Server Component Patterns

Component Infrastructures Illustrated with EJB

Markus Völter, Alexander Schmid and Eberhard Wolff

JOHN WILEY & SONS, LTD

Server Component Patterns

Server Component Patterns

Component Infrastructures Illustrated with EJB

Markus Völter, Alexander Schmid and Eberhard Wolff

JOHN WILEY & SONS, LTD

Telephone (+44) 1243 779777

E-mail (for orders and customer service enquiries): cs-books@wiley.co.uk

Visit our Home Page on www.wileyeurope.com or www.wiley.com

Other Wiley Editorial Offices

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

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

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

John Wiley & Sons Australia Ltd, 33 Park Road, Milton, Queensland 4064, Australia

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

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

Library of Congress Cataloging-in-Publication Data (to follow)

British Library Cataloguing in Publication Data

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

ISBN 0 470 84319 5

Typeset in Book Antiqua by WordMongers Ltd, Treen, Cornwall TR19 6LG, England

Printed and bound in Great Britain by Biddles Ltd., Guildford and Kings 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

Preface

What this book is about

This book is about component-based development on the server. Examples for such technologies are Enterprise JavaBeans (EJB), CORBA Components (CCM) or Microsoft's COM+, which have all gained widespread use recently. To build successful applications based on these technologies, the developer should have an understanding of the workings of such a component architecture – things the specification and most books don't talk about very much. Part I of this book contains a pattern language that describes these architectures conceptually. To provide 'grounding in the real world', Part II contains extensive examples of these patterns using the EJB technology. Lastly, Part III shows the benefits of the patterns for a real application.

Who should read this book

Distributed components, and specifically EJB, are 'complex technologies with a friendly face'. This means that a lot of the complexity is hidden behind simple interfaces or implemented with wizards et cetera. However, to create efficient and maintainable applications based on these technologies, we think it is necessary to understand how these architectures actually work and how some specific design techniques must be used. These techniques are very different from object orientation – trying to use OO techniques for components can result in very inefficient systems.

This book consists of three separate parts. Part I is intended for developers or architects who want to learn about the basic principles and concepts used in any of the mainstream component technologies. We use the form of a pattern language for this task. Because each pattern comprises a short example in EJB, CCM and COM+, you can also use this section to understand the differences between these technologies.

Part I can also help you to create your own specialized component architectures because the concepts are described in a very general way.

Part II serves two main purposes. First, it illustrates the patterns from Part I with more concrete, extensive examples. Since EJB is used for the examples, this section is most useful for EJB developers, although COM+ and CCM users can also learn a lot about the details how the patterns are implemented. You can also regard this part as a concise tutorial for EJB although we assume some basic understanding of EJB.

Part III shows finally what the patterns provide for a developer of an application. It illustrates the benefits of a component-based development approach over a 'normal' approach, using the example of an Internet shopping system developed using EJBs. This part is written as a dialogue between a component 'newbie' and an experienced consultant. It shows how the patterns influence the day-to-day work of an EJB developer.

The structure of the book

This book contains patterns about server-side components. It has three main parts, the purposes of which are outlined below:

- *Foundations* provides an introduction for the book. It first defines the term *component* in the context of this book and distinguishes it from other kinds of components. Secondly, it introduces patterns and pattern languages and explains how they are used in the book.

  Thirdly, it includes four *principles*. Every technology is based on a set of principles – guidelines the developers of the technology had in mind when they designed it. It is important to understand these principles to understand why the technology is as it is. This is also true for component architectures.
- Part I, *A Server Component Patterns Language*, describes a pattern language that 'generates' a server-side component architecture. Because a fundamental principle of patterns is that they are proven solutions, known uses are important. We use the three most important component models as examples, namely Enterprise JavaBeans (EJB), CORBA Components (CCM) and COM+. These examples are just conceptual in nature, and introduced only briefly.

- Part II, *The Patterns Illustrated with EJB*, presents extensive examples of the patterns in Part I using the Enterprise Java Beans technology. It illustrates how the patterns have been applied in this case, thereby explaining the EJB architecture. This part contains EJB source code and UML diagrams.
- Part III, *A Story*, contains a dialogue between two people who discuss the design and implementation of an e-commerce application based on EJB. This provides another way of looking at the patterns.

Example technologies

Patterns can only be considered patterns if they have actually been applied or used. The usual process of pattern writing therefore starts by 'finding' relevant patterns in concrete systems and abstracting the core pattern from such uses.

In the case of the technical patterns in Part I of this book, these concrete systems are the three most popular server-side component architectures, Enterprise Java Beans, CORBA Components and COM+. This section provides a very brief introduction to these technologies and provides references to further reading.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a server-side component architecture defined by Sun Microsystems. It targets enterprise business solutions, with a strong focus on web-based applications and integration with legacy systems. EJB is part of a larger server-side architecture called the Java 2 Enterprise Edition (J2EE).

EJB and J2EE are all-Java solutions, although it is possible to access EJB components (*Beans*[1]) from other programming languages, because a mapping to CORBA and IIOP, CORBA's TCP/IP-based transport protocol, is defined. EJB is not an official standard, it is Sun's property. However, many other companies have taken part in the creation of the architecture and many implementations of the standard exist on the

1. In the context of this book, *Bean* always denotes an Enterprise JavaBean component, not a normal JavaBean.

market, from open source servers to very expensive commercial applications.

A good introduction to EJB is Monson-Haefel's *Enterprise Java Beans* [MH00]. Sun's EJB Specification [SUNEJB] is also worth reading. *Java Server Programming* [SU01] gives a broader look at J2EE, including EJB.

The current version of EJB is 2.0. However, EJB 2.0 did not introduce many new features from the viewpoint of pattern language. It does improve significantly in some areas, mainly in persistence and local access to components using Local Interfaces. But from a conceptual point of view not much has changed – for example, Local Interfaces are still interfaces. As a consequence, this book does not go into very much detail about these new features.

The CORBA Component Model

CORBA (Common Object Request Broker Architecture) is a standard for distributed object infrastructures, a kind of object-oriented RPC (Remote Procedure Call). The standard is defined by the Object Management Group (OMG), a non-profit organization involving many important companies within the IT industry. CORBA is an operating-system independent programming language, and implementations of the standard are provided by many vendors. In addition to low-level remote object communication, CORBA defines additional services known as *Common Object Services* (COS) that take care of transactions, asynchronous events, persistence, trading, naming, and many other less well-known functionalities.

CORBA Components form a component architecture built on top of the CORBA communication infrastructure. The specification is part of CORBA 3.0. Although the specification is finished, there are no commercial implementations available at the time of writing, but several companies have announced products and are working on CCM implementations. Nevertheless, experimental implementations are in progress and early results are available [OPENCCM].

The CORBA Component Model (CCM) shares many properties with EJB: it is actually upward-compatible, and in the future it is expected that the two architectures will merge.

As mentioned before, CCM is still quite new, therefore not much has yet been written about it. The Specification by the OMG [OMGCCM] is

of course worth reading, while Jon Siegel's CORBA 3 book [SI00] also contains a chapter on CCM.

COM+

COM+ is Microsoft's implementation of a component infrastructure. It is tightly integrated with the Windows family of operating systems and is not available for other operating systems[1]. COM+ is language-independent on the Windows platform.

COM+ is basically the new name for DCOM integrated with the Microsoft Transaction Server (MTS), which is, contrary to what its name implies, a run-time environment for components. Transactions are managed by DTC, the distributed transaction coordinator. DCOM itself is a distributed version of Microsoft's desktop component technology COM (Component Object Model). So, in contrast to the other example technologies, COM+ has a relatively long history. On one hand this means it is a proven technology, on the other, it has some rather odd features for reasons of compatibility.

The book market is full of titles about this topic. We found Alan Gordon's COM+ Primer [GO00] to be a good introduction.

Book companion web site

Using a book as a means to communicate knowledge is not without liabilities.

There are two main drawbacks:

- Books take a very long time to prepare. This means that by the time the book is published, some of the information in the book might be outdated, or at least incomplete.
- A book is a one-way vehicle. There are no direct means by which readers can give us feedback or start a discussion.

To overcome these deficiencies, we have created a web site at **www.servercomponentpatterns.org**. It contains source code for the book, links to interesting web sites, a means to contact the authors and

1. COM and DCOM have been ported to Unix by Software AG under the name entireX. To our knowledge, COM+ has not been ported yet, and we don't know of any attempts to do so, currently.

other useful stuff. We will also use the site to publish errata and newly-mined patterns. Take a look at it.

Contacting the authors

We'd like to receive your opinions on this book or any other comments you might have.

The authors can be reached at

- voelter@acm.org
- alex.schmid@t-online.de
- ewolff@mac.com
- As a group, at scp-book@yahoogroups.de

Stefan Schulz, the artist who drew the cartoons, can be reached at sn.schulz@gmx.de.

The cover picture

It is not easy to design a cover for a book. In our case the problem was simplified, because we could use the design and layout of the series, so we only had to find an inset picture to use in the top right corner of the cover. After long discussions we decided to use a picture of a city. Why? Because a city provides an environment, an infrastructure for buildings, just as a container provides an infrastructure for components. In both cases you have to provide or implement some standardized interfaces – in a city it is providing connections for the water and plugs for the power supply lines. Maybe you can find further analogies when reading through the book.

The cartoons

While writing the book, we made a presentation of its patterns at a conference. We wanted to make this entertaining, so we asked a freelance consultant who works for our company whether he could draw some cartoons to illustrate it, one for each pattern. He did, and they were a huge success. We then convinced him that he should draw a cartoon for each pattern in the book.

We think that these aid remembering the patterns because they capture the essence of each pattern in a single illustration. In addition, they make a huge difference to the book in general. We hope you like them as much as we do. You can reach the artist Stefan 'Schulzki' Schulz at sn.schulz@gmx.de.

Notations and conventions

As with any other book, this book uses a set of notations and conventions. We think it is helpful to understand these conventions in advance.

Normal text

For normal text, we use Book Antiqua 11 point. Patterns use special formatting, using bold font for problem and solution sections. References to other patterns in the book are made using the pattern name in SMALL CAPS. References to patterns outside this book are written in *italics*, together with the [Reference] to where the pattern can be found.

Variable references in source code or any other names, tags, or types are placed in *italics* to help differentiate them from normal text and to avoid ambiguities (for example: 'In line 3 you can find the variable *name*' is different from 'this is the variable name').

Source code

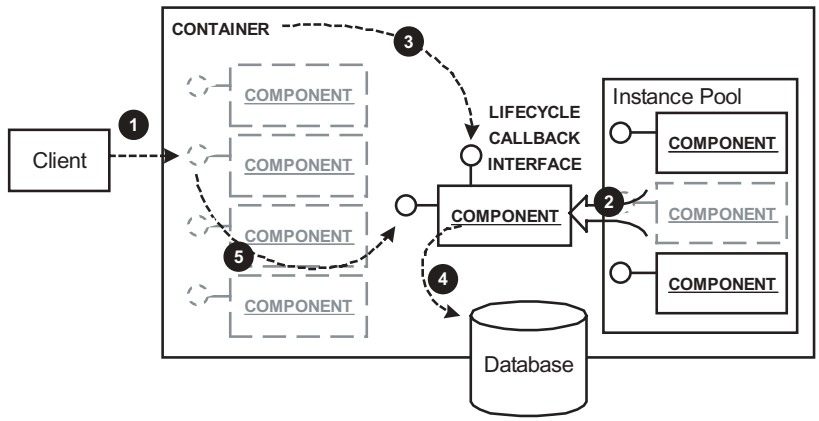
For source code, we use fixed-width fonts. We do not claim that the source code in the examples is complete and can be compiled. For reasons of brevity, we leave out any unnecessary detail. Such detail can be import statements, error handling and any other code we don't consider essential – we use the ellipsis (…) to mark such missing code. We also use comments or pseudocode to simplify the code examples.

Diagrams

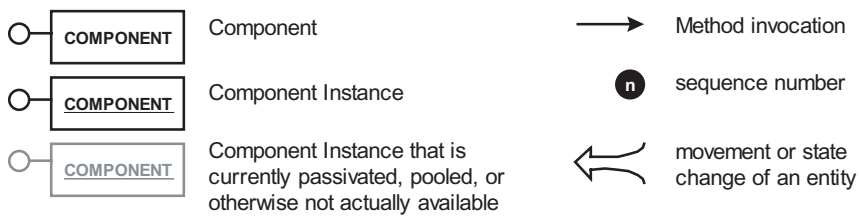
Diagrams are a bit more complex, because we use different notations throughout the book. For structural diagrams, we use UML whenever applicable. For example, a component is usually rendered as a 'class

rectangle' with attached interfaces. For example, the following illustration shows a Component A with two interfaces, IF_{A1} and IF_{A2}.

For many of the structural diagrams, especially in Part I, we use non-UML notations, because UML does not provide very much help here, and forcing everything to be UML by using stereotypes and comments does not help either. The next illustration shows a typical example:



This notation requires some explanation. Basically, the illustrations show UML-like collaboration diagrams, albeit with aggregation shown by real containment and other details. The following legend explains the symbols used – the rest will become clear when you look at the illustrations in the context of their patterns.



To illustrate interactions, we usually use UML sequence diagrams.

Components and instances

To avoid misunderstanding, we have to clarify an important issue about our 'component-speak'. In our terminology, a *component* is the artifact that is developed and deployed, whereas a *component instance* (or *instance* for short) denotes a specific run-time instantiation of the component. Compared to OO terminology, a component corresponds to a *class*, and a component instance corresponds to an *object* (which is called a *class instance* in some languages). In the EJB-specific parts we use *Bean* and *Bean instance* respectively.

Acknowledgements

As with any other book, there are more people involved in its creation than are mentioned on the cover. This even more true for a patterns book. So we'd like to say "thanks" to the following people.

First, we'd like thank Frank Buschmann, who played a double role in the creation of the book. Firstly, he is the series editor of the Wiley patterns series and also approached us to write the book. Secondly, he was our shepherd, supplying us with many useful comments – being an experienced pattern and book author, and understanding the technology we cover in the book, his comments were really helpful.

Second, we'd like to thank our employer, MATHEMA AG, and particularly Hans-Joachim Heins and Michael Wiedeking, for their support of the project.

The patterns in Part I have been (partly) submitted to the EuroPLoP 2001 conference. We received many useful comments from the workshop there, but we'd like to thank particularly Klaus Marquardt, who was our shepherd at the conference and who provided many useful comments on early versions of the patterns.

Of course, Stefan Schulz deserves special thanks, because without his cartoons this book would definitely not be the same. We pressurized him rather highly to meet the book's deadlines, and we're really happy that it worked out.

We'd also like to thank Peter Sommerlad, who, playing Devil's Advocate, challenged us with several critical questions about component-based development in general. Also thanks to Matjaz Juric, who

reviewed the EJB-specific parts rather late in the process and found several glitches and issues, especially regarding the then brand-new EJB 2.0.

We have received comments from many other people, and we'd also like to thank them for their work. These include Jim Coplien, Costin Cozianu, Kristijan Cvetkovic, Jutta Eckstein, Erwin Hoesch, Matthias Hessler, Markus Schiller, Michael Kircher, Jan Materne, Markus Späth, Thomas Neumann, Francis Pouatcha, Stephan Preis, Michael Schneider and Oliver Vogel.

Very special thanks go to Steve Rickaby of WordMongers for the copy-editing and typesetting of the book. Steve did a wonderful job of adjusting language, formatting and layout – after working with the material for a really long time, we would not have had the patience to do that ourselves. Thank you very much!

Last but not least, we'd like to thank Gaynor Redvers-Mutton of John Wiley & Sons, who helped us through all the organizational 'stuff' and the coordination with Wiley's. After all, you don't write a book every day, especially your first one, so it was a great help to have somebody who really cared for all the technical and administrative issues.

Alexander Schmid would like to thank Gudrun Hauk for her patience and support. His contribution to this book wouldn't have been the same without her encouragement.

Eberhard Wolff would like to thank Tanja Maritzen and his family for the support and backup. Without their support and patience, his work for this book would have never been possible.

Markus Völter
Alexander Schmid
Eberhard Wolff

Foreword *by Frank Buschmann*

The book you are holding is about a very important and 'hot' software engineering topic: components. However, it is not 'yet another component book', of which you can find so many on bookshelves these days. This book is the fourth volume of the Wiley series on software design patterns: as such you can expect something special and unique from it.

What makes this book unique in the field of component-based software development is that it intentionally avoids focusing on the presentation and explanation of a particular component platform and its APIs. Instead it focuses on mining the common wisdom that underlies the design and implementation of every successful component, regardless of platform or container. Documented as patterns, this wisdom is presented in a readily-usable form for everybody interested in component-based development.

You can use these patterns, for example, to learn why components and containers are designed and implemented as they are, so that you can understand their underlying vision. Or, with help of examples from the CORBA Component Model (CCM), the Microsoft Component Object Model (COM+), and specifically with the extensive and detailed examples from the Enterprise JavaBeans (EJB) environment in Part II, you can see how these platforms work and why they work as they do. This enables you to use them more effectively.

The 'big picture', which forms the essential core of the component idea, is often not visible explicitly in books that are about only one of these environments. Even if the full picture is presented in such books, it is often buried deep inside endless descriptions of container-specific implementation and API details. In this book the component vision is presented in expressive, bite-sized chunks of information. It therefore complements every other book about components, whether it be a general survey of the discipline, a reference or a programming guide for a specific container or component platform.

The patterns in this book can also, of course, help you to build components that actually provide the benefits that component-based development promises: better modularity, adaptability, configurability and reusability. Even if it does not appear so in theory, in practice this is a challenging and non-trivial task. Not many books provide you with useful guidelines for mastering these challenges – this book does.

The many examples from all major component platforms demonstrate that the patterns in this book are not just about component theory, but also capture the wisdom that experienced practitioners apply when building component-based systems. My personal favorite is Part III. This presents a dialogue between a component novice who wants to use EJB in a new project and an experienced consultant who, with help of the patterns presented in this book, demonstrates to the novice how he developed an EJB-based e-commerce application. The novice asks all those questions that every 'newbie' to components has in their minds, but which are seldom answered in the contemporary body of component literature. Here they are.

The greatest value of this book is, however, less obvious: you can use it to design and implement your own custom component platform. In the presence of CCM, COM+, EJB, and other common 'off the shelf' component environments, this may sound silly at a first glance. However, I mean it very seriously!

Consider, for example, an embedded or mobile system. Such a system endures stringent constraints: limited memory, CPU power, specialized hardware are just a few. Yet it should benefit from components, as enterprise systems already do. For some systems, such as applications for mobile phones, designing with components is the only way to ensure a reasonable profit. Unfortunately, available component environments are either too heavyweight for today's embedded and mobile systems, or are not available at all for the hardware and operating system employed. Consequently, you can be forced to build your own platform and container tuned to the specific requirements and constraints of the system under development.

Even for enterprise systems, the traditional playing field for component platforms such as CCM, COM+, and EJB, a trend towards custom containers is beginning to emerge. Developers complain that the containers available 'off the shelf' either do not provide what they

need, or implement the required functionality in a way they cannot use. As a result, developers may need to implement their own containers, which requires relevant and appropriate guidance. They can find such guidance in this book.

The trend towards custom containers for enterprise systems is not only fuelled by frustrated developers, however. Academia and industrial research have already initiated discussions and research on the look-and-feel of the next generation of containers. These will be very different from the 'one-size-fits-all' philosophy of the containers of today. Instead, new approaches require the assembly, integration and customization of small building blocks to create a component platform that is optimally tuned for its specific application. This requires deep knowledge and understanding about components and how they work. Yet despite the differences between the containers of today and those of tomorrow, their underlying ideas will still be the same, as presented by the patterns in this book. This book is therefore timeless and should survive on your bookshelf while the whole component world around you changes.

I hope you enjoy reading this book as much as I did.

Frank Buschmann
Siemens AG, Corporate Technology

Foreword *by Clemens Szyperski*

Composing new systems from components, and decomposing architectures and designs to enable components, are at the same time both the greatest promises and the greatest challenges in modern enterprise application engineering. Three-tier approaches have become best practice for many such applications, and application-specific complexity is often concentrated at the application-server tier. Component technologies focusing on application servers are reaching maturity and it is thus mandatory to wonder how to best map future enterprise applications to this form. This book promises to provide guidance here – guidance on how to use best component and application server technologies to construct enterprise applications.

Combining the words 'server', 'component' and 'pattern' into a single book title is already a feat. Going further and contributing useful patterns that actually help harness much of the complexity and subtlety of component technologies for server-side (mostly middle tier) systems is a significant contribution. This book delivers on its title and will help architects and designers of enterprise solutions get more out of the powerful middle-tier component technologies we see today.

The authors present a pattern language targeting three audience groups: architects, component implementers and container implementers. With a total of thirty patterns the pattern language is quite rich, covering the ground from the fundamental (component, component interface, component implementation, container, deployment) through subtle detail (identity, persistence, lifetime, remoting, packaging) to the almost esoteric (container implementation).

While some top-level discussion of the CORBA Component Model (CCM) and Microsoft COM+ is presented, the bulk of the text focuses on Enterprise JavaBeans (EJB). Covering the entire pattern language, which itself is more technology-neutral, the authors explain in detail how the pattern language maps to the EJB architecture and design. The mapping begins with the detailed EJB landscape of Stateful and State-

less Session Beans on one hand and Persistent Entity Beans on the other. Zooming in, the authors then describe how to implement such EJB components in some detail, including lifecycle and Deployment Descriptor issues. A discussion of the client side helps an understanding of aspects of remote access, as well an appreciation of the difference from local access, as supported as of EJB 2.0. Concrete advice on design trade-offs affecting performance or scalability is given.

The discussion of container implementation issues is possibly less relevant to most readers as such. However, taking the time to explore the innards of a container actually helps develop a more solid understanding. It is the first duty of any engineer to stay grounded – to understand enough of what's happening in the platform to make educated decisions.

The authors carefully present technical detail where it helps understanding, but avoid drowning the reader in endless tedious detail. Despite the sometimes daunting nature of server-side component technology, the book remains refreshingly readable and easy-going. Especially as a stepping-stone to the deeper literature on various subtopics, this book provides a helpful overview.

Clemens Szyperski
Redmond, March 2002

Foundations

Before actually delving into the ins and outs of patterns, a set of foundations has to be put in place. These foundations include the following:

- A definition of the term *component*. The word has been used in many contexts, and we must make sure that you have the same understanding of the term as we had when we wrote the book.
- Pattern and pattern languages are introduced. While it is not necessary to be a pattern expert to read this book, some background on the purpose of patterns and their form, as well as on the structure of pattern languages, is very useful.
- Last but not least, we want to introduce a set of *principles*. These principles underlie the design of the mainstream component architectures and serve as guidelines for their structure. Understanding these principles helps significantly in understanding component architectures.

What is a component?

Many definitions of the term *component* exist, and even more informal (ab-)uses of the term are common today. We do not attempt to give yet another definition for the term – however, we want to make clear what we understand by the term in the context of this book.

A component definition

Clemens Szyperski [SZ99] proposed a definition for *component*. We will use it as a starting point for further discussion:

> *A software component is a unit of composition with contractually-specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Let's consider some parts of this definition in detail:

- *A unit of composition*. Calling a component a unit of composition actually means that the purpose of components is to be composed together with other components. A component-based application is thus assembled from a set of components.
- *Contractually-specified interfaces*. To be able to compose components into applications, each component must provide one or more interfaces. These interfaces form a contract between the component and its environment. The interface clearly defines the services the component provides – it defines its responsibilities.
- *Explicit context dependencies only*. Software usually depends on a specific context, such as the availability of database connections or other system resources. One particularly interesting context is the set of other components that must be available for a specific component to collaborate with. To support the *composability* of components, such dependencies must be explicitly specified.
- *Can be deployed independently*. A component is self-contained. Changes to the implementation of a component do not require changes (or a reinstallation) to other components. Of course, this is only true as long as the interface remains compatible[1].
- *Third parties*. The engineers who assemble applications from components are not necessarily the same as those who created the components. Components are intended to be reused – the goal is a kind of component marketplace in which people buy components and use them to compose their own applications.

This definition of the term *component* is very generic, and thus it is not surprising that the term is used to mean rather different concepts. While this is acceptable in general, we must make sure that we – meaning you, the reader and us, the authors – have the same understanding of the term in the context of the book. The following are examples of uses of the term *component*.

---

1. The meaning of *compatible* in this context is not completely agreed upon. In general it means that if interface A is compatible with interface B, then A as to provide the same set of operations as B, or more. Whether this requires some form of explicit subtyping depends on the component architecture.
   Note that in current practice the semantics of operations is not specified in interfaces - so no guarantees can be made regarding semantic interface compatibility.