. For example the `[Serializable]` attribute specifies that the annotated type should be marshaled by value in the case of a remote invocation. Developers can define their own attributes, which are .NET types themselves. The compiler then instantiates them and serializes them into the assembly, together with the compiled MSIL code. At runtime it is possible to use reflection to find out about the attributes of a .NET element and react accordingly.

In addition to processes and threads, there are additional execution concepts in .NET, such as *application domains*. While threads only define a separate, concurrent execution path, processes in addition define a protection domain. If one process crashes, other processes remain unaffected. As a consequence, communication between processes involves a significant overhead due to process context switching. .NET application domains provide the context of a protection domain independently of the separate, concurrent execution path, and without the context switching overhead.

.NET Remoting pattern map

The following illustration shows the basic structure of the .NET Remoting framework. It also contains annotations of pattern names showing which component is responsible for realizing which pattern.

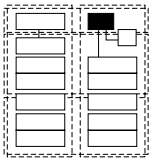
The following overview does not show the behavioral patterns (*Lifecycle* Patterns and *Client Asynchrony* Patterns).



Thumbnails of this diagram in the margin serve as an orientation map during the course of this chapter, by highlighting the area of the above diagram to which a particular paragraph or section relates.

A simple .NET Remoting example

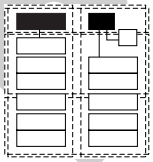
To introduce .NET Remoting we will start with a simple example. Instead of using the stereotypical 'Hello World!', we use a different but equally simple example in which we build a distributed system for the medical domain. Our introductory example comprises a remotely-accessible `PatientManager` object that provides access to the patients' data in a hospital's IT system.



Let us consider the remote object first. In .NET, a remotely-accessible object must extend the `MarshalByRefObject` class, as shown in the following example. `MarshalByRefObject` is thus the base class for all remotely-accessible objects in .NET. The name is an allusion to the fact that the object will not be marshaled (that is, serialized) to a remote machine. Instead only the remote object's reference will be marshaled to the caller if an instance is passed as an operation parameter or as a return value.

```
using System;
namespace PatientManagementServer
{
  public class PatientManager: MarshalByRefObject
  {
    public PatientManager()
    {}
    public Patient getPatientInfo( String id )
    {
      return new Patient( id );
    }
  }
}
```

Here we define the `PatientManager` class, which provides a single operation to retrieve a `Patient` object. The class resides in the `PatientManagementServer` namespace.



.NET does not require a separate interface for remote objects – by default, the public operations of the implementation class are available remotely. To make sure we do not need the remote object class definition (`PatientManager`) in the client process, we provide an interface that defines publicly-available operations. In accordance with .NET naming conventions, this interface is called `IPatientManager`. It is located in a namespace `PatientManagementShared`, which we will use for the code that is required by client and server. The namespaces `PatientManagementServer` and `PatientManagementClient` will be used for server-only and client-only code, respectively.

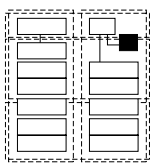
```
namespace PatientManagementShared
{
  public interface IPatientManager
  {
    Patient getPatientInfo( String patientID );
  }
}
```

The implementation must now of course implement this interface to ensure that the `PatientManager` is subtype-compatible with `IPatientManager`. This is necessary because the client will use the `IPatientManager` interface to declare references to remote `PatientManager` objects.

```
namespace PatientManagementServer
{
  public class PatientManager: MarshalByRefObject,
    IPatientManager
  {
    public PatientManager()
    {}
    public Patient getPatientInfo( String id )
    {
      return new Patient( id );
    }
  }
}
```

The operation `getPatientInfo` returns an instance of `Patient`. This class is a typical example of a *Data Transfer Object* [Fow03, Mic04c, Sun04a], which is serialized by value when transported between server and client (and vice versa). In .NET such objects need to contain the `[Serializable]` attribute in their class definition, as shown below. Note that, just as for the interface, the `Patient` class is also defined in the `PatientManagementShared` assembly, because it is used by both client and server.

```
namespace PatientManagementShared
{
  [Serializable]
  public class Patient
  {
    private String id = null;
    public Patient( String _id )
    {
      id = _id;
    }
    public String getID()
    {
      return id;
    }
  }
}
```



To continue, we need to get a suitable server application running for our initial example. This needs to set up the Remoting framework and publish the remote object(s). We postpone the details of setting up the

Remoting framework, and thus just use a 'black box' helper class[1] to do this for us, `RemotingSupport`. With this, the server application becomes extremely simple:

```
using System;
using RemotingHelper;
using System.Runtime.Remoting;
using PatientManagementShared;

namespace PatientManagementServer
{
  class Server
  {
    static void Main(string[] args)
    {
      RemotingSupport.setupServer();
      RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(PatientManager), "PatientManager",
        WellKnownObjectMode.Singleton );
      RemotingSupport.waitForKeyPress();
    }
  }
}
```

The second line of the main method is the interesting part: here we publish the `PatientManager` remote object for access by clients. The registration also includes a definition of a name by which the remote object will be available to clients. The following URL will serve the purpose of an ABSOLUTE OBJECT REFERENCE:

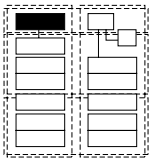
```
http://<the serverHost>:<the port>/PatientManager
```

Note that there is no central LOOKUP system in .NET: clients have to know the host on which an object can be looked up by its name. They therefore must use the URL above. The URL contains information about the transport protocol used to access the object. Since .NET supports different communication *channels* (see below), it is possible, after appropriate configuration, to reach the same object using a different URL. An example using the TCP channel is:

```
tcp://<the serverHost>:<another port>/PatientManager
```

1. This class is not part of the .NET Remoting framework, we have implemented it for the sake of this example. The class itself consists just of a couple of lines, so it does not hide huge amounts of complex code. We show its implementation later.

Note also that we use LAZY ACQUISITION here. `WellKnownObjectMode.Singleton` specifies that only one shared instance is accessible remotely at any time. This instance is created only when the first request for the `PatientManager` remote object arrives. For details of the activation mode, see *Activation and bootstrapping* on page 208.

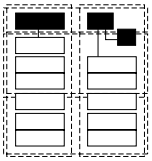


Last but not least, we need a client. This is also relatively straightforward:

```
namespace PatientManagementClient
{
  class Client
  {
    static void Main(string[] args)
    {
      RemotingSupport.setupClient();
      IPatientManager patientManager =
         (IPatientManager)Activator.
            GetObject(typeof(IPatientManager),
                      "http://localhost:6642/PatientManager" );
      Patient patient = patientManager.getPatientInfo("42");
      Console.WriteLine("ID of the "+
        "retrieved Patient:"+patient.getID());
    }
  }
}
```

First we invoke the `setupClient` operation on the `RemotingSupport` class. Then we look up the remote object using the URL scheme mentioned above. We use the `Activator` class provided by the .NET framework as a generic factory for all kinds of remote objects. The port 6642 is defined as part of the `RemotingSupport.setupServer` operation called by the `Server.Main` operation. Note that we use the interface to declare and downcast the returned object, not the implementation class. We then finally retrieve a `Patient` from the remote `PatientManager` and, for the sake of the example, ask it for its ID.

Setting up the framework



We use the `RemotingSupport` helper class to set up the Remoting framework. Internally, this class uses a .NET API to do the actual work. Alternatively, it is also possible to set up Remoting using certain elements in the application configuration XML file. Each .NET application can have an associated configuration file that controls various aspects of the application's use of the .NET framework. The file must be

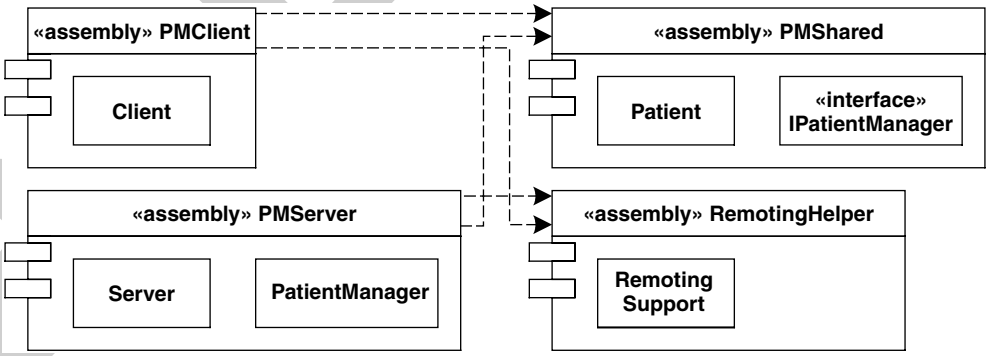
loaded by the program manually. The file contains, for example, the versions of assemblies that should be used, security information, as well as setup information for Remoting. Although we show a snippet of the XML file later in this chapter, we will not consider it in detail in this book.

Assemblies for the simple example

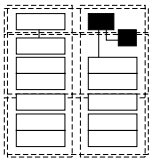
Assemblies in .NET are binary files that are used for deployment. An assembly is represented by a DLL (dynamically loaded library) or an executable. As recommended above, all elements of a particular namespace are put into the same assembly, so we use the following assemblies in this example:

- `PMShared` contains artifacts that are needed by client and server.
- `RemotingHelper` contains everything needed to set up the .NET Remoting framework.
- `PMServer` contains all the code for the server.
- `PMClient` contains all the code for the client.

The following illustration shows the assemblies, the classes they contain, and the dependencies among the assemblies.

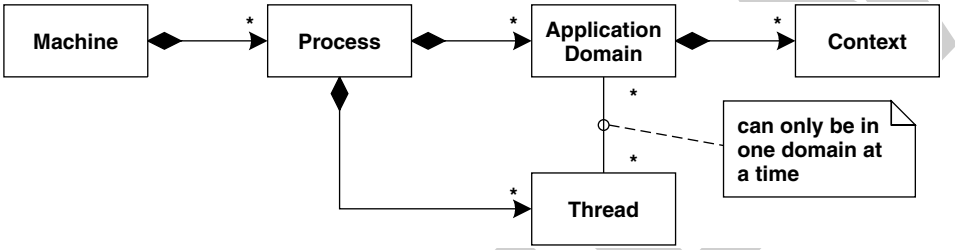


Remoting boundaries



While distributed object middleware is typically used to communicate between two processes that are usually located on different machines, this is technically not completely correct for .NET Remoting.

.NET provides two important additional concepts: *application domains* and *contexts*. The following illustration shows the relationship of the concepts, while the next two sections explain some details.



Application domains

In a .NET managed environment a process can contain several *application domains*. An application domain is a logically-separate 'space' inside a process. Applications running in different application domains are isolated from each other – faults in one application domain cannot affect the code running in other application domains inside the same process. This is mainly achieved by verifying the loaded MSIL code with respect to boundary violations. The code is only executed if no such violations are detected. As a consequence, application domains provide the same isolation features as processes, but with the additional benefit that communication between application domains need not cross process boundaries. Using application domains improves communication performance and scalability, since crossing process boundaries is avoided. However, to communicate between two application domains, .NET Remoting has to be used.

Note that application domains are not related to threads. An application domain can contain several threads, and a thread can cross application domains over time. There is a distinct relationship between assemblies and application domains, though. A specific assembly is always executed in one application domain. For example, you can run the same assembly in several application domains, in which case the code is shared, but the data is not.

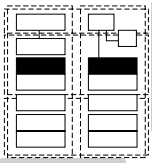
Application domains can be set up manually using the `AppDomain.CreateDomain` operation. This takes a set of parameters such as the domain's working directory, the configuration file for the domain, as well as the search path the runtime uses to find assemblies. Once an application domain is set up, you can use the `Load` operation to load a specific assembly into the domain. `Unload` allows you unload the assembly without needing to stop the process. Finally, you can create instances of a `Type` in a specific application domain using the `CreateInstanceFrom` operation.

In the remainder of this chapter, we will however consider Remoting from the perspective of crossing process boundaries.

Contexts

Contexts provide a way to add INVOCATION INTERCEPTORS to objects running in the same application domain. Again, .NET Remoting is used to insert proxies that handle interception, thus contexts can be used as CONFIGURATION GROUPS. Details are explained later in the section on INVOCATION INTERCEPTORS, or in [Low03].

Basic internals of .NET Remoting



In contrast to other platforms, such as natively-compiled C++, the .NET platform is a relatively dynamic environment. This means that reflective information is available, instances can be asked for their types, and types can be modified, or even created, at runtime. As a consequence, many classes for which you would have to generate and compile source code manually on a native platform can be generated on the fly by the .NET runtime – no separate source code generation or compilation step is required.

An example of such an automatically-generated class is the CLIENT PROXY. You never see any source code for these classes, and there is no code-generated server-side skeleton as part of the INVOKER. The INVOKER – called a *dispatcher* in .NET – is a .NET framework component that uses reflection, as well as the information passed in remote method invocation requests, to invoke the target operation on the remote object dynamically.

As usual, the communication framework is an implementation of the BROKER pattern [BMR+96]. This, as well as the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER, form an integral part of the .NET framework, and are also supported by the .NET runtime itself.

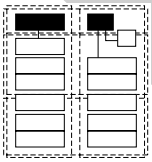
Each remote object instance has a unique OBJECT ID. Using Visual Studio.NET's debugger, we can look into the state of a remote object instance and see the unique OBJECT ID, called `_ObjURI`. In case of our `PatientManager`, it looks like the following:

```
/e478bad4_a6c0_43a1_ae5e_4f7f9bd2c644/PatientManager
```

Another attribute of the `PatientManager` remote object instance is the ABSOLUTE OBJECT REFERENCE. This reference contains several attributes: among others, it contains the OBJECT ID shown above and the communication information in the form of one or more `ChannelData` objects. The URL is the same as the one we introduced before, which allows clients to access remote objects: `tcp://172.20.2.13:6642`

As we shall see later, a remote object can be accessed remotely through several different *channels*. Channels are communication paths that consist besides other things of a protocol and a serialization format. A channel is connected to a network endpoint with a specific configuration. In the case of TCP, this would be the IP address and the port, here 6642. Channels have to be configured when the server application is started.

Error handling in .NET



REMOTING ERRORS are reported using subclasses of `System.SystemException`. For example, if the CLIENT REQUEST HANDLER is unable to contact the server application because of network problems, it throws a `WebException` or a `SocketException` (both extending `SystemException`) to the client, depending on whether a TCP or an HTTP channel is used to access the remote object. To distinguish application-specific exceptions from REMOTING ERRORS clearly, a convention says that application exceptions must not subclass `SystemException`: instead it is recommended that

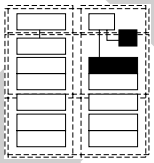
you use `System.ApplicationException` as a base class. An example of a user-defined exception follows:

```
using System;
using System.Runtime.Serialization;

namespace PatientManagementShared
{
  [Serializable]
  public class InvalidPatientID: ApplicationException
  {
    public InvalidPatientID( String _message ) : base(_message)
    {
    }
    public InvalidPatientID(SerializationInfo info,
                            StreamingContext context):
                            base(info, context)
    {}
  }
}
```

Note that the class has to provide a so-called 'deserialization constructor' (the constructor with the `SerializationInfo` and `StreamingContext` parameters) and it has to be marked `[Serializable]`, otherwise the marshaling will not work. The exception's body is empty, because we have no additional parameters compared to `Exception`.

Server-activated instances



.NET provides two fundamentally different options for activating remote objects:

- Remote objects can be activated by the server. In this case, a client can just contact a remote object and does not have to worry about its creation and destruction.
- Remote objects can be created and destroyed explicitly by a client. The lifecycle of these CLIENT-DEPENDENT INSTANCES is thus controlled by a specific client, and not by the server.

This section considers alternatives for server-side activation, while the following section examines CLIENT-DEPENDENT INSTANCES more extensively.