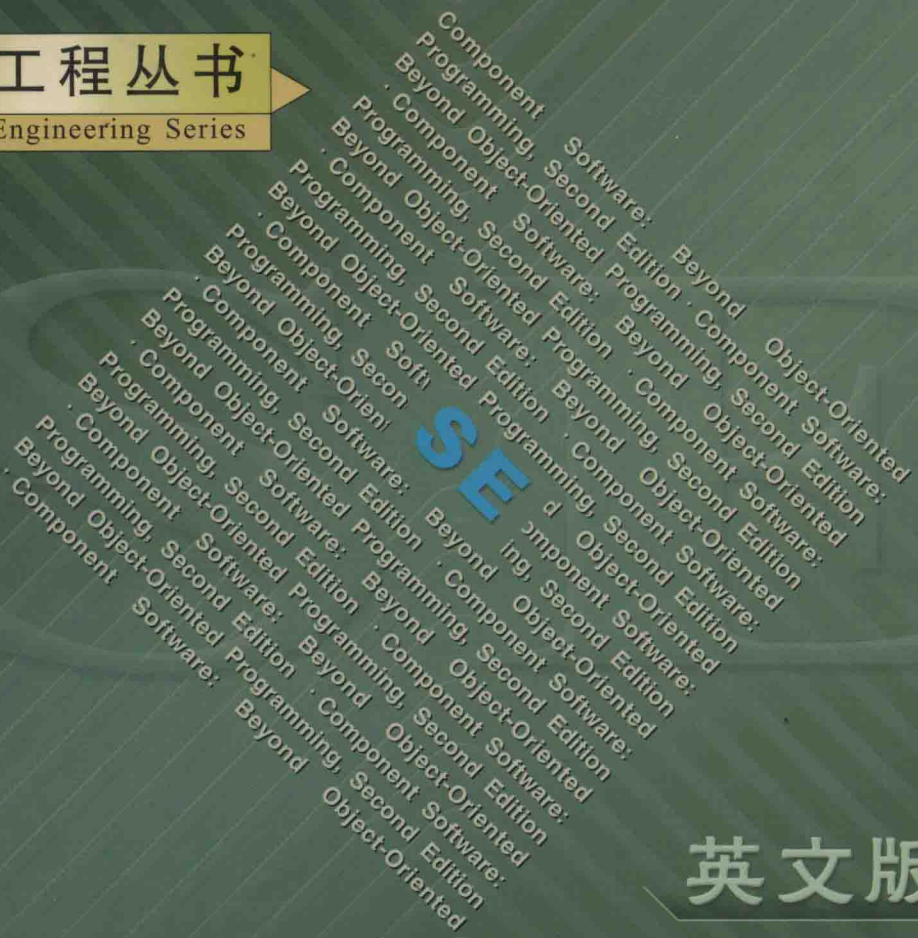
软件工程丛书
Software Engineering Series



英文版

构件化软件
——超越面向对象编程（第二版）

Component Software: Beyond Object-Oriented Programming, Second Edition

[美] Clemens Szyperski　Dominik Gruntz　Stephan Murer　著

電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn

英文版

构件化软件

——超越面向对象编程（第二版）

Component Software: Beyond Object-Oriented Programming, Second Edition

自从这本经典著作的第一版发行之后，又出现了大量的构件化软件技术。EJB、J2EE、CORBA 3、COM+ 以及 .NET 的出现，则是超越 OOP 的构件化软件市场正在成熟的有力证据。本书为我们客观地描述了构件的市场前景；提供了对市场动力的独特观察，该动力影响着系统的部署；并揭示了深层次的实际问题及解决方案。

本书将帮助软件开发人员、系统架构师、CTO 以及系统集成人员理解构件化软件内部的技术问题及市场动力。

第二版的最新内容：

- 市场领先技术的全面更新，包括 COM+、CORBA、EJB 以及 J2EE
- 对一些正在出现的技术的优势与不足进行评价，例如.NET、CORBA构件模型、XML Web 服务，并介绍了它们如何与构件及 XML 的相关标准进行合作
- 在 Java 和 Component Pascal 的基础之上增加了利用 C# 编写的新例子

Clemens Szyperski：瑞士Oberon微系统公司的创始人之一，并参与了BlackBox构件构造器（面向构件编程的最早的开发环境之一）的研制。他是工业界与学术界活跃的演讲者，还参与了多个国家的国家研究基金的评审工作。Szyperski教授发表了许多学术文章并撰写了若干部著作。

ISBN 7-5053-8927-0

9 787505 389274 >

责任编辑：冯小贝
封面设计：毛惠庚

ISBN 7-5053-8927-0/TP · 5178　定价：59.00 元

构件化软件

——超越面向对象编程（第二版）

Component Software: Beyond Object-Oriented Programming, Second Edition

英文版

软件工程丛书

构件化软件

——超越面向对象编程

（第二版）

（英文版）

Component Software

Beyond Object-Oriented Programming

Second Edition

Clemens Szyperski
[美] Dominik Gruntz 著
Stephan Murer

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内容简介

本书全面介绍了软件构件技术涉及的各种问题。作者以构件与市场的关系作为入口，逐步转入对构件、接口、对象、模式、框架、体系结构等基本概念与应用的讨论。书中结合OMG、Sun和Microsoft的解决方案，介绍了构件模型与构件平台；并且在此基础之上，讨论了构件的体系结构，以及构件的发布、获取、组装等与开发过程相关的问题。最后，本书简介了软件构件技术的市场前景。全书覆盖面广，内容丰富，语言简练，并从不同的角度进行了分析、预测，是一本优秀的软件技术参考书。

本书适合于从事软件设计及开发的软件开发人员、系统架构师、CTO、系统集成人员等。

版权贸易合同登记号：图字：01-2002-5480

图书在版编目（CIP）数据

构件化软件——超越面向对象编程 = Component Software: Beyond Object Oriented Programming，Second Edition：第二版 /（美）泽帕斯基（Szyperski, C.）等著. - 北京：电子工业出版社，2003.8
（软件工程丛书）
ISBN 7-5053-8927-0

Ⅰ. 构...　Ⅱ. 泽...　Ⅲ. 软件工程 - 英文　Ⅳ. TP311.5

中国版本图书馆CIP数据核字（2003）第062597号

责任编辑：冯小贝
印 刷 者：北京兴华印刷厂
出版发行：电子工业出版社　http://www.phei.com.cn
　　北京市海淀区万寿路173信箱　邮编：100036
经　　销：各地新华书店
开　　本：787 × 980　1/16　　印张：38.75　字数：868千字
版　　次：2003年8月第1版　　2003年8月第1次印刷
定　　价：59.00元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077

Preface to the second edition

Writing a book is hard work; preparing a new edition of one's own old words is even harder in many ways. My motivation for venturing into this work is the strong and positive feedback and encouragement I received over the past years from so many of my readers. The first edition of this book achieved a level of worldwide recognition well beyond my hopes. Today, the topic area is prominent enough to attract many good authors to write books on the many facets of component software. Some of this work comes to my attention in its early stages, for instance when submitted to a conference where I serve on the program committee. More mature work reaches me in my function as series editor of Addison-Wesley's Component Software Series. Yet, all this is only scratching the tip of the proverbial iceberg: *much* is happening in this field at large. Any fair and complete coverage of this ballooning field is now close to impossible and I make no pretense that this second edition gets close to such coverage. Instead, I hope to include what I perceive as the major trends, both as a continuation from what I described in the first edition and also entirely new developments that have emerged since.

In its first edition, this book has been adopted as primary or recommended reading for many university courses in countries around the globe. Close to my heart is the fact that the first edition was translated into Polish – my family name is Polish – but reading it is entirely beyond my own language skills as I hardly speak two Polish words. Some of these developments are traced on a web page I maintain (there is a link from my homepage at www.research.microsoft.com/~cszypers/). Some of the problems I had reported as open in the first edition have attracted the attention of several researchers, leading to progress on several fronts: this second edition reports on some of the progress made.

Concurrent to these scientific developments, we have seen an explosive development of component software technologies. On the one hand, many technologies did not survive long after I closed the first edition in mid 1997 – OpenDoc and SOM are two visible cases; there are many others. On the other hand, many of the technologies relevant today were not even around back then. For example, Enterprise JavaBeans and Java 2 Enterprise Edition on the Java front, as well as the CORBA Component Model and CORBA 3 had yet to hap-

pen. CORBA had yet to embrace Java and J2EE had yet to embrace CORBA. COM+ had just become visible and .NET did not exist back then. XML and UML were just appearing on the radar screen, but hadn't had their overwhelming impact yet. Practically all XML-related standards (XML Schema, XML Namespaces, XPath, XLink, XPointer, XQuery, XSL, XSLT, and others) had yet to be publicized. Web Services and their supporting standards (SOAP, WSDL, UDDI, and so on) were entirely unheard of. Much of the work leading to many of these had, of course, been going on behind the scenes – and for years – but it had been far too early for any useful coverage to be included in a work like this book.

At the time of writing the first edition, it had been painfully clear that for component technologies to go much further, domain-specific standards were an absolute requirement. Much has happened since, especially in connection with XML. Put under pressure by a rapidly tightening need for businesses to form business-to-business chains, and put into agreeable form by the technology-neutral and thus "harmless" XML approach, domain-specific standards are now mushrooming. Organizations such as BizTalk, DMTF, IETF, OAG, OASIS, OMG, UDDI, W3C, and WS-I rapidly build repositories of XML-based domain standards. Domain-specific organizations in many industries are adding to this gold rush. Clearly, we will soon see too many rather than too few standards in many important domains, which will undoubtedly lead to a shake-out over the coming years. (However, notice that the world seems to have an insatiable hunger for standards!)

In line with many hopeful predictions, yet still not quite as explosive as some had hoped, the market side of software components has also matured significantly since this book first appeared. There are now several companies, including ILOG and Rogue Wave Software, deriving most of their revenue from software components and several others fully focusing on making the market-side work, including ComponentSource and Flashline. The latter companies include warehousing, brokering, and mediation services that bridge supply and demand sides, just as is already well-established practice in the component worlds of other engineering disciplines.

All in all, it is now time for a second edition. The theme, the balanced and critical viewpoint (I hope), overall structure, and emphasis on foundations and principles have not changed. A myriad of detail-level improvements and corrections re-establish the link to this quickly evolving field. The most significant additions can be found in Part Three, covering the state-of-the art component technologies. Part Four used to be about the next generation of technologies and problems to tackle. This has now changed and offers, instead, a perspective on components meeting architecture and processes. While this has always been the main theme of the fourth part, it is now possible to draw on rich examples from current technologies rather than on speculation of what might be.

For guidance on how to read this book and on whom it addresses, consult the original preface that I retain in its entirety.

The endless struggle for perfect terminology

What is a software component? As with the first edition, this book has many pages on that fundamental question. It contains three different definitions that adopt different levels of abstraction: a first one is found at the very beginning of the original Preface; a second in Chapter 4; and a final one in Chapter 20. The existence of more than one definition in this book – and quite a few more cited from related work (see Chapter 11) – has led to some turbulence. Krzysztof Czarnecki and Ulrich Eisenecker (2000), in their excellent book *Generative Programming*, went as far as claiming that the term "component" (and thus "software component") cannot be defined – for a brief discussion see section 11.12.

I received a lot of feedback on the first edition that addressed my choice of terminology, telling me that I had overstretched certain terms. I ran into particular trouble with my use of the terms "binary form" and "no persistent state," both of which I claimed a software component had to comply with. This has led to toing and froing on various occasions, but I have ended up defending my original choice of words. Such disputes over words have led to rather productive opinion-forming exchanges over the deeper issues – the one that has run the longest and is my favorite being the "Beyond Objects" series of monthly columns in *Software Development* magazine (www.sdmagazine.com), created by Roger Smith. This series includes contributions by Grady Booch, Cris Kobryn, Bertrand Meyer, Bruce Powel Douglass, Jeff Scanlon, and me; others might chip in as the series evolves. I encourage readers to browse these columns as they are naturally closer to the pulse of time than a book can be.

New terminology in this second edition focuses on two developments – the growing importance of component deployment, and the relationship between components and services. To address the deployment process, I now distinguish deployable components (or just components) from deployed components (and, where important, the latter again from installed components). Component instances are always the result of instantiating an installed component – even if installed on the fly. Services are different from components in that they require a service provider. A service is an instance-level concept – where such instances can be component instances. These instances are "live" and thus require grounding in concrete hardware, software, and organizational infrastructure. The term "service" is unfortunately even more overloaded than the term "component." I did not try to rename the many things called service throughout the book, following the many established usages of this word. Instead, I use the term "web service" when referring to a service that is concretely provided, ultimately by some organization (or individual). This convention isn't strictly accurate, as non-web services can have the same properties, but trying to establish an entirely new term, such as "provided service" seemed worse. (To be even more precise, most concrete discussion in this book is about XML web services – a subset of web services that relies on XML as the fundamental representation format.)

I have tried to improve some terminology over the first edition to minimize misunderstandings. After careful deliberation, I decided to change terms only in two cases. I avoided changes in all other cases to maintain continuity from the first edition and avoid confusion that would be caused by the many references to this book that can be found in the wider literature. (For the same reason, I also decided to leave the top-level chapter structure intact.)

The first change is from old "binary form" to new "executable form." This new term makes it much more obvious that I am after a form for components that is defined relative to some execution engine, whether this is a script interpreter, a JIT compiler, or a processor, and that I am not insisting on the binary format dictated by a particular processor or operating system. This change causes some slight friction when discussing the notions of "COM as a binary standard" and "binary release-to-release compatibility." I retained the use of "binary" in these widely established cases. The new term is also somewhat too specific in that a software component also contains metadata and resources (immutable data), none of which are executable in a strict sense, but then neither are they necessarily binary. (For completeness, a degenerate component might contain nothing but such non-executable items. Other authors have thus opted for "machine interpretable.") Finally, there is a danger that some might interpret executable as meaning "must have a `main()` entry point," which clearly isn't intended. With terminology it is impossible to win.

The second change is from old "no persistent state" to new "no observable state." This addresses a common confusion, that persistence in the sense of external stable storage is somehow involved here, which wasn't the intention. Another common confusion cannot be addressed by simple terminology change. This is that whenever I say "component" (or, more precisely, "software component") I am not referring to object-like instances, but, rather, to notions that are more stable across time and space, such as classes, modules, or immutable prototype objects. Components are the units of deployment and, often, components contain classes or other means to create regular instances (objects). I am not, in general, worried about stateful objects, but merely exclude stateful components, which amounts to excluding the observable use of global variables (aka static variables). Occasionally it is appropriate to use such variables for caching purposes – thus the restricted exclusion of an observable state only. Much of this confusion has been triggered by the discussion of whether or not to support objects that carry state across transactional session boundaries in systems such as COM+ or EJB (EJB does allow such objects; COM+ does not.) As should be clear by now, such "stateful objects" and the claim that software components have no (observable) state have nothing to do with each other.

Updated statement and time stamp

I completed the second edition in the first half of 2002 – after a lengthy journey of well over a year's duration. I wish to acknowledge that I have added yet another bias to my list of biases – this time by joining Microsoft Research in 1999. While I hope that I succeeded in retaining the balance of my original work, I certainly understand if readers are more skeptical about this than they were before this development. After all, I am now employed by one of the primary parties involved, rather than being an academic observer with a hand in a small Swiss company alone (a role that I happily still retain). However, this book should certainly not be seen as necessarily coinciding with the views of Microsoft. Some may sense that I am overly or prematurely enthusiastic about .NET or web services, but I gave the same benefit of the doubt to then-young Java (JavaBeans, for instance, emerged while I was working on the first edition) and today I am giving it to the CORBA Component Model.

To offset such skepticism, I invited Dominik Gruntz to carefully review and contribute to the core chapter on Java and Stephan Murer to do the same for the core chapter on OMG standards and technologies. Both are long-standing friends who had already helped with their comments on the draft of the first edition. I am most grateful to them both for accepting my invitation and helping me to uphold the spirit of this book into its second edition. I would like to thank Christian Becker, Bill Councill, Scott Crawford, George Heineman, and an anonymous reviewer for reviewing the entire draft and providing many useful comments and suggestions. Hans Jonkers and Ron Kay provided further comments on the basis of the first edition. Alistair Barros drew on his extensive experience and helped with many details regarding EJB servers.

Any remaining mistakes and possibly undue bias are of course mine.

Clemens Szyperski
Redmond, May 2002

Preface

(The following preface remains unedited from the first edition. References to "recent" developments should be seen from a historic perspective in our fast-moving world.)

Software components enable practical reuse of software "parts" and amortization of investments over multiple applications. There are other units of reuse, such as source code libraries, designs, or architectures. Therefore, to be specific, *software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.* Insisting on independence and binary form is essential to allow for multiple independent vendors and robust integration.

Building new solutions by combining bought and made components improves quality and supports rapid development, leading to a shorter time to market. At the same time, nimble adaptation to changing requirements can be achieved by investing only in key changes of a component-based solution, rather than undertaking a major release change.

For these reasons, component technology is expected by many to be *the* cornerstone of software in the years to come. There exists at least one strong indicator: the number of articles and trivia published on these matters grows exponentially. Software component technology is one of the most sought-after and at the same time least-understood topics in the software field. As early as 1968, Doug McIlroy predicted that mass-produced components would end the so-called software crisis (Naur and Randall, 1969). With component technology just on the verge of success in 1997, this is a 30-year suspense story.

Software components are clearly not just another fad – the use of components is a law of nature in any mature engineering discipline. It is sometimes claimed that software is too flexible to create components; this is not an argument but an indication of immaturity of the discipline. In the first place, component markets have yet to form and thus many components still need to be custom-made. Introduction of component software principles at such an early stage means: preparing for future markets.

Even in a pre-market stage component software offers substantial software engineering benefits. Component software needs modularity of requirements,

architectures, designs, and implementations. Component software thus encourages the move from the current huge monolithic systems to modular structures that offer the benefits of enhanced adaptability, scalability, and maintainability. Once a system is modularized into components, there is much less need for major release changes and the resulting "upgrade treadmill" of entire systems.

Once component markets form, component software promises another advantage: multiplication of investment and innovation. Naturally, this multiplier effect, caused by combining bought and custom-made components, can only take effect when a critical mass is reached – that is, a viable market has formed. For components to be multipliers, there needs to be a competitive market that continually pushes the envelope – that is, it continually improves cost–performance ratios. However, creating and sustaining a market is quite a separate problem from mastering component technology. It is this combination of technical and economic factors that is unique to components.

It is indeed the interplay of technology and market strategies that is finally helping components to reach their long-expected role. However, it would be unfair to say that technically this has been possible since the early days of objects. After all, objects have been around for a long time: Simula's objects, for example, date back to 1969. The second driving force behind the current component revolution is a series of technological breakthroughs. One of the earliest was the development at Xerox PARC and at NeXT in the late 1980s. The first approach that successfully created a substantial market came in 1992 with Microsoft's Visual Basic and its components (VBXs). In the enterprise arena, OMG's CORBA 2.0 followed in mid-1995. The growing popularity of distribution and the internet led to very recent developments, including Microsoft's DCOM (distributed component object model) and ActiveX, Sun's Java and its JavaBeans, the Java component standard.

There is one technical issue that turned out to be a major stumbling block on the way to software component technology. The problem is the widespread misconception of what the competing key technologies have to offer and where exactly they differ. In the heat of the debate, few unbiased comparisons are made. It is a technical issue, because it is all about technology and its alleged potential. However, it is just as much a social or societal issue. In many cases, the problems start with a confusion of fundamental terminology. While distribution, objects, and components really are three orthogonal concepts, all combinations of these terms can be found in a confusing variety of usages. For example, distributed objects can be, but do not have to be, based on components – and components can, but do not have to, support objects or distribution.

The early acceptance of new technologies (and adoption of "standards") is often driven by non-technical issues or even "self-fulfilling prophecies." Proper standardization is one way to unify approaches and broaden the basis for component technology. However, standards need to be feasible and practical. As a sanity check, it is helpful if a standard can closely follow an actual and viable implementation of the component approach. What is needed is a demonstration

of the workability of the promised component properties, including a demonstration of reasonable performance and resource demands. Also, there need to be at least a few independently developed components that indeed interoperate as promised.

For a good understanding of component software, the required level of detail combined with the required breadth of coverage can become overwhelming. However, important decisions need to be made – decisions that should rest firmly on a deep understanding of the nature of component software. This book is about component software and how it affects engineering, marketing, and deployment of software. It is about the underlying concepts, the currently materializing technologies, and the first stories of success and failure. Finally, it is about people and their involvement in component technology.

This book aims to present a comprehensive and detailed account of most aspects of component software: information that should help to make well-founded decisions; information that provides a starting point for those who then want to dig deeper. In places, the level of detail intentionally goes beyond most introductory texts. However, tiring feature enumerations of current approaches have been avoided. Where relevant, features of the various approaches are drawn together and directly put into perspective. The overall breadth of the material covered in this book reflects that of the topic area; less would be too little.

Today there are three major forces in the component software arena. The Object Management Group, with its CORBA-based standards, entered from a corporate enterprise perspective. Microsoft, with its COM-based standards, entered from a desktop perspective. Finally, Sun, with its Java-based standards, entered from an internet perspective. Clearly, enterprise, desktop, and network solutions will have to converge. All three players try to embrace the other players' strongholds by expansion and by offering bridging solutions. As a result, all three players display "weak spots" that today do not withstand the "sanity check" of working and viable solutions. This book takes a strategic approach by comparing technical strengths and weaknesses of the approaches, their likely directions, and consequences for decision making.

Significant parts of this book are non-technical in nature. Again, this reflects the very nature of components – components develop their full potential only in a component market. The technical and non-technical issues are deeply intertwined and coverage of both is essential. To guide readers through the wide field of component software, this book follows an outside-in, inside-out approach. As a first step, the component market rationale is developed. Then, component technology is presented as a set of technical concepts. On the basis of this foundation, today's still evolving component approaches are put into perspective. Future directions are explained on the grounds of what is currently emerging. Finally, the market thread is picked up again, rounding off the discussions and pointing out likely future developments.

Who should read this book – and how: roadmaps

As wide as the spectrum of this book is so, it is envisioned, are the backgrounds and interests of its expected readers. To support such a variety of readers, the book is written with browsing in mind. Most chapters are relatively self-contained and so can be read in any order, although sequential reading is preferable. Where other material is tightly linked, explicit cross-references are given. For selective "fast forwards," various references to later sections are given to aid skipping to natural points of continuation. Forward references are always of an advisory nature only and can be safely ignored by those reading the book sequentially.

Professionals responsible for a company's software strategy, technology evaluation, or software architecture will find the book useful in its entirety. Reading speed may need to be adjusted according to pre-existing knowledge in the various areas covered in Parts Two and Three. The numerous discussions of relative advantages and disadvantages of methods and approaches are likely to be most useful.

Managers will find the coverage sufficiently general to enable the formation of a solid intuition, but may want to skim over some of the more detailed technical material. In the end, decisions need to be based on many more factors than just the aspects of a particular technology. To this end, the book also helps to put component technology into perspective. A suggested path through this book is Part One; Chapters 4, 8, and 11 of Part Two; Chapters 12 and 17 of Part Three; Part Five.

Developers will appreciate the same intuition-building foundation, but will also find enough detail on which to base technical decisions. In addition, developers facing multiple platforms or multiple component approaches will find the many attempts at concept unification useful. Fair technical comparison of similarities and differences is essential to develop a good understanding of the various tradeoffs involved; terminology wars are not. A suggested path through this book is Parts Two, Three, and Four, supported by Parts One and Five if market orientation is required.

Academics and students on advanced courses will find the book a useful and rich source of material. However, although it could serve as reference reading for various units, it is not a textbook. Those studying units focusing on component technology will benefit the most from reading this book, including coverage of specific component technologies such as Java or ActiveX. If the units are on software engineering, students will also benefit from the information in these pages. Finally, if the units are on advanced or comparative programming languages, students could find this book useful as they may expand to language issues in component technology. A suggested path through this book depends on the needs of the particular subject. Part Two, and in particular Chapter 4, forms a basis; Chapters 5, 6, and 7 can be included for more intensive courses or postgraduate studies. The remaining chapters of Part Two can be included selectively. Part Three offers a rich selection of detailed information on current technology. Part

Four explores current developments. Parts One and Five may be of interest for readers on courses with an organizational or market perspective.

Statement and time stamp

I completed this book in the first half of 1997. In a rapidly emerging and changing field, a certain part of the material is likely to be out of date soon. I tried to avoid covering the obviously volatile too deeply and, instead, aimed at clear accounts of the underlying concepts and approaches. For concreteness, I nevertheless included many technical details. I am a co-founder of Oberon microsystems, Inc., Zurich (founded in 1993), one of the first companies to focus fully on component software. In addition to carefully introducing and comparing the main players, I frequently drew on Oberon microsystems' products for leading-edge examples and comparison. These include the programming language Component Pascal, the BlackBox component framework and builder, and the component-oriented real-time operating system Portos with its development system Denia. The choice of these examples clearly reflects my involvement in their development, as well as my active use of several of these tools in university courses. Despite this personal bias, I aimed at a fair positioning of all the approaches I covered.

This book in its present form would not have been possible without the help of many who were willing to read early drafts and supported me with their scrutiny and richness of comments and ideas. In particular, I would like to thank Cuno Pfister, who reviewed the entire draft, some parts in several revisions, and provided numerous comments and suggestions. Daniel Duffy, Erich Gamma, Robert Griesemer, Stephan Murer, Tobias Murer, Wolfgang Pree, and Paul Roe also commented on the entire draft. Dominik Gruntz, Wolfgang Weck, and Alan Wills provided deep and important comments on selected chapters. Marc Brandis, Bert Fitié, John Gough, and Martin Odersky provided further important comments. Remaining mistakes and oversights are of course mine.

Clemens Szyperski
Brisbane, June 1997

About the author

Clemens Szyperski joined Microsoft Research at its Redmond, Washington, facility in 1999 to continue his work on component software. He is currently also an Adjunct Professor of the Faculty of Information Technology at the Queensland University of Technology (QUT), Brisbane, Australia, where he was previously an Associate Professor. He joined the faculty in 1994 and received tenure in 1997. From 1995 to 1999 he has been director of the Programming Languages and Systems Research Centre at QUT.

From 1992 to 1993 he held a Postdoctoral Fellowship from the International Computer Science Institute (ICSI) at the University of California at Berkeley. At ICSI he worked in the groups of Professor Jerome Feldman (Sather language) and Professor Domenico Ferrari (Tenet communication suite with guaranteed Quality of Service).

In 1992, Clemens received his PhD in Computer Science from the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, where he designed and implemented the extensible operating system Ethos under the supervision of Professor Niklaus Wirth and Professor Hanspeter Mössenböck. In 1987, he received a degree in Electrical Engineering/Computer Engineering from the Aachen University of Technology (RWTH), Germany. Ever since joining ETH in 1987, his work has been heavily influenced by the work of Professor Wirth and Professor Jürg Gutknecht on the Oberon language and system.

In 1993, he co-founded Oberon microsystems, Inc., developer of BlackBox Component Builder, first marketed in 1994 and one of the first development environments and component frameworks designed specifically for component-oriented programming projects. In 1997, Oberon microsystems released the new component-oriented programming language Component Pascal. He was a key contributor to both BlackBox and Component Pascal. In 2000, Professor John Gough, Dean of Information Technology at QUT, ported Component Pascal to the Microsoft .NET common language runtime.

In 1999, Oberon microsystems spun out a new company, esmertec, inc., that took the hard realtime operating system then called Portos and turned it into JBed, an industry-leading hard realtime operating system for Java in embedded systems.

Clemens has been a consultant to major international corporations. He served as an assessor and reviewer for Australian, Canadian, Irish, and US federal funding agencies and for learned journals across the globe. He served as a member of program and organizing committees of numerous events, including ECOOP, ICSE, and OOPSLA conferences. He has published numerous papers and articles, several books, and frequently presents at international events.