Server Component Patterns

Server Component Patterns

Component Infrastructures Illustrated with EJB

Markus Völter, Alexander Schmid and Eberhard Wolff

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

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 II 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 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.

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

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

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].

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.

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.

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 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.

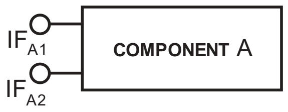
Variable references in source code or any other names, tags, or types are placed in *italics* to help differentiate them from normal text and to avoid ambiguities (for example: 'In line 3 you can find the variable *name*' is different from 'this is the variable name').

Source code

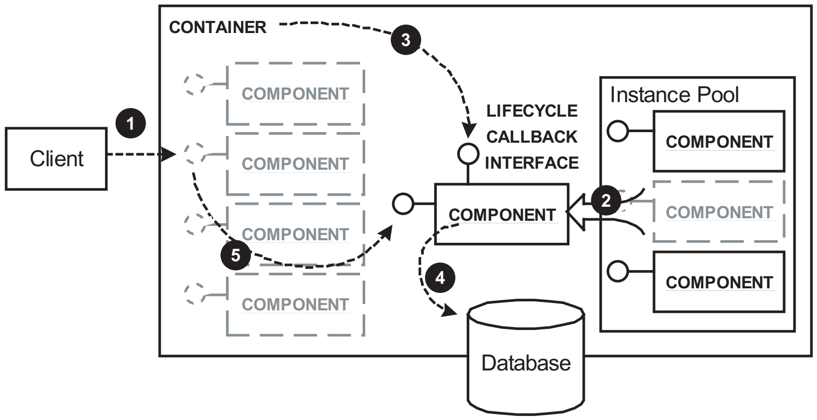
For source code, we use fixed-width fonts. We do not claim that the source code in the examples is complete and can be compiled. For reasons of brevity, we leave out any unnecessary detail. Such detail can be import statements, error handling and any other code we don't consider essential – we use the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

Diagrams

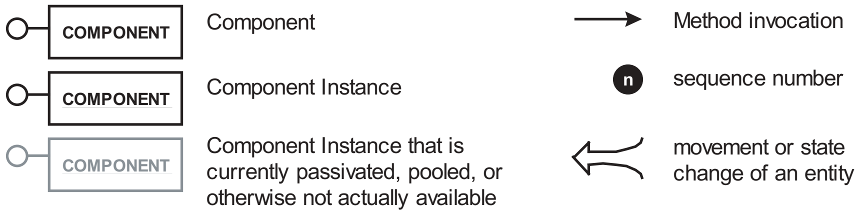
Diagrams are a bit more complex, because we use different notations throughout the book. For structural diagrams, we use UML whenever applicable. For example, a component is usually rendered as a 'class rectangle' with attached interfaces. For example, the following illustration shows a Component A with 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.


COMPONENT
Component
COMPONENT
Component Instance
COMPONENT
Component Instance that is currently passivated, pooled, or otherwise not actually available
Method invocation
n
sequence number
movement or state change of an entity

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 Stateless 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 under-

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.

standing of the term in the context of the book. The following are examples of uses of the term *component*.

- *Windows-DLLs*. DLLs in the Windows world are often termed 'components'. While this fits Szyperski's definition, it does not fit all the additional aspects we will define for components below. We see a DLL more as a plug-in [MA00], not as a component.
- *RAD components*. The first widespread use of the term *component* was in the context of GUI 'building blocks' in the IDEs of Rapid Application Development (RAD) tools. Smalltalk was the first environment in which these kinds of components (or *controls*, or *widgets*) were used, and they became really popular with Microsoft's Visual Basic Controls (VBX). Their main focus was to facilitate reuse in graphical user interfaces.

  These components can operate in two 'modes': at design time, the programmer can set properties by using the IDE. These properties, the design-time state, is then stored and can be used for initialization later at run-time. This concept has found widespread use in many IDEs, among them subsequent version of Visual Basic (OCX, ActiveX), Visual C++ (OCX, ActiveX), Borland's Delphi, C++ Builder and Kylix (VCL, CLX), and many others. The concept was extended later to include GUI components with no visual appearance.
- *Distributed Objects*. Distributed Objects, as introduced by CORBA, RMI and DCOM, also exhibit many of the characteristics of components. They have an explicit interface, allowing their implementation to be changed. They also encapsulate a specific functionality (as any object does), and they are usually used with many other distributed objects in the context of distributed applications. However, they do not require a container to run in, which means that the principle of *Separation of Concerns* (see page 14) does not apply here.

While all these uses of the term *component* are valid, this is not what we understand by the term in the context of this book. To clarify things, let's add additional properties to the definition:

- *A component is coarse-grained*. In contrast to a programming language class, a component has a much larger granularity and therefore usually more responsibilities. Internally, a component can be made up of classes, or, if no OO language is used, can be made up of any other suitable constructs. Component-based development and OO are not technically related.
- *They require a run-time environment*. Components cannot exist on their own, they require a 'something' that provides them with necessary services (see *Separation of Concerns* principle on page 14).
- *Remotely accessible*. To support distributed, component-based applications, components need to be remotely accessible.

This definition, or rather this set of properties, of components fits the so-called *distributed*, or *server-side*, components, as exemplified by Enterprise JavaBeans (EJB), CORBA Components (CCM) and Microsoft's COM+. The book focuses exclusively on this kind of component.

Distributed components can be seen as an extension of distributed objects. Technically this is correct – CCM makes this quite obvious. Many patterns have been published for distributed object computing, most notably those in the second volume of Pattern Oriented Software Architecture by Schmidt, Stal, Rohnert, and Buschmann ([SSRB00]). While these patterns can be used to implement an efficient CONTAINER and a COMPONENT BUS, they are at a level below the one that this book covers. If you really need to implement your own component architecture or want to create your own container for EJB or CCM, be sure to read these patterns.

There is another concept we need to mention here to define the relationship to the content of this book: *business components.* While the distributed components introduced above provide several benefits (especially because of the separation of concerns), they are still too small to structure large enterprise applications. We need even larger entities. This is where business components come into play: business components are subsystems of an application that fulfil a specific, usually quite complex task. In today's IT landscape, they typically consist internally of many distributed components. Business components have a wider scope, going far beyond technical considerations, they also influence management, the development process, project organization et cetera [HS00].

Patterns and pattern languages

Over the past couple of years, patterns have become part of the mainstream of object-oriented software development. They appear in different kinds and forms.

The most popular patterns are those for software design, pioneered by the Gang-of-Four book [GHJV94] 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 [BMRSS96], [SSRB00], analysis [FO96] and even non-IT topics such as organizational or pedagogical patterns [PPP], [FV00]. These kinds of patterns are not so well-known, mostly because fewer people are faced with the problems addressed by them. Some of the patterns are even domain-specific, and therefore naturally have a narrower focus and a smaller audience.

Classification of the patterns in this book

The patterns in this book can be divided into *architectural* patterns and *design* patterns. It is not easy to draw the line between architecture and design, and often the distinction is purpose-built for a specific situation. For a rough distinction, let's refer to the definition of software architecture from Bass, Clements, and Kazman [BCK98]:

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

The important aspect of this definition is the term *externally-visible,* because this defines a natural distinction from design: design deals with the *internal* structures.

Some of the patterns in the first section of this book are architectural in nature, because they describe the externally-visible aspects of a server-side component architecture. Examples are COMPONENT or CONTAINER. Some other patterns should be classed as design patterns, because they document internal structures of such an architecture, for example COMPONENT PROXY or LIFECYCLE CALLBACK.

What is a Pattern?

A pattern, according to the original definition of Alexander[2] [AL77], 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 *pattern*. It is probably a bad idea to cite Alexander in this way, because he explains this definition extensively. In particular, how can we distinguish a pattern from a simple recipe? Take a look at the following example:

Context	You are driving a car.
Problem	The traffic lights are red.
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 [HILL] proposes another, slightly longer definition:

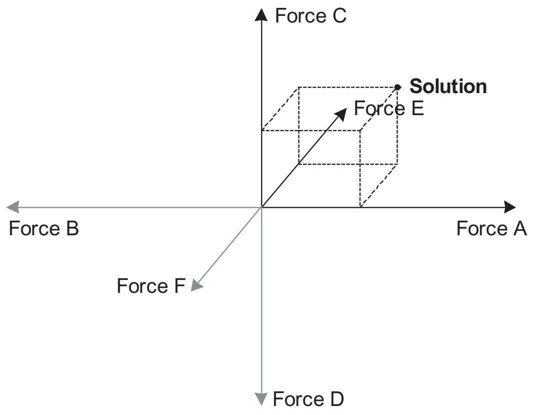
> *Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.*

He mentions forces. Forces are considerations that somehow constrain or influence the solution of a pattern. The set of forces

2. In his book, *A Pattern Language – Towns • Buildings • Construction* [AL77] Christopher Alexander presents a pattern language 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.

builds up *tension*, usually formulated concisely as a problem statement. A solution for this problem has to balance the forces somehow, because usually the forces cannot all be resolved optimally – a compromise has to be found.

The pattern, in order to be understandable by the reader, 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 several concrete problems. However, the solution should be concrete enough to be really useful, and it should include a certain 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. Note that a pattern is not merely a set of UML diagrams.

Patterns are never 'new ideas'. Patterns are *proven* solutions to recurring problems. So known uses for a pattern must always exist – at least three. In software patterns, this means that systems must exist that are implemented according to the pattern. The usual approach to writing patterns is not to invent them from scratch – instead they are discovered in, and then extracted from, real-life systems. These systems then serve as known uses for the pattern. To find patterns in software systems, the pattern author has to abstract the problem/ solution pair from the concrete instances found in the systems at hand. Abstracting the pattern while preserving comprehensibility and practicality is the challenge of pattern writing.

There is another aspect of what makes a (good) pattern, the *quality without a name* (QWAN). The quality without a name cannot easily be described, the best approximation is *universally-recognizable aesthetic beauty and order*. So, a pattern's solution must somehow appeal to the aesthetic sense of the pattern reader – in our case, to the software developer. While there is no universal definition of beauty, there certainly are some guidelines as to what is a good solution and what is not. For example, a software system should be efficient, flexible, easily understandable while addressing a complex problem, et cetera. The principle of beauty is an important guide for judging whether a technological design is good or bad. David Gelernter details this in his book Machine Beauty [GE99].

From Patterns to Pattern Languages

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

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

- *Compound patterns* (also known as *composite patterns*) are patterns that are assembled from other, smaller patterns. These smaller patterns are usually already well known in the community. To form a useful compound pattern, it is not enough simply to combine a set of other patterns and see what happens. It is essential that the newly-created compound pattern actually solves a distinct problem, and not just a combination of the problems of its contained patterns. A compound pattern also resolves its own set of forces. An example for a compound pattern is *Bureaucracy* by Dirk Riehle [RI98].

- A *family of patterns* is a collection of patterns that solves the same general problem. Each pattern either defines the problem more specifically, or resolves the common forces in a different way. For example, different solutions could focus flexibility, performance or simplicity. Usually each of the patterns also has different consequences. A family therefore describes a problem and *several* proven solutions. The reader has to select the appropriate solution, taking into account how he might want to resolve the common forces. A good example is James Noble's Basic Relationship Patterns [NO97].
- A *collection,* or *system of patterns* comprises several patterns from the same domain or problem area. Each pattern stands on its own, sometimes referring to other patterns in the collection in its implementation. The patterns form a system because they can be used by a developer working in a specific domain, each pattern resolving a distinct problem the developer might come across during his work. A good example is Pattern Oriented Software Architecture by Buschmann, Meunier, Rohnert, Sommerlad, and Stal (also known as POSA – see [BMRSS96]).

The most complex form of combining patterns is a *pattern language*. There are several characteristics of pattern languages:

- A pattern language has a language-wide goal. The purpose of the language is to guide the user to reach this goal. The patterns in a pattern language are not necessarily useful in isolation. In Part I the goal is the definition of a component architecture.
- A pattern language is *generative* in nature. Applying the pattern language generates a whole, the goal of the language. This generated whole should be 'beautiful' in the sense of QWAN. The component architecture generated by the pattern language in Part I is probably not perfectly beautiful, but it resembles the best practices currently in use.
- To generate the whole, the pattern language has to be applied in a specific order. This order is defined by one or more sequences. The sequence for the patterns in Part I can be found at the beginning of Part I.

- Because the patterns must be applied in a specific sequence, each pattern must define its place in this sequence. To achieve this, each pattern has a section called its *context*, which mentions earlier patterns that have to be implemented before the current pattern can be implemented successfully. Each pattern also features a *resulting context* that describes how to progress. It contains references to the patterns that can be used to help in the implementation of the current pattern.

Pattern languages are the most powerful form of patterns, because they do not only specify solution to specific problems, but also describe a way to create the whole, the overall goal of the pattern language.

The Pattern form used in this book

There are probably few disciplines in technology and science where the form of a document is as important as in the patterns community. There are 'religious wars' over whether a specific form is useful, nice, elegant or not. In the patterns community, *form* describes the way in which the pattern is structured. The GoF book introduced a form which consists of thirteen sections, each beginning with a heading [GHJV94]. This form has several advantages, particularly, because it structures the document really well and facilitates cross-referencing and scanning. This form is well suited to complex patterns. There are other pattern forms, such as the POSA form, which used the GoF form as a basis. It addresses some of the problems that have occurred while the community used the GoF form, and for example contains the forces more explicitly. The POSA form is still a very strictly-structured form, well suited to the detailed description of complex patterns.

However, there are also forms that involve less ceremony. For example, the SEMINARS pattern language [FV00] uses a simplified Alexandrian form, consisting only of problem with forces, solution and examples.

Most of the patterns in this book are relatively small and simple, conceptually. They can therefore be described in a more concise, less structured form. In our book, we use the form introduced by Alexander in his classical book on patterns in architecture [AL77]. This form tries to produce a pattern description which is nearer to prose. This makes the patterns more readable, which is beneficial for pattern languages. The Alexandrian pattern form also consists of several sections separated by delimiters and/or specific usages of font attributes. Let's look at each of the sections:

A pattern begins with a **name**. As in any other pattern form, the name is critical to the pattern. The name has several important tasks to fulfil:

- It describes the content of the pattern in a few words. A good pattern name gives a hint to the solution in the pattern, and after reading and understanding the pattern, the name becomes 'obvious' to the reader
- The name is used within the pattern language to refer to the pattern. To make such references easier, it should be possible to embed the name in a sentence, for example, 'a CONTAINER can use several different strategies to DISTINGUISH IDENTITIES.
- The name(s) of the pattern(s) form a vocabulary for the problem domain. So, the names of the patterns within a language should be similar in style and length, in order to form a homogeneous 'feeling'.

The next section after the pattern name is the pattern's **context**. As described above, the context helps to define the sequence through the pattern language. It names the patterns that must have been implemented in order to implement the current pattern.

The next section, delimited by three stars (❊ ❊ ❊) describes the **problem** the pattern resolves. This description should be concise and general. The section should also mention the conflicting forces that have to be balanced in the solution. The problem statement is printed in bold font.

After the problem, non-bold, is the pattern's **body**. It contains additional information to further illustrate the problem. Other contents of the body include a simple example, and, sometimes rationales that are important to realize fully the problem described previously.

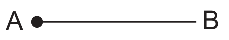
Then, after the word 'Therefore', the **solution** to the problem is given, again in bold. The solution is kept rather short and describes what you have to do to solve the problem stated previously. A discussion follows the solution, including the consequences of applying the patterns, advantages and drawbacks, and sometimes a rationale why other possible solutions should not be used.

The **resulting context** follows after another three stars. This describes 'where to go from here'. It contains references to patterns that can be used to help in implementing the current pattern.

The last section gives **example** uses of the pattern. The concrete form of the examples differs among the different sections in the book. We go into details below.

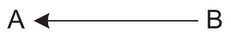
Relationships between the patterns

We mentioned previously that a pattern language takes its power mainly from the way in which the patterns are related to each other: we have identified three distinct relationship types. The following table shows their semantics and the graphical notation used in the language maps:



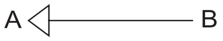
A provides context for B

As in Alexander's patterns, pattern A provides the context for pattern B. That means that once you have implemented pattern A, you can also implement B. However, B is not necessary for A to work.



B is required to make A work

If you want to apply pattern A, you must also implement pattern B, because A does not work otherwise. However, B can also be used in other contexts, which is why it is a separate pattern from A. Usually, A is a higher-level pattern than B, B helps to solve a particular aspect of A.



B is a specialization of A

B is a specific form of A, that is, it usually has a more specific context than A. It provides a specific adaptation of A for certain circumstances.

Principles for component architectures

The design and creation of every software architecture is guided by a set of principles. These can be seen as high-level goals or guidelines. Understanding and accepting these principles is crucial to understanding the architecture, because they guide architectural decisions. *Information Hiding* is such a principle, important for understanding object-oriented programming.

Component architectures are no exception, as they also build on a set of principles. We consider the following principles to be important: *Separation of Concerns*, *Multitier Systems*, *Functional Variability* and *Development Roles*.

You might ask what the actual difference between patterns and principles is. It is the level of abstraction, or concreteness. The principles in this book are far more abstract than the patterns that follow. However, principles also solve a specific problem in a certain context. So the principles can be patterns in another setting. For example, take the MULTITIER ARCHITECTURE principle. It can be seen as an extension of the *Layers* pattern described in POSA. However, in this book, it is such a fundamental building block that it is presented as a principle. Another way of distinguishing patterns from principles is the following observation: *patterns define a specific software configuration and set of interactions*. It is something you can *create*. For example, SEPARATION OF CONCERNS is more like a general mind-set – there are many ways in which this can be realized.

Principles serve as a kind of guideline, or rationale, for the structure of specific patterns. Many patterns will refer to the principles to explain why a pattern is as it is.

Separation of Concerns

In today's rapidly-changing world, in which businesses and their processes are evolving rapidly, it is crucial for a business to adapt its software systems to support these changing processes. Business systems are so complex and expensive that it is vital to preserve the investments once they have been made. This is even true when the business and the requirements are changing significantly. The primary goal when building mission-critical applications is therefore to ensure they can evolve to adapt to changing requirements.

There is another kind of change to be accommodated: technology. Every couple of months, new technological possibilities arise, and usually businesses want to use them. The reasons are many:

- *Real advantages*. There might be real advantages of using new technologies, so adopting them results in a more efficient system, less cost, or other measurable advantages.
- *Compatibility/Integration*. A system might need to retain compatibility with other systems that have been migrated to a new technology.
- *'Early Bird'*. A specific technology has been adapted (too) early, and must now be changed because it does not perform well, has not become an industry standard, or the vendor does not support it any more.
- *Non-functional requirements*. Your requirements require specific levels of performance, availability, or fail-over change, and your current technology cannot provide it.
- *Hype*. New technologies are adapted because of over-enthusiasm. Decision makers decide that they need to be 'up-to-date' with their software system.
- *Developers*. A new technology is quite often adopted because the developers want to improve their résumé.

To make sure that changes can be introduced into a system with a reasonable amount of work, changes should be localized, and a change in requirements should ideally result in a change at only one place in the system. The changes that can happen to a system can be grouped: you can easily see that changes to the business logic and changes to technology have different reasons and constraints. You want to ensure that changes to one aspect do not require changes to other aspects. This principle is called *Separation of Concerns*. We can identify two fundamentally different concerns in typical business software: *functional concerns* and *technical concerns*.

Functional concerns

Functional concerns are the realization of the business requirements in a software system. Another name for functional concerns is *business logic*. They are usually described by domain experts in a requirements document. For example, look at the following requirement for an e-commerce application:

> *A customer can add products to a shopping cart by selecting the appropriate link in the catalog view. When a new product is added, the new total price of the shopping cart has to be recalculated. VAT and shipping is only added when the customer chooses to 'buy' the items in the shopping cart.*

The requirement above is subject to change, eventually. For example, the business could decide that only registered customers are allowed to actually buy their shopping cart contents. The following requirements could be added:

> *[…] Only registered customers are allowed to buy goods. If the current customer is not registered, he has to register before he can actually buy the shopping cart contents.*

These requirements could have been written by someone who does not know anything technically about how to implement an e-commerce application.

Creating a software system means implementing the functional requirements using specific software artifacts. However, this is usually not that easy, because you have to take into account additional constraints, the technical concerns. These include threading policies, memory management, transactional integrity, security, load-balancing and fail-over.

When an application is developed initially, you often don't really know about its future load, for example. It should be possible to raise the maximum load of the system, raise its availability, or add different security policies without changing the business code. You should be able to migrate to different, more powerful hardware or to run your application on a cluster.

The benefits of separating these concerns

The key to efficient software development is the separation of concerns, if this is possible (see below). This provides the following benefits:

- *Independent Evolution*. As mentioned before, because the two different concerns of the application are separated, both can evolve independently of the other – at least, to some extent. For example, a business process can easily be modified without changing the underlying transaction processing infrastructure. Alternatively, a new load-balancing algorithm can be introduced without requiring changes to the business logic.
- *Management of Complexity*. If the two concerns can be separated, this results in reduced complexity of the system as a whole, because the two concerns can be considered independently. Basically, this is a specific kind of modularization, and the goal of modularization is to make complexity manageable.
- *Separation of Roles*. The developers working on the business logic, the functional requirements, do not need to bother about technical concerns. They need not be experts in transactional programs, security et cetera – they can focus completely on their business requirements. On the other hand, technically-skilled people who know the ins and outs of transaction programming or load-balancing don't need to bother about the functional requirements (which they are usually not interested in anyway). As a consequence, developers can work on what they know best, and are thus more productive – and usually happier!
- *Reuse of technical infrastructure*. If the technical infrastructure is isolated, independent of the functional requirements, then the infrastructure can be reused in several different projects. Once

the technical infrastructure is standardized, it can even be purchased from a third-party vendor. For example, EJB is such a standardized component architecture.

As a result of separating the concerns you have two 'blobs', or problem domains, one for the functional concerns, one for the technical concerns. To make up a complete application, they have to be re-integrated. Component architectures provide this integration in a well-defined manner.

Because separation of concerns allows functional and technical concerns to be considered separately, it is very important that integration testing is done properly. Everybody should feel responsible for the success of the overall system, and some developers on the project must be dedicated to integration testing.

You may ask whether separation of concerns along the lines of technical and functional concerns is really possible. Is it really possible that the domain expert, the programmer for the functional requirements, to ignore technical concerns? And is it really possible to provide frameworks, tools and generators to provide the technical issues?

First of all, the domains in which component-based systems have been successfully used are typically business domains. They basically consist of transactional programs that modify business data. Here, separating the concerns in the way we suggest has proved to be possible. Separation of concerns does not come free – in today's implementations, some performance penalty has to be paid. However, this is acceptable in typical business systems, and can easily be compensated for by using more powerful hardware. In the embedded systems domain, however, it is not so obvious how separation of concerns, and thus component infrastructures, can be realized, because the performance penalties are usually not acceptable there. In addition, the separation of technical and functional concerns in this domain is not so well agreed. However, there are prototypes [MV02] that show how this can be done in embedded systems, generative programming [CE00] can help to overcome the performance problems.

The second part of the answer to the question concerns today's available component architectures: they do not yet reach the goal of separating the concerns completely, although they do quite well. Take EJB as an example. You can easily run an application on a cluster instead of a single machine without changing anything in your Beans implementations, provided of course that the application server you use supports clustering. This will provide better fail-over and increase the sustainable load. Of course, programmers have to stick to some guidelines to make the separation of concerns work. For example, database connections have to be acquired and released according to a specific scheme.

So, is separation of concerns really possible? We think yes, although not in all domains to the same extent – but perhaps today's architectures do not yet reach the goal completely.

There are currently several research projects that explore separation of concerns as a general means to structure software systems. Multi-Dimensional Separation of Concerns [HDSC] and aspect-oriented programming [AOP] are the most popular examples.

Multitier Architecture

Not too long ago, a piece of software was more or less self-contained: a program running on a certain operating system. Later, external resources such as databases or message queues became an increasingly important part of IT application systems. Today, the requirements for mission-critical enterprise systems are different. Some of these requirements are outlined below:

- An application has several different groups of users. Each of these groups uses the software for a different purpose. Imagine an e-commerce application. The customers browse the catalog and buy products. Naturally, they use a web-based interface (browser) for this purpose, because a simple-to-use, ubiquitous interface with no installation overhead on the client side is necessary. On the other hand, the employees of the company running the e-commerce application might want to manage orders, package them and manage invoices and cash flow. For this kind of requirement a different kind of user interface is usually required. The focus is not so much on ease-of-use – *efficient* use is much more important (ease-of-use is also important, of course, but it's not the primary concern). A Java or C++ application might be better suited than an HTML client.

To make this kind of flexibility possible, the user interface should be decoupled from the business logic itself. Remote access to the business logic is necessary.

- Scalability is an explicit requirement for many of today's applications. In Internet-based markets it is hard to predict how much load a system will have to sustain over its lifetime, because a competitive edge can arise quickly. Imagine a small e-commerce site that has 1000 visitors a day. This can easily be hosted on a single machine that runs the database, the web server and the business logic. What happens if this site is voted 'Best innovative web shop of the month' by a popular magazine? Within a couple of days the web site may have to cope with 10 or 100 times more users. The system must scale up accordingly, as this cannot be handled by one machine. As a consequence of such a requirement, the database might run on a separate host, the web server(s) could run on a cluster, and the business logic may have to be segmented according to some partitioning scheme.

  To allow for this kind of scalability and for load-balancing, it must be possible to distribute the different parts (or *layers*, see [BMRSS96]) of an application over different hosts without requiring fundamental changes to the application itself. Even distributing a single layer over several hosts should be possible.

- Availability is another important concern. Consider a business that relies heavily on its software applications, such as banking, insurance or e-commerce sites. If their system suffers unplanned down-time of several hours per year or minutes per day, the business might lose large amounts of money, or even customers.

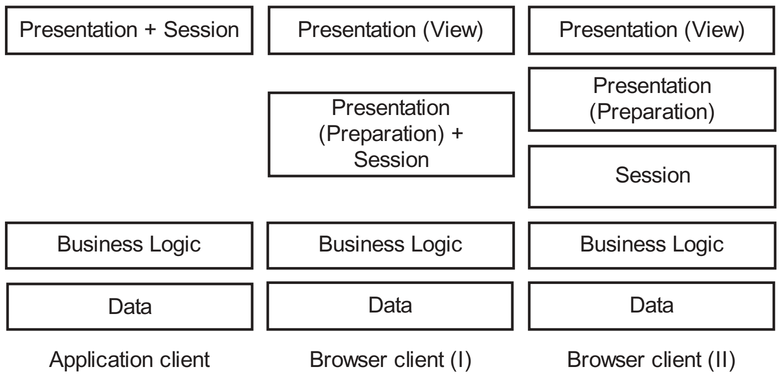
  As reliability and availability are very important concerns here, critical system components will be installed redundantly and fail-over strategies put in place.

Because hardware failure is even more critical than software failure in these kinds of applications, it is necessary to run specific services on more than one machine, once again requiring the possibility of distributing functionality over several machines.

Such requirements are not new to software systems. They have applied to military or communication systems, as well has large business systems such as airline reservation, for thirty years or more. The change is that they apply to almost every serious system today. Many more developers must be able to create systems that meet these requirements than thirty years ago.

A popular solution for many of these problems is to structure the system into *layers* (see [BMRSS96]). Each layer has a well-defined responsibility and accesses only the layers below it to implement its functionality. In the case of distributed systems, each of these layers can be placed on a separate machine and can be accessed remotely. A layer that is accessed remotely is called a *tier*.

There are many possible layering structures. The most well-known options are shown in the following illustration:

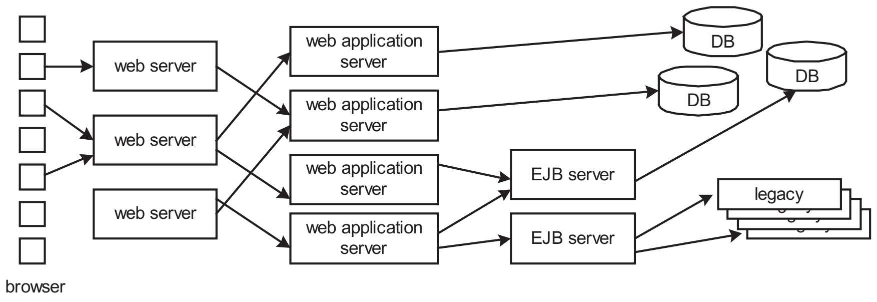


The *Application client* requires the client application to keep track of the state of the session. Only if real business logic has to be processes is the business logic layer accessed. The business logic layer itself uses the data layer for storing its persistent state.

The *Browser client* scenarios are different. A browser can only show content and collect user input, much like the form-based communication of the IBM 3270 terminals. Before the browser can display content, it has to be prepared – the HTML has to be created. This is the responsibility of the web server. In addition, the browser itself cannot keep track of the application state, therefore this state has to be managed somewhere else. Usually, as in *Browser client (I)*, this is done by the web server's session management. If this is not possible, for example because a system features several web servers, requiring the session state to be held centrally, session management can also be handled by a separate layer, as in *Browser client (II)*. This type of session management is technically usually a part of the business logic.

Process or machine boundaries can be added between any of these layers because each is remotely accessible. A typical set-up is to use one tier for the database, one for business logic, session management and presentation preparation, and a third tier for presentation (view). It is also possible to split each of the layers and distribute the parts over different machines.

What a real-life configuration actually looks like depends on the required performance, load factor, reliability et cetera. A typical installation can be seen in the following illustration:



In the illustration, the system is partitioned in several ways:

- The static HTML content is provided by 'normal' web servers.
- Dynamic content is forwarded to web application servers that directly access a database for catalog data, user preferences et cetera, usually for read-only access. They also manage sessions.

- Whenever interactive business logic is needed, the web application servers use business object servers in the next layer, which in turn access the databases or legacy systems.

This configuration ensures that the load is handled as early in the system as possible. The BO (Business Object) servers are only used when they are really needed to execute transactional business processes. This is also true for the web application servers, which provide read-only and catalogue data. As a consequence, each of the tiers can be scaled individually according to the usage profile encountered in practice.

Please note that all of the requirements described are technical in nature. The business logic programmer should not need to worry about these problems. The system architecture should take care of this. This is possible because of the principle of separation of concerns.

Functional Variability

To be able to cope with today's requirements for time-to-market and flexibility it is important to be able to reuse available software assets. The principles introduced above already help in this respect, especially separation of concerns: by separating technical concerns, these can be reused independently of the concrete business requirements and can be purchased from third-party suppliers.

But you also want to reuse your components, your functional building blocks. You do not want every department of your company to implement their own *Employee* component, for example. You must make sure that reuse is actually possible.

Two things seem important in this respect: firstly, you must make sure that your components have the right *scope*. You should never mix two functionalities in one component, because this would make it impossible to reuse each of these separately. The second problem is a bit harder, however. Once you identified *Employee* as a good scope for a component, how do you actually define it? What attributes and behavior does it need? Finding this out requires a sound analysis of the *Employee* entity. The requirements of all company departments must be taken into account, which is very hard, if not perhaps impossible.

It is even harder to take into account potential changes over time. Behavior of the entity might change, or it might have to store additional information as part of its state.

To overcome these problems we need to make the functionality of the component sufficiently variable. This means that the business logic embedded in a software system needs to be configurable to some extend. It is important to realize that this has nothing to do with technical concerns, which are handled by the CONTAINER and the ANNOTATIONS. Instead, we aim for flexibility in the concept of the entity itself. For example, an *Employee* entity could be designed to contain only the core attributes and behavior. In addition, it could provide a list of name-value pairs to store additional attributes, which can be defined specifically for each use case (or department, in our example). The component need not be changed, and reuse is encouraged and simplified[3].

This kind of variability is especially important for software that is intended explicitly for reuse, which is the case in an open market of off-the-shelf components that can be used to assemble applications. To gain a significant market share, the manufacturer here needs to consider the requirements of many customers, otherwise they will prefer to develop their own software rather than buy off-the-shelf components.

While the component architectures usually only provide a very limited support for this kind of flexibility, some application design patterns can help significantly. There are also several patterns available in the patterns community, for example *Strategy* [GHJV94], *Property List* [SR98], *Configurability* [SO99] or *Reflection* [BMRSS96].

3. Of course, you must make sure that the *Employee* component is still conceptually an Employee component. The additional attributes should therefore only be used for additional data that does not change the nature of the concept. If not, reuse is made harder because everybody uses the component completely differently.

Development roles

Although tools and processes, as well as architectures such as components or frameworks, provide help, software development is still a very complex task. For projects of any reasonable size, more that one developer is required. Together, these people will more easily cope with the complexity of a software system than one person can alone. To simplify the work of a team it is useful if the underlying architecture supports a well-defined division of work.

Separation of Concerns already mentions that different developers can implement technical and functional concerns. Based on this discussion, component architectures usually distribute the work to the following development roles:

- *Component Developer*. The responsibility of the component developer is to specify the COMPONENT INTERFACES[4] for the components in the system, to create implementations for the components and to specify the relationships between the components. He does not need to worry about technical concerns.
- *Container Developer*. The container developer creates the CONTAINER in which the components will be installed for execution. He is the person who is responsible for implementing the technical concerns. He does not need to worry about business logic. Note that it is common to buy the container from a third-party vendor – it is usually part of an application server or operating system.
- *Deployer*. The task of the deployer is to manage and configure the container, install the components in the container and make sure they run smoothly.

Many component architectures define even more roles. EJB, for example, specifies seven different roles – the *Deployer* role is separated into *Assembler*, *Deployer*, and *Container Administrator*, the *Container Developer* is separated into *Server Provider* and *Container*

4. Of course, the interfaces might also be specified by an analyst or the system architect. In general, this means that the *Component Developer* role might be separated into several subroles, usually related to the different project phases.

Provider. However, the coarse-grained distinction presented above is enough for our purposes.

In the pattern language in Part I we will show which pattern is most relevant to each of these roles. The language contains patterns for all three of them.

Components: a silver bullet?

This book elaborates on the properties and advantages of component architectures. But we would like to point out a couple of things that should be considered *before* you decide to use a component architecture like EJB, CCM or COM+:

- *Programmers' discipline*. Component architectures help you achieve a lot of benefits. For example, they allow you to create reusable entities that are not too dependent on other entities. However, to exploit these benefits fully, the developer has to contribute. For example, a thorough analysis phase is necessary to break the application into reusable parts. Additionally, there are many ways in which interfaces can be specified: strongly-typed interfaces, as mandated by EJB, CCM and COM+, are relatively hard to change once they are deployed and used, whereas a weakly-typed interface using generic method based on a strongly-typed one, for example based on key-value pairs for attributes or using strings as method calls, can be more flexible, but also required more thorough tracking of implicit dependencies. As you can see, sound software engineering principles still apply.
- *Accidental complexity*. Component architectures can provide scalability, fail-over and other beneficial features straight out of the box. The developers of business systems need not directly care about these issues. However, you might know that there 'is no such thing as a free lunch'. These features don't come for free. The framework implementing the technical concerns of a component architecture (usually called an application server) is a complex system in its own right. This can introduce additional complexity into a system – POSA2 calls this kind of complexity

accidental complexity, in contrast to the inherent complexity introduced by the direct requirements.

  For example, distributed systems, which component based systems usually are, introduce additional error conditions, debugging is much more complex and the application server itself has to be managed and is a candidate for failure. While component architectures undoubtedly have their benefits, you have to decide for each project whether you really need the features, and whether they are worth the additional overall system complexity they introduce.

- *Standards and today's technology.* When you use technologies such as EJB, CCM or COM+ you will usually buy an application server from a third-party vendor. Although they are all 'standards-compliant', there are always vendor-specific extensions, or slight differences in interpreting the standard. In addition, in some areas standards will always leave some aspects for the implementers to decide, to allow them the opportunity to create a business advantage.

  As a consequence, different application servers have significant differences regarding performance, scalability et cetera. Porting an application from one vendor's server to another is not always possible. Lock-in can occur if you do not encapsulate vendor-specific aspects.

So, as with any technology, be sceptical about what vendors and standardization bodies say. Do a profound analysis and decide which technology is right for your project. Sometimes there are much simpler solutions than you might think at first glance.

Part I A Server Components Pattern Language

This part of the book contains a pattern language that describes an architecture for a server-side component infrastructure. This part of the book is organized into eight chapters. Each chapter contains two to five patterns that naturally belong together. The chapters are arranged so that they progressively go into more detail:

- Chapter 1, *Core Infrastructure Elements*, introduces components and the container and goes into some detail about different kinds of components. This is a conceptual chapter that lays the basis for the more detailed patterns.
- Chapter 2, *Component Implementation Building Blocks*, describes components in more detail. A component usually has a well-defined interface, and the container needs a way to control the lifecycle of an instance. Means are also needed to control some aspects of the container's implementation of technical aspects.
- Chapter 3, *Container Implementation Basics*, is a relatively long chapter that goes into some detail about typical implementations of containers. The container usually needs to use techniques such as code generation to adapt its generic parts to specific components, resource management techniques such as instance pooling to manage its resources, or interception to make its own behavior configurable.

- Chapter 4, *A Component and its Environment*, describes how a component interacts with its environment. This includes access to configuration parameters, resources, access to the container, and access to other components in the system.
- Chapter 5, *Identifying and Managing Instances*, describes how component instances can be created or rediscovered by clients or other components.
- Chapter 6, *Remote Access to Components*, describes how components can be accessed remotely. Remote access to distribute an application over several machines is important for reasons of scalability and fail-over.
- Chapter 7, *More Container Implementation*, provides further details about implementing a container. This includes error handling, inspecting the component and automatically generating parts of the component's implementation.
- Chapter 8, *Component Deployment*, describes how a component is brought into a container and packaged efficiently for transport and distribution.

The patterns can be read in sequence, although some forward references cannot be avoided. When this occurs, we give a very short description, usually one sentence, to introduce the referenced pattern briefly.

Language map

A complete map of our pattern language would be *very* big and not very helpful. This is why we do not include a complete language map. Instead, each chapter starts with a map of the patterns covered in the respective chapter. The patterns from earlier chapters that are used to define the context are greyed out. At the time of writing we were still experimenting with the form of the language map. If you find a suitable form that allows the whole language to be displayed effectively, please contact us and we will happy to publish it on the book's web site at www.servercomponentpatterns.org.

Sequences through the language

There are several ways in which the patterns in this part can be read. You can read all patterns in the sequence in which they appear in the book, of course. However, you might want to read only some of the patterns to get an overview of the whole thing, or you might want to set a specific focus, such as container implementation. This section therefore provides several alternative sequences through the language.

Component architecture overview sequence

This sequence should be used by readers who just want to get a high-level overview of how component infrastructures work in general:

1. COMPONENT
2. CONTAINER
3. COMPONENT INTERFACE
4. LIFECYCLE CALLBACK
5. ANNOTATIONS
6. COMPONENT IMPLEMENTATION
7. VIRTUAL INSTANCE
8. COMPONENT PROXY
9. COMPONENT CONTEXT
10. NAMING
11. MANAGED RESOURCE
12. COMPONENT HOME
13. COMPONENT BUS
14. CLIENT-SIDE PROXY
15. CLIENT LIBRARY
16. COMPONENT INSTALLATION
17. COMPONENT PACKAGE

Component developer details

This sequence should be used by readers who aren't too worried about container implementation details, but who do want to know everything about components and their implementation.

1. COMPONENT
2. ENTITY COMPONENT
3. SERVICE COMPONENT

12. COMPONENT CONTEXT
13. NAMING
14. MANAGED RESOURCE

4. SESSION COMPONENT
5. CONTAINER
6. COMPONENT INTERFACE
7. LIFECYCLE CALLBACK
8. ANNOTATIONS
9. COMPONENT IMPLEMENTATION
10. IMPLEMENTATION RESTRICTIONS
11. VIRTUAL INSTANCE

15. PRIMARY KEY
16. HANDLE
17. COMPONENT HOME
18. SYSTEM ERRORS
19. COMPONENT INSTALLATION
20. COMPONENT PACKAGE
21. ASSEMBLY PACKAGE
22. APPLICATION SERVER

Container implementation details

This sequence is especially interesting for readers who don't want to understand components in depth, but who do want to build their own component infrastructures and need to learn the details of container implementation.

1. COMPONENT
2. CONTAINER
3. COMPONENT INTERFACE
4. LIFECYCLE CALLBACK
5. ANNOTATIONS
6. COMPONENT IMPLEMENTATION
7. IMPLEMENTATION RESTRICTIONS
8. VIRTUAL INSTANCES
9. INSTANCE POOLING

15. MANAGED RESOURCE
16. PLUGGABLE RESOURCES
17. PRIMARY KEY
18. HANDLE
19. COMPONENT HOME
20. COMPONENT BUS
21. INVOCATION CONTEXT
22. CLIENT LIBRARY
23. CLIENT REFERENCE COOPERATION

10. PASSIVATION
11. GLUE CODE LAYER
12. COMPONENT PROXY
13. COMPONENT CONTEXT
14. NAMING

24. SYSTEM ERRORS
25. COMPONENT INTROSPECTION
26. IMPLEMENTATION PLUG-IN
27. COMPONENT INSTALLATION
28. APPLICATION SERVER

A conversation

For a pattern language is it critical that you, the reader, understand how the different patterns go together. You need to get the 'big picture'. We therefore want to introduce the patterns with a hypothetical dialogue, between a component software consultant and a Java programmer from a large company's development staff – his statements are given in *italics*. If you don't understand all of it, don't worry. Just read the patterns and come back to the conversation afterwards. Part III of this book provides a much more detailed conversation based on EJB.

❊ ❊ ❊

Ok, we have to do this new enterprise system for our customer iMagix, Inc. They want us to create an e-business solution for their rapidly growing Internet site. They expect significant growth in the future, so the system must be scalable. And because they have strong competitors, they must be able to add new features quickly to always be one step ahead … You know, the usual things, and it must be out by next November. This gives us just six months to go…

Is it just this system they want to build, or is the system a starting point for further systems. What I want to say is: is reuse necessary and welcomed?

Mmm, yes, I think so.

Are they focussed on a specific technology?

They say, we're free to do anything, as long as it works fine… But looking at their current development, they seem to like Java.

Ever thought of using a component-based approach?

Of course! I mean, everything is component-based today. Another buzzword. Can't get around it, can you?

Yes, but actually, beyond the hype, CBD[1] has some pretty neat advantages. Should we talk about these a bit?

Well, why not? After all, that's why we hired you.

Ok. So the basic assumptions are: you want to build an N-TIER SYSTEM to separate user interface from data storage and from business logic, and to distribute your system over several machines for load-balancing and fail-over. The second thing is: you want to SEPARATE CONCERNS.

What's that?

In this context, that means that you explicitly distinguish between functional aspects of your application – usually called business logic – and technical requirements, such as transactions, scalability, persistence, security…

Yeah, but why? I mean you always somehow separate the two, but…

But if you do this systematically, you can buy something that takes care of your technical requirements. A so-called CONTAINER. This saves a lot of work.

Ok, sounds interesting. And where do I put my business logic?

You put it into COMPONENTS, which live inside the CONTAINER. Each COMPONENT represents a process, a business entity or a service. Together, COMPONENTS and the CONTAINER make up the business tier of the application. Of course, you might need databases, web servers and browsers as well.

And what does that imply for maintenance and evolution of the software? Sounds like I get my application built from separate 'blocks of functionality'.

First thing is, you can reuse technical and functional concerns sepa-

1. Component Based Development [SZ99]

rately, because they are kept separate throughout the whole lifecycle of your application. You can buy a new version of your CONTAINER without changing the COMPONENTS. Second, your COMPONENTS need to have some specific characteristics to make all this work. Each COMPONENT must only be accessed via a well-defined COMPONENT INTERFACE. And each COMPONENT must exhibit a certain degree of functional variability, or flexibility, in order to be reused in several contexts.

But life isn't that simple! You cannot completely separate technical and functional concerns. For example, you have to demarcate transactions or check security restrictions…

Yes that's why you use so-called ANNOTATIONS. These let you specify – as opposed to program – the way the CONTAINER should handle your COMPONENTS from a technical point of view.

Mmmmm, ok. Sounds as if that could work. So, we have a component for each business entity and process – this can get to be quite a lot over time… The, er… how did you call it?

CONTAINER?

Yes, CONTAINER, the CONTAINER probably needs to do something to keep resource usage acceptable, especially memory. I mean, each component might need significant amounts of memory.

Yes, absolutely. Basically, the CONTAINER uses only VIRTUAL INSTANCES. There is a logical identity, for example a customer 'James Bond', and the physical identity of a specific component instance – both can be mapped as required using PASSIVATION and INSTANCE POOLING. This helps to manage resources effectively.

Ok, I understand that the CONTAINER assigns logical identities to physical identities as needed – but how can he know which one is needed?

Basically it uses a COMPONENT PROXY as a placeholder for the real instance, and uses it to 'catch' an incoming request and make sure that the required logical identity has a physical instance assigned to it. And by the way, LIFECYCLE CALLBACKS are used to let a COMPONENT instance impersonate different logical identities over time.

You're beginning to convince me. But… hmm…, how do I get in touch with these COMPONENTS, *I mean, basically they are some kind of remote object, aren't they? At least, we want to have them remotely accessible.*

Yes you're right. There is a two-step process for obtaining a reference to a COMPONENT instance. You use COMPONENT HOMES to create the concrete instances and a NAMING service to get a reference to the home.

Ok, I understand NAMING, *I know it from CORBA. But what is a* COMPONENT HOME*?*

You know the GoF book, *Design Patterns* by Gamma, Helm, Johnson and Vlissides? [GHJV94]

Oh, of course!

Good. So, a COMPONENT HOME is basically an application of the *Factory* pattern that manages all instances of a specific COMPONENT TYPE.

Mmm, you talked of ANNOTATIONS *that determine the behavior of the* CONTAINER*. Isn't it too much overhead for the* CONTAINER *to look at these* ANNOTATIONS *all the time at run-time?*

Yes, it would be if these ANNOTATIONS were interpreted at run-time. But there is an additional step, called COMPONENT INSTALLATION, where, among other things, the ANNOTATIONS are read by the CONTAINER and a GLUE-CODE LAYER is generated. This code is directly executed at run-time and serves as an adapter between the generic CONTAINER and specific COMPONENTS with their individual technical requirements.

But listen, a COMPONENT *cannot live on its own. You have to access certain parts of the environment at times, for example for database connections, or – I guess – to look up other* COMPONENTS*.*

Right. The CONTAINER provides each COMPONENT instance with a COMPONENT CONTEXT, to access the world outside the COMPONENT. Or at least parts of it – those parts the CONTAINER wants the COMPONENT to access.

Yes, but…

And!… In addition, the CONTAINER also MANAGES RESOURCES. It automatically creates and controls database connection pools, for example. Your COMPONENTS can use these pools without caring about how and when the connections are created.

Ok ok, I give up! One last question: If the CONTAINER *manages transactions and security for me, then it must have more information than is available in a regular method call. I know that CORBA OTS [OMGOTS] uses a mechanism called context propagation. Is this also true for CBD?*

Yes, we generally call it the INVOCATION CONTEXT. The transport protocols allow for that.

❊ ❊ ❊

After this discussion, the software company started to look more closely at component technologies. Several weeks later, the programmer came up with a couple of additional questions, or rather, observations. Once again, he talked to the consultant.

❊ ❊ ❊

Hi, how are you?

I'm fine. What about you and your COMPONENTS?

We're doing quite well, thanks. However, while looking at EJB, COM+ and CCM, we found some interesting things I'd like to discuss with you. For example, they have several types of COMPONENTS.

Yes, there are ENTITY, SERVICE and SESSION COMPONENTS. Each type has its own characteristics regarding persistence, concurrency and so on. The persistent one needs a PRIMARY KEY for identification. References to the others can be stored persistently using a HANDLE.

Right. And in addition, all COMPONENT *models impose some* IMPLEMENTATION RESTRICTIONS *on the* COMPONENTS *to make sure they can run in the* CONTAINER *without compromising its stability. More, they distinguish two kinds of errors: application errors and* SYSTEM ERRORS. *The* CONTAINER *provides some default error handling for the latter, such as aborting the current transaction.*

Right. It seems you've learned quite a bit. There are other interesting things. For example, COMPONENT INTROSPECTION can be used to help build a graphical ANNOTATION builder tool.

But tell me, what kind of support is there to really build complete applications from these components? Is there any help for assembling them?

Yes. Some component architectures provide a way to specify which REQUIRED INTERFACES a component needs in order to function correctly. COMPONENT PACKAGES and ASSEMBLY PACKAGES are also important aspects. They help to distribute complete COMPONENTS.

Yes, and I learned that some architectures provide PLUGGABLE RESOURCES, kind of custom-defined MANAGED RESOURCES.

Indeed. PLUGGABLE RESOURCES help to integrate your CONTAINER with other, perhaps legacy, back-end applications by defining an interface to plug them into the CONTAINER.

So, tell me, is all this reality today? Are COMPONENTS the next silver bullet that solves all of our problems?

There is never such a thing as a silver bullet. Today's COMPONENT architectures, no matter whether EJB, CCM or COM+, are far from perfect. They have some flaws in their design, and some commercially-available CONTAINERS suffer from compatibility problems and the usual bugs – after all, it's still a quite new technology. But nevertheless, COMPONENTS can solve some problems quite neatly.

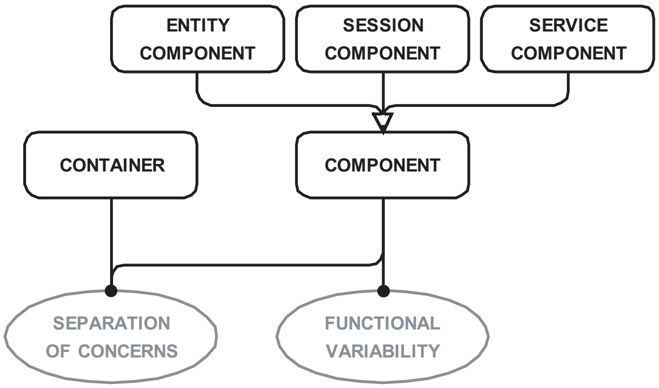
❋ ❋ ❋

And they were happy and used COMPONENTS till the end of their careers…

We hope that this brief conversation has motivated you to read further. After reading the patterns, it is a good idea to come back to this section and read it again. You will see it with new eyes, and the overall picture should become even clearer.

1 Core Infrastructure Elements

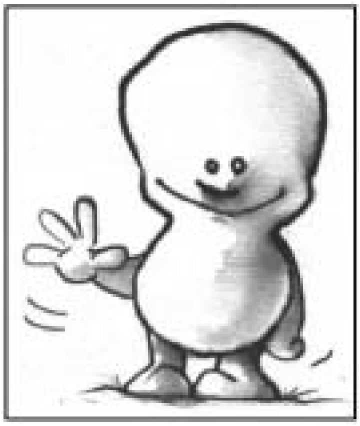
The patterns in this chapter make up the fundamental building blocks of a component architecture. They are based directly on the Separation of Concerns and Functional Variability principles.



COMPONENT describes the basic structure of an individual component in a component architecture. Each COMPONENT implements a well-defined functionality. The CONTAINER provides a run-time environment for COMPONENTS, adding the technical concerns. There are different specialized kinds of COMPONENTS. SERVICE COMPONENTS are used to implement stateless, API-like services, ENTITY COMPONENTS represent persistent business entities and SESSION COMPONENTS are a representation of a client session on the server.

Component

You apply SEPARATION OF CONCERNS. As a consequence, your system is divided into two distinct parts: the functional part and the technical concerns.



❃ ❃ ❃

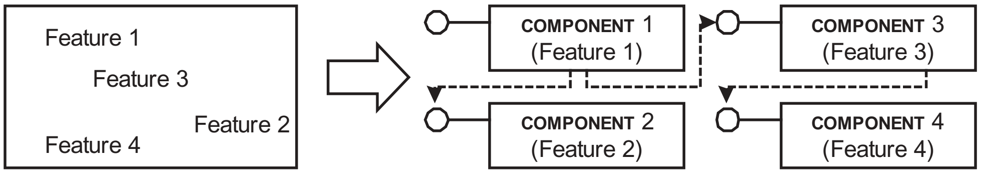
A single big chunk of functionality that is separated from the technical concerns is better than mixing the two concerns. However, the chunk is still just a big 'ball of mud': changes to a particular area of the system's functionality might still affect the whole system. Moreover, independent deployment and update of each part are impossible and you want to be able to reuse distinct functional parts of the system separately.

The primary reason for applying the principle of SEPARATION OF CONCERNS is to allow the evolution of a system and to promote reuse. Yet it is not sufficient to apply this principle at the most coarse-grained system level only. To be able to develop, evolve, reuse, handle and combine the functionality of a system most effectively and efficiently, it is also important to separate different functional parts from each other. Ideally, system development happens by merely assembling existing reusable functional parts.

Separating different functional parts is not sufficient though. In today's world, environments, markets, customers, requirements and company structures change fairly quickly. Most such externally-influenced changes also result in changes within the functional parts of a software system. Immediate response to such change requests is only possible, however, if we can localize changes and minimize the effects of changing a particular functional part on other functional parts. For example, when changing the implementation of a single functional part, we do not want to have to recompile the entire system. We also need to encapsulate our functional parts appropriately.

Therefore:

Decompose the functionality of your application into several distinct COMPONENTS, making each COMPONENT responsible for providing a specific self-reliant part of the overall functionality. Let every COMPONENT implement its responsibilities in full without introducing strong dependencies to other COMPONENTS. Compose the complete application by assembling the required functionality from these loosely-coupled COMPONENTS.



❊ ❊ ❊

There are two important concepts here: coupling and cohesion. A COMPONENT should exhibit a high level of cohesion, while being only loosely coupled to other COMPONENTS. This means that each COMPONENT should provide a set of features that naturally belong together, but should not directly depend on the internals of other COMPONENTS. To achieve this loose coupling, COMPONENTS access each other only by COMPONENT INTERFACES. The COMPONENT INTERFACE is separate from the COMPONENT IMPLEMENTATION.

As a consequence of decomposing the functional concerns into COMPONENTS, as described above, each COMPONENT becomes a small 'application' in its own right. The application's functionalities are well encapsulated and localized. Changing a specific functionality only requires changes to one COMPONENT. Only the specific COMPONENT has to be redeployed after a change.

As the COMPONENT INTERFACE is physically separated from the COMPONENT IMPLEMENTATION, clients need not be changed when the implementation of a COMPONENT is changed. In a way, this approach is a very sophisticated and consistent form of modularization.

Applications in turn are created by 'wiring' a set of COMPONENTS. The application itself is reduced to orchestrating the collaboration of the COMPONENTS. Different applications can be created by combining COMPONENTS in different ways. However, while an application might reuse a specific COMPONENT, it might require a slight modification of behavior or structure. The principle of variability must therefore be employed.

A COMPONENT implements only functional concerns. However, all functional concerns must be complemented with technical concerns to fulfill the requirements of a particular system. Provide these technical requirements to COMPONENTS via a CONTAINER into which COMPONENT can be INSTALLED without affecting other COMPONENTS as long as the COMPONENT INTERFACES remain stable.

Note that a COMPONENT is a passive entity. It waits for invocations from clients (which might be real remote clients or other COMPONENTS), executes the invoked operation, and then returns to passive mode again. This means that a COMPONENT instance cannot start doing something on its own. This has important consequences for the way in which the CONTAINER can be implemented: the CONTAINER can safely assume that if no method invocations arrive for a specific instance, the instance will never execute any code – thus it could for example be removed from memory until the next invocation arrives. See VIRTUAL INSTANCE and COMPONENT PROXY for details.

We already mentioned that a COMPONENT consists of several parts. A COMPONENT INTERFACE is used to reduce dependencies and to make the implementation exchangeable. COMPONENT HOMES are used to manage the instances of a specific COMPONENT – they provide a way to create, find or destroy such instances. A COMPONENT IMPLEMENTATION is of course also necessary, as it implements the functional concerns of the operations specified in the interface. Last but not least, you need to define ANNOTATIONS for the COMPONENT. These specify how the COMPONENT should be integrated with the technical services the CONTAINER provides.

It is not always simple to find suitable areas of functionality that lend themselves to being encapsulated as one COMPONENT. Finding such areas is a task usually done during the analysis phase and requires some experience. Component architectures don't provide help here, however, but they usually provide different types of COMPONENTS for different types of requirements:

- ENTITY COMPONENTS are used to represent business entities and usually contain persistent state.
- SERVICE COMPONENTS are stateless and provide services to the application.
- SESSION COMPONENTS are stateful and are usually used to represent small processes or use cases.

Each of these types has different characteristics and requires specific treatment by the CONTAINER – in practice, different CONTAINERS are used for each kind of COMPONENT.

❊ ❊ ❊

EJB, CCM and COM+ are all based on the concept of a COMPONENT. In all three cases, an application is built by assembling, or 'wiring', COMPONENTS. All three technologies consider COMPONENTS the smallest technical building block of a component-based system.

There is another notion of 'component' that is also called a Business Component [HS00] It spans several layers, such as user/session/application or logic/persistence, and can therefore consist of several COMPONENTS as defined here. Business COMPONENTS are not what we understand when we talk about a COMPONENT.

EJB is a relatively new component model that has no legacy history. It uses the Java technology and has found widespread use for enterprise applications. EJB is part of the Java 2 Enterprise Edition (J2EE) [SUNJ2EE], a complete suite of technologies for building server-side business applications.

The CORBA Component Model (CCM) has two ancestors: CORBA and EJB. It takes the EJB component model to the CORBA world, retaining almost all of the aspects of CORBA while still being compatible with EJB. CORBA clients and EJB clients can therefore access CCM COMPONENTS. Of course, some features such as multiple interfaces are not accessible to them. At the time of this writing, only partial implementations of the CCM are available.

COM+ is the successor to MTS and DCOM, which in turn is an extension of COM to allow remote access to COMPONENTS. COM itself – the Microsoft Component Object Model – has been around for a long time in several variations and with several names, such as COM, OLE, or ActiveX. COM+ supports the concept of *component aggregation*. The interfaces of aggregated COMPONENTS can be 'published' by aggregating COMPONENTS, enhancing reuse while hiding the aggregation structure from clients.

Container

According to the principle of SEPARATION OF CONCERNS, you decompose your functional requirements into COMPONENTS.

❊ ❊ ❊

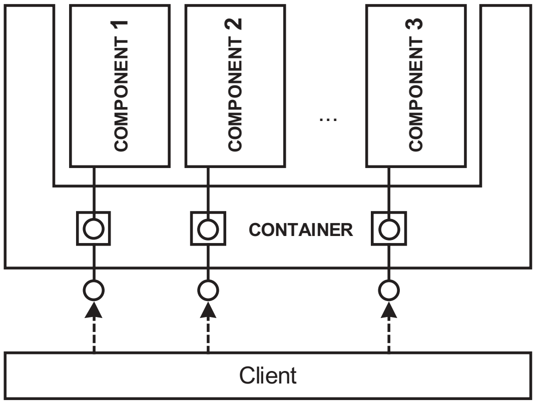
Your COMPONENTS address functional concerns only, that is, they contain pure business logic that does not care about technical concerns. However, to execute these COMPONENTS as part of a concrete system, you also need something that provides the technical concerns and integrates the COMPONENTS with their environment. Moreover, you want to be able to reuse the technical concerns effectively.

There are two kinds of technical concerns: those that clients require, such as security, transactions, and those that the COMPONENT itself requires, such as concurrency, persistence, and access to infrastructure resources. COMPONENTS not only need to benefit from these technical concerns without being tied to a particular infrastructure or platform, such as a transaction monitor or a security infrastructure. These services must also be provided non-intrusively: they have to be added 'from the outside'. From the client's perspective, the COMPONENT should appear to be a single entity providing their functionality integrated with the required technical concerns. From the COMPONENT's perspective, the technical concerns should be provided 'magically' – the COMPONENT code must not be 'polluted' by code implementing technical concerns.

As many COMPONENTS and COMPONENT-based applications depend on the same technical concerns, their implementations should be reusable. In other words, you do not want to reinvent and re-implement the technical concerns for each application over and over again. The goal is to standardize and implement these services generically, for example using frameworks or code-generation techniques.

Therefore:

Provide an execution environment that is responsible for adding the technical concerns to the COMPONENTS. This environment is generally called CONTAINER. Conceptually, it wraps the COMPONENTS, thus giving clients the illusion of tightly-integrated functional and technical concerns. To implement these technical concerns in a reusable manner, a CONTAINER uses frameworks and other generic techniques such as code generation.[1]



✲ ✲ ✲

To integrate the COMPONENTS with the CONTAINER while not polluting their code with technical concerns, use ANNOTATIONS. ANNOTATIONS specify the technical requirements for a particular COMPONENT separately from their COMPONENT IMPLEMENTATION. To allow the CONTAINER to apply these ANNOTATIONS to the COMPONENTS requires an explicit step, the COMPONENT INSTALLATION. This prepares the CONTAINER to host and execute the new COMPONENT as specified. Usually, this involves the creation of a GLUE-CODE LAYER, which adapts the generic parts of the CONTAINER to a specific COMPONENT.

There are different ways in which the combination of the generic parts of a CONTAINER and a GLUE-CODE LAYER share their work. Either the CONTAINER is already able to provide most functionality generically, and the GLUE-CODE LAYER merely plays the role of an adapter. Or, the CONTAINER is basically just a collection of hooks calling back into the GLUE-CODE LAYER, which then does most of the work. The extreme case is that the CONTAINER is generated altogether during COMPONENT INSTALLATION.

To optimize resource consumption and performance, a CONTAINER usually provides VIRTUAL INSTANCES. These decouple the lifecycle of a logical entity that is visible to client applications or other COMPONENTS from its physical representation in the CONTAINER.

As the different COMPONENT types – SERVICE, SESSION and ENTITY – have different characteristics, you need to provide different CONTAINERS. These are integrated in an APPLICATION SERVER that provides all the services common to the different CONTAINER types.

The management of resources used by a COMPONENT such as a database connection is also a technical concern. The CONTAINER must therefore MANAGE RESOURCES on behalf of the COMPONENTS it hosts. The CONTAINER cooperates with the APPLICATION SERVER over this.

To realize high-availability requirements or to provide load-balancing, CONTAINERS can be federated, or clustered. A set of physi-

1. Note that it is not the goal of this book to provide details of CONTAINER implementation. We focus on the interfaces needed by COMPONENTS and the CONTAINER to facilitate their cooperation.

cally separated CONTAINERS, which usually run on different machines, is logically joined to act as one CONTAINER. This is transparent both to the client and to the COMPONENT developer.

❊ ❊ ❊

EJB requires special software that plays the role of the CONTAINER. Usually, the CONTAINER is embedded in a larger application, a J2EE APPLICATION SERVER. A J2EE-conforming APPLICATION SERVER has to provide additional services, such as NAMING, accessible through the JNDI API, transactions, or Servlet and JSPs for the dynamic creation of web content. From EJB 2.0 onwards, messaging is integrated with EJBs in the form of Message-Driven Beans, which are Beans that act as 'message receivers'.

Usually, J2EE APPLICATION SERVERS are implemented in Java and provide separate CONTAINERS for each COMPONENT type. EJB CONTAINERS have to conform to the EJB specification to allow seamless exchange of COMPONENTS between different CONTAINER vendors. Nevertheless, they are free to offer different levels of quality of service. For example, some CONTAINERs might offer load balancing, high availability, or advanced caching strategies while other CONTAINERs do not.

CCM also uses the concept of a CONTAINER, although no complete implementations are on the market at the time of writing. CCM explicitly defines different CONTAINER types for the different types of COMPONENTS, thus there are entity, service, process and session CONTAINERS. To implement these different types of CONTAINERS, the facilities already provided by the Portable Object Adapter[2] (POA) are used. This means that a particular CONTAINER is based on settings for threading, life-span, activation, servant retention, and transaction policies. Because all this is already part of 'ordinary' CORBA, it is possible to access a COMPONENT from a client that does not know that the CORBA object to which it has a reference is actually a COMPO-

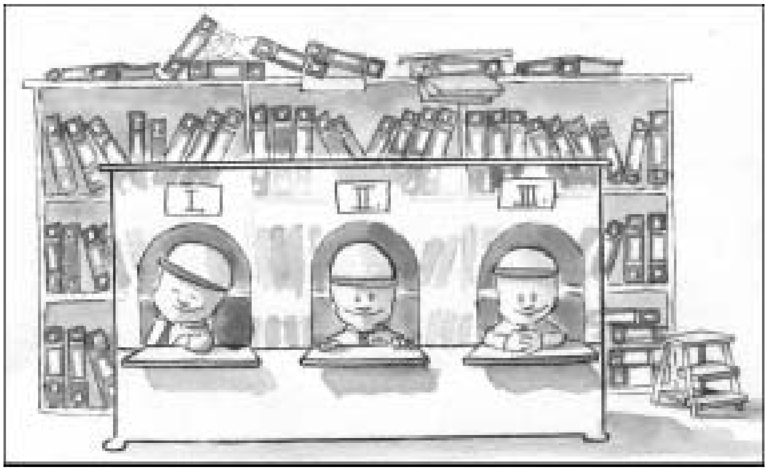
2. The Portable Object Adapter is a framework for managing CORBA server objects inside an application. In contrast to the Basic Object Adapter, the BOA, it is more flexible (for example regarding activation and eviction policies) and it is better standardized and therefore more portable.

NENT. Some features such as multiple interfaces are of course not accessible for ordinary CORBA clients. In addition to the technical concerns mentioned above, CCM CONTAINERS also handle persistence in a well-defined way. The persistent state of a COMPONENT can be specified abstractly, and a mapping to logical storage devices can be provided. These logical storage devices are then mapped to real databases with CONTAINER-provided tools.

In COM+, the role of the CONTAINER is played by parts of the Windows 2000 operating system. In the pre-COM+ days, the Microsoft Transaction Server (MTS) played the role of the CONTAINER and was not part of the operating system. Every COM+ application, which consists of several COMPONENTS in their own DLLs, runs in a surrogate process controlled by COM+. It handles threading, remoting, synchronization, and pooling. COM+ technically provides only one type of COMPONENTS (SERVICE COMPONENTS) but SESSION COMPONENTS can easily be emulated by the programmer. Persistence is not explicitly addressed by the COM+ CONTAINER.

Service Component

You use COMPONENTS to implement your functional requirements, and you are now trying actually to define the COMPONENTS into which the application can be decomposed.



❊ ❊ ❊

Your application requires 'service providers', COMPONENTS that implement some kind of componentized API. The required COMPONENT typically modifies other COMPONENTS or external (legacy) systems. It serves as a facade for a subsystem that possibly also consists of several other COMPONENTS.

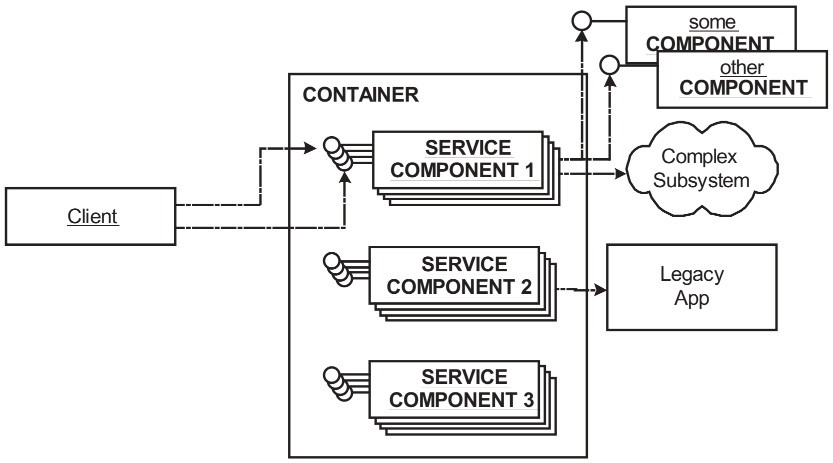
Processes, or 'micro-workflows', are the basis for every business application. It is critical to ensure that specific processes are always executed correctly and consistently. It should not be possible for client applications to modify a business entity freely by invoking any operation in any order, as this might compromise consistency. All modifications should be embedded in a well-defined process. It is never a good idea to rely on the clients of your COMPONENTS to implement the processes correctly. Off-loading the responsibility for system consistency to client applications is risky, because you cannot enforce the correctness of clients. In addition, reusing such processes is difficult. You therefore want to make such processes part of your COMPONENT model. To ensure data consistency is preserved, a set of modifications might even have to run within a single transaction, and only after checking permissions.

When integrating legacy systems, you want to ensure that they 'feel' the same for application programmers as all the other parts of your COMPONENT-based application. You therefore need to provide a componentized interface to legacy systems. The relevant COMPONENT does not do very much on its own – it just forwards the request to a legacy system, usually using a different communication protocol or after some adaptations of the parameters.

Last but not least, you may want to use the componentized equivalent of the *Facade* pattern [GHJV94] to hide underlying complexity or to improve communication by reducing network communication overhead in case of remote access.

Therefore:

Encapsulate processes and micro-workflows as SERVICE COMPONENTS. SERVICE COMPONENTS are stateless, which means they cannot carry state changes from one invocation to another. This allows the CONTAINER to implement this kind of COMPONENT efficiently. This comes at the price that all operations have to be self-sufficient: they must implement a specific process and then terminate.



❊ ❊ ❊

A SERVICE COMPONENT can be used as a facade around a more complex system – be it a set of other COMPONENTS, a legacy application, or a relational database. The operations in such a COMPONENT can create a transactional bracket around the other operations called from within the implementation.

As mentioned above, SERVICE COMPONENTS are stateless. This does not mean that they cannot have attributes, but it means that there is no guarantee that a change in the state occurring during one operation is visible in a subsequent operation. All operations must finish their jobs themselves. While this might seem like a drawback, it is not a problem if this kind of COMPONENT is used in the situations for which it is intended. For example, if they just serve to call a service routine in a legacy system or to modify a set of business entities, no COMPONENT state is necessary. Of course, the COMPONENT can retain state that cannot be changed by the clients, such as the network address of the encapsulated legacy system.

Being stateless has a number of significant advantages from a performance viewpoint. The concrete identity and lifetime of an instance of a SERVICE COMPONENT are unimportant, because it has no state: the CONTAINER can create and destroy instances whenever it wants, and for the client all these instances appear to be equal.

The CONTAINER can use this to its advantage. For example, it can use POOLING or other load-balancing policies. POOLING means that the CONTAINER keeps a pool of identical instances and uses whatever instance is available to serve a client request. This instance can even reside in another CONTAINER, maybe even on another machine. Clients accessing the COMPONENT concurrently have the illusion of concurrent service from one COMPONENT – but there is no concurrency in the instances, because the CONTAINER makes sure that each client accesses its own instance. Nevertheless, there is no guarantee that an instance will serve the client for more than one operation. To manage the instances in the pool, the CONTAINER requires the COMPONENTS to provide LIFECYCLE CALLBACK operations, which it will call in order to notify the instance of lifecycle events.

This kind of COMPONENT is very close to the typical transaction programs running in a transaction monitor. The high performance and scalability of Online Transaction Processing (OLTP) applications stem from the fact that the transaction programs are stateless and can therefore be managed efficiently.

If a SERVICE COMPONENT is used as a facade to a more complex subsystem, in addition to preserving consistency, it can also reduce network traffic between the client and the CONTAINER, and thus improve performance. Instead of invoking several remote operations on several COMPONENTS subsequently, a client just invokes one remote operation on the facade SERVICE COMPONENT, and the subsequent operations are executed locally by the SERVICE COMPONENT.

❊ ❊ ❊

In EJB, SERVICE COMPONENTS are implemented by Stateless Session Beans. The fact that a Session Bean is stateless is be defined using the Deployment Descriptor, which is an implementation of ANNOTATIONS. Instances are accessed by clients by calling the *create()* operation on the Bean's HOME. As INSTANCE POOLING is usually used by the CONTAINERS, calling *create()* does not necessarily mean that a new physical instance is actually created. Usually a pooled, inactive instance will be returned to the calling client.

CCM also provides SERVICE COMPONENTS. They are declared using a *composition service* declaration and basically have the same properties as Stateless Session Beans in EJB.

In COM+ COMPONENTS are by default SERVICE COMPONENTS. Technically, this is the only kind of COMPONENT available in COM+. Other kinds of COMPONENTS derive from this kind of COMPONENT, although SESSION COMPONENT functionality can be achieved easily. Multi-client access is handled by defining threading policies ('apartments').

Entity Component

You use COMPONENTS to implement your functional requirements. You need to access business entities.

❊ ❊ ❊

Business entities are usually stored persistently in database systems or in legacy applications. To have a homogeneous programming model, you want to access these entities in the same way as any other COMPONENT instance. Such a 'componentized entity' must have a unique identifier and preserve the consistency guarantees made by the underlying database system.

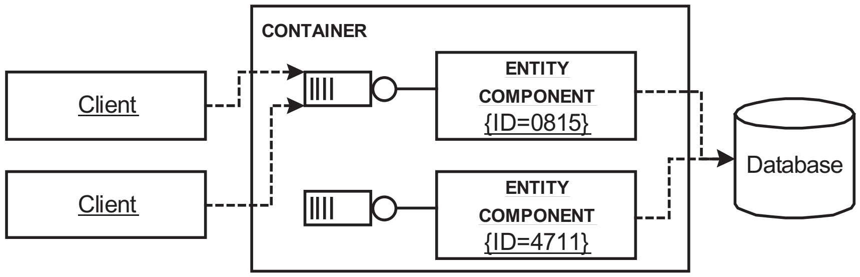
Accessing business entities stored in back-end systems can be achieved using a SERVICE COMPONENT that provides operations that manipulate the back-end entities directly. However, a SERVICE COMPONENT does not take care of issues that arise when an entity is modified by several clients concurrently. In addition, the stateless API-like programming model is not really convenient, because you cannot have one COMPONENT instance representing one business entity – instead you have to call each method with the unique identifier of the business entity.

While there are situations in which such a SERVICE COMPONENT-based approach is a good choice, you generally want to provide a more sophisticated interface, one that allows the CONTAINER to make use of the knowledge that the COMPONENT actually represents an entity. This allows the CONTAINER to realize more sophisticated resource optimization techniques.

Your programming model will be more 'object like' and thus more natural, because you have references to an entity with a unique identity and can manipulate it directly. A SERVICE COMPONENT can only represent an interface that allows you to modify underlying entities, but they cannot represent the entity itself.

Therefore:

Encapsulate business entity types as ENTITY COMPONENTS. An ENTITY COMPONENT instance represents a particular business entity. Its state is persistent in a store such as a database. Each instance has a unique identifier. The CONTAINER will coordinate concurrent access to a specific instance and ensure that consistency is preserved.



❊ ❊ ❊

Once the CONTAINER knows that a COMPONENT instance actually represents a specific entity, it can provide several additional services to the programmer:

- It can enforce consistency in the face of concurrent access, independent of the underlying back-end system. It can even provide transactions when the underlying system is not transaction-aware, or the CONTAINER can use the underlying persistence mechanism (such as a database) to implement synchronization of entity access.

- It can provide a more natural programming model in which an instance of the COMPONENT represents a specific entity rather than an API used to modify entities.
- It can implement optimizations based on the fact that it knows which entity the COMPONENT instance represents.
- It can even support the mapping of the COMPONENT to the underlying back-end system.

This is very convenient for the COMPONENT programmer, because, as all the other services are provided by the CONTAINER, they do not need to be re-implemented over and over again. In addition, the programming model for accessing entities is simple, homogeneous and independent of the back-end system used to store the entities.

Of course, all this does not come free. To make it work, the CONTAINER needs to know which entity is actually represented by an instance. PRIMARY KEYS are usually used to give each instance a unique identity. A typical optimization for ENTITY COMPONENTS is POOLING. In contrast to POOLING stateless SERVICE COMPONENTS, POOLING ENTITY COMPONENTS requires more control over the instance by the CONTAINER. The CONTAINER usually uses some form of LIFECYCLE CALLBACK to notify the instance of important events.

An ENTITY COMPONENT, however, also has liabilities. The CONTAINER's freedom to use its load-management techniques is limited, due to the fact that the consistency of the entity has to be ensured in face of multiple clients' concurrent accesses. ENTITY COMPONENTS therefore usually impose a relatively large performance overhead. This means that ENTITY COMPONENTS should be used only when the additional services provided by this kind of COMPONENT are really needed.

Although not explicitly designed for this task, ENTITY COMPONENTS can also be used to represent very long processes that need to be persistent. CCM provides a persistent component called a *process component* for exactly this purpose.

❊ ❊ ❊

EJB provides Entity Beans which implement the concept outlined in this pattern. Their persistence aspect can either be implemented by the Bean itself (implemented manually by the developer) or it can be handled by the CONTAINER. This is called *Bean-Managed Persistence* (BMP) or *Container-Managed Persistence* (CMP), respectively. EJB 2.0 additionally introduces concepts to let the CONTAINER manage relationships (associations) between ENTITY COMPONENTS. For each Entity Bean, a PRIMARY KEY class has to be defined. An instance of the PRIMARY KEY class uniquely identifies an instance of the COMPONENT. This PRIMARY KEY class must be *Serializable* (in the Java sense), so that it can be stored persistently in a database and passed around by value. The CONTAINER automatically manages synchronization for Entity Beans making sure that concurrent access by multiple clients is serialized.

The CORBA Component Model also provides ENTITY COMPONENTS. These are declared using the *composition entity* keywords in CIDL. Their persistence is usually handled by the CONTAINER. How this is done can be defined using the Persistent State Definition Language (PSDL), which provides a way to define how the CONTAINER should handle persistence. In CCM, the PRIMARY KEY of an ENTITY COMPONENT is a CORBA *value type*, to ensure proper transport semantics. Note that this PRIMARY KEY is not the same as the CORBA object reference (see VIRTUAL INSTANCE).

COM+ does not natively provide the concept of an ENTITY COMPONENT. You have to code the use of the persistent storage and state management manually.

Session Component

You use COMPONENTS to implement your functional requirements. You want to implement use cases as part of the COMPONENT model and SERVICE COMPONENTS are too limited.

❋ ❋ ❋

As part of your application you need to keep client- or session-specific state as part of the COMPONENT model, usually to model more complex business processes. This client-specific state is typically not persistent, and some type of garbage collection is required to clean up state data that is not used any more.

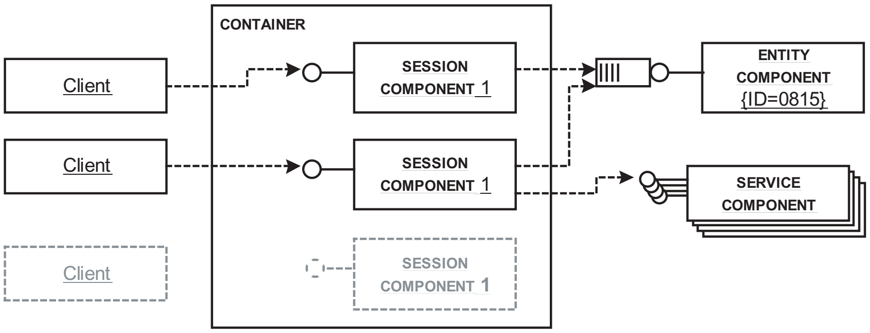
SERVICE COMPONENTS do well if simple processes must be implemented and if those processes can be implemented within one operation, or if the necessary persistent state is kept in a back-end system. However, in some cases you may need to keep state in the COMPONENT that is specific to *one* client. This means that if you have many clients, you will have many COMPONENT instances. You have to manage these potentially many instances efficiently to avoid resource problems in the CONTAINER.

ENTITY COMPONENTS seem suitable at first glance – however, they are too heavyweight: their state is stored persistently and they offer coordinated concurrent access by multiple clients with the help of the CONTAINER. This is not necessary for such a client-specific state – it does not need to be persistent, and it is not accessed concurrently, because the state is client-specific (thus there can be only one client per instance). Note also that such a session-state is not an entity, so it does need to have a PRIMARY KEY.

What you need is something between a SERVICE COMPONENT and an ENTITY COMPONENT.

Therefore:

Provide a third kind of COMPONENT, the SESSION COMPONENT. SESSION COMPONENTS do have client-specific state, but this state is not persistent. No concurrency is allowed, thus no synchronization overhead is implied. The CONTAINER ensures that resource usage is kept reasonable even for large numbers of clients.



✵ ✵ ✵

SESSION COMPONENTS are exactly what is necessary for representing client-specific state on a server. The CONTAINER assumes that only one client accesses the COMPONENT instance, so no synchronization is necessary. It is usually the responsibility of the client programmer to make sure that this is actually true.

You have to decide whether it really makes sense to expect only one client to access the SESSION COMPONENT, because of course the one client can pass the reference to other clients. In such a case, it might be better to use an ENTITY COMPONENT, because you might need features like coordinated access to the COMPONENT.

As the number of simultaneous SESSION COMPONENTS is directly proportional to the number of clients, SESSION COMPONENTS might exceed your server's memory capacity. To avoid this, the CONTAINER uses PASSIVATION. PASSIVATION stores temporarily-unused SESSION COMPONENT instances on secondary storage. To allow the instance to prepare for passivation or to 'wake up' after reactivation, SESSION COMPONENTS need to provide LIFECYCLE CALLBACK operations for the CONTAINER to invoke at suitable times.

In contrast to ENTITY COMPONENTS, where destruction (or removal) is a semantically-important operation, removing a SESSION COMPONENT is merely a way for the client to tell the CONTAINER that it no longer needs the instance. Because clients can crash, the CONTAINER needs to employ some type of garbage collection to get rid of instances that are no longer needed. Usually, a timeout is used to determine when the CONTAINER can remove SESSION COMPONENT instances.

Although a SESSION COMPONENT has no logical identity, you might still want to store a reference to such an 'anonymous' instance. A HANDLE is an object that you can pass around by value and store on secondary storage, and that allows you to 'de-reference' it to connect with the object from which it was created.

Note that as a SESSION COMPONENT is not persistent, is will usually not survive a server crash. So if the server does not provide fail-over and SESSION COMPONENT replication in a cluster, be sure not to store mission critical data in a SESSION COMPONENT – use ENTITY COMPONENTS instead.

A SESSION COMPONENT can also be used to 'emulate' long transactions. As most database systems don't support long transactions and optimistic locking efficiently, because severe locking problems can occur, usually a SESSION COMPONENT is used to collect information from the user or a system, and then to execute a traditional short transaction at the end of the process, after re-checking the preconditions for the transaction.

❊ ❊ ❊

In EJB the same Bean type is used for SERVICE COMPONENTS and SESSION COMPONENTS. Both are implemented as Session Beans. A SESSION COMPONENT in EJB is called a *Stateful Session Bean*. To mark a Session Bean as a *Stateful* you can use the Deployment Descriptor (ANNOTATIONS). SESSION COMPONENTS differ in their lifecycle from SERVICE COMPONENTS. Thus it seems an unlucky design decision to implement both COMPONENT types with one Bean type: Session Beans. In fact, some LIFECYCLE CALLBACKS are only called on Stateful Session Beans and never called on Stateless Session Beans, because the concept of PASSIVATION does not apply to Stateless Session Beans. Separate Bean types with different LIFECYCLE CALLBACK interfaces would have been much better here.

CCM implements SESSION COMPONENTS under the names *Process Component* and *Session Component*. They are defined using the *composition process/session* declaration. Process Components have persistent state, whereas the state of Session Components is transient. Process Components live logically until they are explicitly destroyed. In contrast to the CCM's Entity Components, Process Components do not have a PRIMARY KEY associated with them.

COM+ does not directly support SESSION COMPONENTS, but they can be simulated using the Shared Property Manager and structured storage facilities. In addition, you can create a Moniker for a COMPONENT in different persistent stores. A Moniker allows you keep data longer than a transaction, by storing the state in the persistent store. The COMPONENT can be reconstituted later from the Moniker. Alternatively, it is also possible to keep the session/transaction open by not committing or aborting when returning from a function. The client can access the same instance repeatedly until the session/transaction is committed or aborted.

Summary

A COMPONENT is the central building block for the functional concerns of an application. COMPONENTS always need to run in a CONTAINER, which adds the technical concerns to make the COMPONENT complete from a client's point of view. The CONTAINER controls the lifecycle of the COMPONENT instances. COMPONENTS need to implement a LIFECYCLE CALLBACK interface to make this possible.

As described, there are three different kinds of COMPONENTS:

- SERVICE COMPONENTS
- ENTITY COMPONENTS
- SESSION COMPONENTS

It is not always easy to decide when to use each kind of COMPONENT. Below we give some rough guidelines of when to use each COMPONENT type and contrast some of their features.

Representing business entities

A business entity can be represented in two ways. The first is to use ENTITY COMPONENTS. An ENTITY COMPONENT appears 'object-like' to the client, as it carries state and has operations to work with this state. It is therefore quite intuitive to use. As pointed out in the respective pattern, an ENTITY COMPONENT provides locking, concurrency synchronization and transactional integrity. The CONTAINER can use sophisticated optimization techniques.

However, in many systems the features provided by the CONTAINER for ENTITY COMPONENTS are not required. They don't come free – using ENTITY COMPONENT implies an overhead. For example, consider a stock trading application where – at noon on an average stock exchange day – everybody wants to know the current value of the stock symbols. Access is read-only, but massively parallel. Here, a direct access to the stock symbol's value using a SERVICE COMPONENT might perform significantly better[3]. Another situation arises

3. Of course this depends heavily on the CONTAINER's implementation. As the CONTAINER might implement many optimizations for ENTITY COMPONENTS, they might have advantages over the alternatives.

when you have a procedural API "behind" the business entity. Here it might be unnatural to use ENTITY COMPONENTS. SERVICE COMPONENTS, which provide an API to modify the corresponding data (called a TYPE MANAGER), might be more intuitive.

Implementing sessions

Typical sessions (such as a non-persistent shopping cart on a web site) are usually implemented with SESSION COMPONENTS. They belong to *one* client exclusively. Using SESSION COMPONENTS is fine, but they do not provide persistence. Thus, if a server crashes, or the shopping cart must be available for several days or weeks, it might be better to 'misuse' an ENTITY COMPONENT for this kind of session.

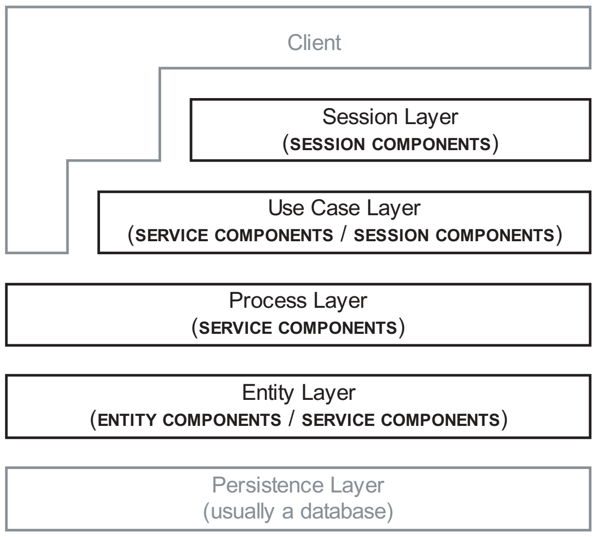
SESSION COMPONENTS also do not scale as well as SERVICE COMPONENTS, as their number depends directly on the number of clients. Therefore it might be necessary to avoid SESSION COMPONENTS completely to achieve a more scaleable application.

Processes or workflows

Most component architectures (with CCM as the notable exception) do not define a specific COMPONENT type for workflows or processes. However, this is not really a problem, because a process usually has the same properties as an entity: it can be active for a long time, it can be identified uniquely, and it can be accessed by several clients, often concurrently. Thus, you will must use ENTITY COMPONENTS to represent processes.

Overall architecture

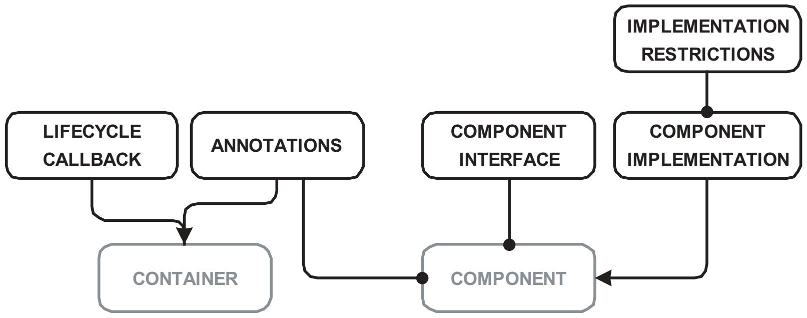
To make system truly reusable, you should explicitly differentiate between generally useful COMPONENTS (usually processes or entities), COMPONENTs that orchestrate well-defined interaction sequences between themselves (usually SERVICE or SESSIONS COMPONENTS) and application- or use-case specific COMPONENTS that are created for a particular application (again, usually SERVICES or SESSIONS). A layered system of these types of COMPONENTS is often the result:



Your reusable entities or processes are never directly exposed to the client. Clients only interact with use-case specific COMPONENTS. This prevents your reusable entities from interface bloat driven by application-specific use cases.

2 Component Implementation Building Blocks

This chapter explains which building blocks make up a COMPONENT. COMPONENTS promise several benefits, but to realize them, COMPONENTS must be implemented appropriately.



Now that you have COMPONENTS that encapsulate specific areas of self-contained application functionality, you need well-defined COMPONENT INTERFACES through which clients can access these COMPONENTS.

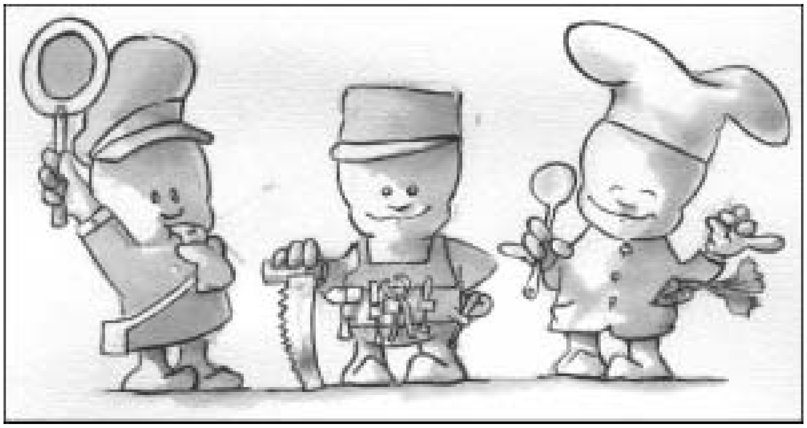
The COMPONENTS and the respective COMPONENT IMPLEMENTATIONS implement only the functional requirements – the COMPONENT IMPLEMENTATION is never accessed by clients directly. Instead, the CONTAINER adds the code to handle the technical concerns necessary to access the instance correctly. Of course the COMPONENTS must tell the CONTAINER what kind of technical concerns should be added to them and how they should be added. ANNOTATIONS are used for this purpose.

The CONTAINER must efficiently manage its resources to provide scalability. This requires optimized management of the COMPONENT instances themselves: the lifecycle of these instances is especially important in this respect. COMPONENTS therefore have to implement a LIFECYCLE CALLBACK interface, which will be used by the

CONTAINER to notify the instances of important events in their lifecycle, allowing instances to react accordingly. To make sure that the COMPONENT IMPLEMENTATION does not interfere with the CONTAINER's work, you might require some IMPLEMENTATION RESTRICTIONS for the COMPONENTS.

Component Interface

You decomposed the functional concerns of an application into COMPONENTS. Your system should be made up of loosely-coupled, collaborating COMPONENTS.



❊ ❊ ❊

A COMPONENT needs to collaborate with other COMPONENTS to be really useful. However, to facilitate reuse, you want to make sure that the COMPONENT and its clients do not depend on the implementation of other COMPONENTS. Furthermore, you want to be able to evolve the implementation of a COMPONENT without affecting other COMPONENTS or clients that use it. Last but not least, you may want to provide multiple alternative implementations for a specific COMPONENT.

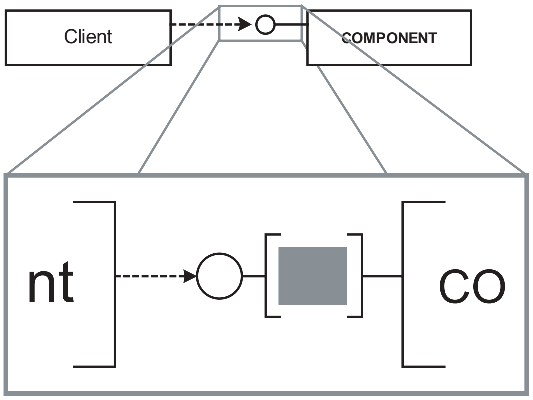
COMPONENTS are introduced to serve as functional building blocks from which you can assemble an application. One of the primary goals of decomposing application functionality into COMPONENTS is to provide a means to evolve each unit of self-reliant functionality independently of other parts of an application. If COMPONENTS directly depend on each other, either from a compilation or a deployment point of view, the use of COMPONENTS as independent building blocks would be questionable.

Instead, what you need is a means to decouple the clients of a COMPONENT from its implementation – clients, whether actual clients or client COMPONENTS, should rely only on a specification of *what* the COMPONENT does, not *how* the COMPONENT does it. This independence of the *how* can even go as far as ensuring that the programming language, or even the underlying operating system, is transparent to the client.

If this is achieved, you can reuse COMPONENTS as black box entities. When you build applications from COMPONENTS, you assemble them based on the functionality they provide. You reuse COMPONENTS as a building blocks, 'wiring' them together to form the complete system.

Therefore:

Provide a COMPONENT INTERFACE that declares all the operations provided by a COMPONENT, together with their signatures and, ideally, their semantics. Use this interfaces as a contract between the client and the COMPONENT. Let clients access a COMPONENT only through its interface.



✵ ✵ ✵

The COMPONENT INTERFACE must be strongly separated from the COMPONENT IMPLEMENTATION. It must also be possible to use a different COMPONENT IMPLEMENTATION with the same COMPONENT INTERFACE, in order to be able to evolve the implementation. Of course, if the implementation changes in a way that requires a change to the interface, because operations are added or signatures are changed, the COMPONENT INTERFACE must be adapted accordingly.

In current practice, a COMPONENT INTERFACE only specifies the signature of the defined operations. The semantic meaning of an operation is usually defined only as a plain text specification. Note that the semantics are not the same as the implementation. You can easily specify the functional semantics of an operation without saying anything about its concrete technical implementation, for example using pre- and post-conditions as proposed in Meyer's design-by-contract [ISE]. As an interface does not formally define its semantics, a client cannot be sure that the COMPONENT actually does what the client expects, because:

- The client's expectations may be wrong, or
- The implementer misunderstood the requirements wrong and implemented the COMPONENT INTERFACE incorrectly.

To enhance the chances for inter-operability, accessing the interface should be standardized. Ideally a binary standard should be provided. This allows clients developed in different programming languages or for different operating systems to access the COMPONENT INTERFACE. Details of remote access to the COMPONENT are handled by the COMPONENT BUS, which hides the details of remote transport.

As the COMPONENT INTERFACE is separated from the COMPONENT IMPLEMENTATION, and because clients will only ever see and depend on the COMPONENT INTERFACE, the COMPONENT IMPLEMENTATION is exchangeable. To achieve this exchangeability in practice, make sure that the COMPONENT INTERFACE does not imply anything about the COMPONENT IMPLEMENTATION.

A COMPONENT can have multiple COMPONENT INTERFACES. If this is the case, you might want to provide a way for a client to query the COMPONENT about its COMPONENT INTERFACES and access them separately, further reducing dependencies. Using version tags, this can help to enable each interface to evolve separately. Moreover, this approach allows a more role-oriented process of software development, in which the interfaces define the different roles that can be performed by a COMPONENT. As these role-specific COMPONENT INTERFACES are a kind of *Adapter* [GHJV94], they allow otherwise unrelated COMPONENTS to be used by the same client, because the role provides the COMPONENT INTERFACE to them expected by the client.

Providing COMPONENT INTERFACES separately from the COMPONENT IMPLEMENTATION has additional advantages, especially for the CONTAINER – it is the basis for many optimizations. The CONTAINER usually generates a COMPONENT PROXY that formally implements the COMPONENT INTERFACE and is used to attach the COMPONENT to the COMPONENT BUS. On the other hand, it invokes LIFECYCLE CALLBACK operations, which are necessary to provide VIRTUAL INSTANCES, in turn the basis for PASSIVATION and POOLING. To actually implement the functional requirements, the COMPONENT PROXY delegates invocations to the COMPONENT IMPLEMENTATION.

Some sources advocate the use of completely generic COMPONENT INTERFACES. For example, such an interface could consist of exactly one operation called *do(taskXML)*: *resultXML* that only operates with XML data. It takes an XML string as parameter, which, conforming to a certain DTD, specifies the task to be executed. The result is again specified in XML.

The advantage of this approach is that a change in the COMPONENT IMPLEMENTATION never requires changes to the COMPONENT INTERFACE, thus you never need to recompile clients. However, the dependencies are still there, they are just not reflected in the COMPONENT INTERFACE, and thus not enforced by the compiler or the CONTAINER. For example, as you still have to change the COMPONENT IMPLEMENTATION to implement new requirements, you still need to reinstall the COMPONENT IMPLEMENTATION, and clients might need adaptation to provide newly required data in XML. Another disadvantage of this approach is that it circumvents some of the mechanisms provided by the CONTAINER. As the operation to be executed is specified as part of the XML data, you cannot use ANNOTATIONS in the way intended. It is thus questionable whether this kind of COMPONENT INTERFACE is actually an advantage.

❊ ❊ ❊

All example technologies use this pattern. Because EJB is limited to Java for COMPONENT development, the COMPONENT INTERFACE is specified using Java's *interface* construct, but with some additional rules and restrictions on the operation parameters used in COMPONENT INTERFACES. From version 2.0 onwards, EJB provides two kinds of interfaces: those used for regular, potentially remote access (already implemented in pre-2.0 EJB), and so-called *Local Interfaces*, that serve as an optimization for Bean-to-Bean communication in the same APPLICATION SERVER.

The two interfaces are completely independent of each other – they can even have different operations. A Bean can have remote only, local only or both kinds of interfaces. EJB Remote Interfaces must inherit from *javax.ejb.EJBObject*. Operations must throw the *java.rmi.RemoteException*. This is necessary to allow the framework to report SYSTEM ERRORS such as failed network communication or resource problems. Local Interfaces must extend *javax.ejb.EJBLocalObject*, and the operations must not throw *RemoteExceptions* – the subset of SYSTEM ERRORS that can still occur in non-distributed settings is signaled using an *EJBException*, a subclass of *RuntimeException.*

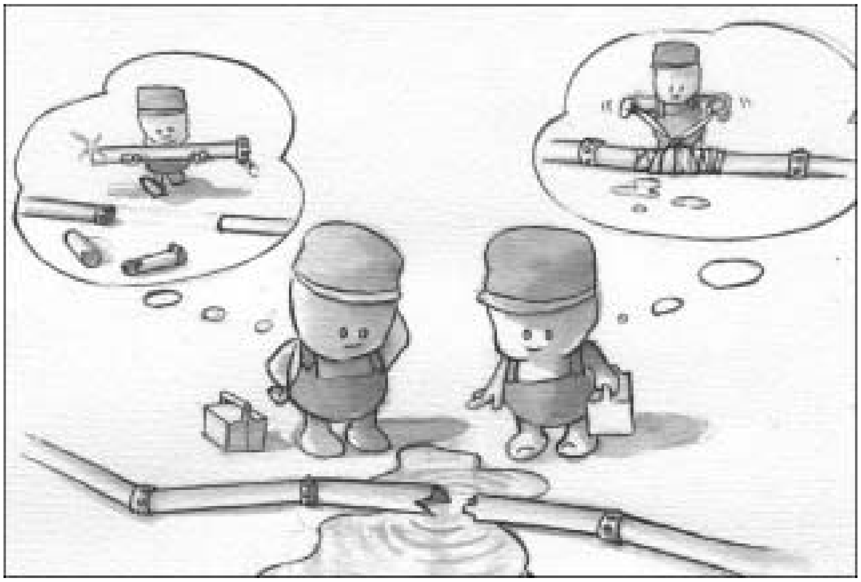
An EJB COMPONENT, a Bean, can only implement one Local and one Remote Interface formally. These interfaces can of course inherit from several base interfaces. Client references to COMPONENT instances are declared using the interface or any of its base interfaces as their static type. Client code only depends on the interface, just as it is described in this pattern.

In CCM, the programmer starts by defining one or more interfaces in IDL, CORBA's Interface Definition Language. These interfaces are not yet related to any COMPONENT. Each could be implemented by an ordinary CORBA object. In the second step, a COMPONENT is defined that *supports* or *provides* one or more of the interfaces. *Supported* interfaces are implemented by the COMPONENT directly. *Provided* interfaces are implemented as a separate facet – a client has to request this interface from the COMPONENT at run-time explicitly. The definition of COMPONENTS as a collection of interfaces is done using Component IDL (CIDL), an extension of IDL that allows a more succinct notation for COMPONENTS. Because IDL and CIDL are programming-language independent and many programming language mappings exist, it is possible to implement CCM COMPONENTS in several different programming languages. Among others these are C++, Java, Ada, Cobol, Smalltalk, and Perl. Of course this is only possible if a CONTAINER for the language is available, which is not the case for any language at the time of writing. Each interface is given a unique repository ID to identify interfaces in the CORBA interface repository. For the ID, several different formats exist. It is up to the developer to ensure its uniqueness over time and space.

In COM+, a COMPONENT can also support multiple interfaces. All interfaces must inherit from *IUnknown*, an interface that allows a client to query the COMPONENT for other supported interfaces. Each interface is identified by a globally-unique interface identifier (GUID) that is generated automatically and guaranteed to be unique. Again, clients are programmed only for the interface, and know nothing about how the interface is implemented. COM+ uses type libraries that are generated from IDL or proxies/stubs to make type information available to clients. Type libraries for the interface offer a sort of COMPONENT INTROSPECTION that is usually used by script language clients.

Component Implementation

You use a COMPONENT INTERFACE to define the business operations for a COMPONENT and to decouple clients from the internals of your COMPONENT.



❊ ❊ ❊

After you have defined the COMPONENT INTERFACE, you need to provide an implementation of the business logic. This has to happen in a way that keeps the implementation decoupled from the COMPONENT INTERFACE and allows the CONTAINER to add the technical concerns.

The COMPONENT INTERFACE provides the declarations of the operations the COMPONENT will provide, and thus forms a contract between clients and the COMPONENT. This functionality has to be implemented somewhere.

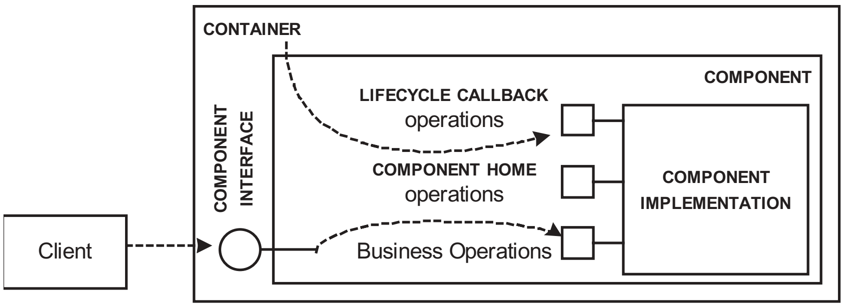
Apart from the core business logic, the following need to be implemented:

- The operations defined in the COMPONENT HOME. The COMPONENT HOME declares operations that allow clients to manage the set of instances of a COMPONENT, for example creating new or finding existing COMPONENT instances.
- The LIFECYCLE CALLBACK operations. LIFECYCLE CALLBACK operations have to be implemented by COMPONENTS to allow the CONTAINER to manage an instance's lifecycle by invoking the operations at specific times.

Note that it is not possible to just 'attach' the COMPONENT INTERFACE directly to the implementations, because the CONTAINER must have a way to add code that takes care of the technical concerns, for example to check the permissions before the operation is called or to start a transaction – just as specified in the ANNOTATIONS.

Therefore:

Use a COMPONENT IMPLEMENTATION that is separate from the COMPONENT INTERFACE. This provides implementations for all the required operations: business, LIFECYCLE CALLBACK, and COMPONENT HOME. These implementations can be implemented in one software entity, or alternatively, distinct entities can be used to implement the different kinds of operations.



❊ ❊ ❊

There are different ways in which the necessary operation implementations can be provided.

Either:

- The operations of all interfaces are defined in one entity, typically a class, or,
- You can use separate implementation entities for the COMPONENT INTERFACE and the HOME INTERFACE.

The COMPONENT IMPLEMENTATION will not be attached to the COMPONENT INTERFACE directly, and it may not even implement the interface by means of the programming language used. To make sure the technical concerns are implemented correctly, it is the CONTAINER's responsibility to attach the COMPONENT INTERFACE to the COMPONENT IMPLEMENTATION. The CONTAINER does this by using a separate entity, the COMPONENT PROXY, that handles the technical aspects and delegates business functionality to the COMPONENT IMPLEMENTATION.

The CONTAINER has to manage the COMPONENT instances efficiently in order to keep resource usage within reasonable bounds. Depending on the type of the COMPONENT, it uses PASSIVATION or POOLING to achieve this. To make this possible, the CONTAINER provides VIRTUAL INSTANCES: a physical COMPONENT instance can represent different logical entities during its lifetime. Because it is the COMPONENT IMPLEMENTATION that keeps the functional state, the physical instances' lifecycles must be managed. The COMPONENT IMPLEMENTATION must therefore provide LIFECYCLE CALLBACK operations and implement them correctly.

If you provide one COMPONENT IMPLEMENTATION that implements both the HOME INTERFACE and the COMPONENT INTERFACE, an instance might be used for different tasks during its lifecycle: to receive invocations of business operations, a COMPONENT IMPLEMENTATION must be assigned to a specific business entity – otherwise an invocation has no 'target'. However, to execute COMPONENT HOME operations, such an assignment is not necessary. COMPONENT HOME operations operate on the set of instances of a COMPONENT, and they have no effect on the instance on which it is called, as with static methods in a class definition. Thus, for example, if POOLING is used in the case of ENTITY COMPONENTS, an instance that is pooled and that is not currently assigned to represent a business entity can be used to execute COMPONENT HOME operations.

As mentioned above, the COMPONENT IMPLEMENTATION usually does not implement the COMPONENT INTERFACE by means of the programming language. However, for each operation declared in the COMPONENT INTERFACE, the COMPONENT IMPLEMENTATION must provide a corresponding method, associated for example by a naming convention. The correspondence between the operation declared in the interface and its implementation is ensured during COMPONENT INSTALLATION – it is not necessarily checked by the compiler. Because of this loose coupling between the COMPONENT INTERFACE and the COMPONENT IMPLEMENTATION, the same implementation can be used with different interfaces, or vice versa. So a new COMPONENT providing new functionality can be created by using an older IMPLEMENTATION, if the methods are similar.

To make sure that a COMPONENT IMPLEMENTATION does not conflict with the CONTAINER and its implementation, IMPLEMENTATION RESTRICTIONS usually need to be put in place.

❊ ❊ ❊

In EJB, there is one artifact, a class, that contains implementations of all the operations of the complete COMPONENT. This includes the business operations declared in the Remote and/or Local Interface, as well as the operations declared in the COMPONENT HOME. LIFECYCLE CALLBACK operations are also implemented in this single class.

Because the CONTAINER uses physical COMPONENT instances for several purposes (see VIRTUAL INSTANCES), not all operations are specific to a certain COMPONENT instance. For example, in the case of Entity Beans, the operations defined in the COMPONENT HOME can be invoked while the physical instance is 'idle', which means that the physical instance does not represent a logical entity. These operations can be considered 'static' with regards to the Bean. EJB 2.0 allows the definition of business operations on the Home Interface. They can be used if a set of instances should be modified or queried.

The COMPONENT IMPLEMENTATIONS have to implement the *javax.ejb.SessionBean* or *javax.ejb.EntityBean* interfaces, which declare the required LIFECYCLE CALLBACK operations for the specific COMPONENT type.

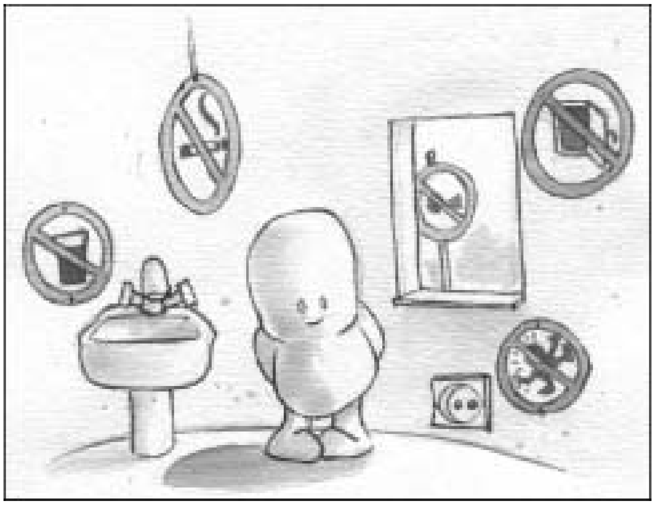
In CCM, the developer has to define which artifacts are responsible for implementing the different interfaces. These artifacts are called *executors*. The executors for the COMPONENT INTERFACE and for the COMPONENT HOME can all be implemented by one class, or, alternatively, in several. CCM implementations are expected to provide abstract base classes or interfaces that the implementation has to implement. As mentioned above, the COMPONENT INTERFACE of a CCM COMPONENT can be made up of different *facets*, each providing a separate interface. The implementation of the facets can also be distributed over several artifacts: each 'facet executor' is called a *segment.* To optimize performance and use of resources, each segment can be managed separately, that is, it can be passivated or pooled separately.

In COM+, the COMPONENT IMPLEMENTATION can be somewhat awkward if it is done at a low level. Microsoft therefore provides the so-called *Active Template Library* (ATL), which provides base classes and templates to help the developer with the implementation. Several additional wizards help even more. Thus, the implementation class (a *CoClass*), provides the implementation of business methods and LIFECYCLE CALLBACKS. If constructor-like initialization is necessary, the COMPONENT has to implement the *IObjectConstruct* interface. Because COM+ provides only SERVICE COMPONENTS, there is no need to provide different create/find/delete operations, as are needed for EJB or the CCM in a COMPONENT HOME.

The default home provides one operation called *CreateInstance()*. This operation has to be implemented in a separate *CoClass* that inherits from *IClassFactory*. Note that COMPONENT IMPLEMENTATIONS in COM+ don't necessarily need to be programming language classes: they just have to be compilable into a DLL that provides the operations defined in the COMPONENT INTERFACE. This is because not all COM+ implementation languages are object-oriented. For example, it is possible to define COM+ COMPONENTS using the C programming language.

Implementation Restrictions

You are using COMPONENTS and let them run them in a CONTAINER. The COMPONENTS' functionality is implemented with a COMPONENT IMPLEMENTATION.



❊ ❊ ❊

The CONTAINER runs several COMPONENT instances in one address space. It takes care of many technical requirements. To make sure this can work, however, the CONTAINER has to make certain assumptions about the behavior of the COMPONENTS. Otherwise, COMPONENTS might behave in ways that would prevent the CONTAINER from handling the technical concerns appropriately.

For example, take a CONTAINER that employs a specific scheduling algorithm to ensure synchronization. If a COMPONENT starts its own threads that run in the background and invoke operations on itself asynchronously, this might indeed compromise the correctness of the application, especially synchronization aspects.

Alternatively, a COMPONENT might receive a method invocation from a client and then block, for example waiting for a socket connection to arrive. This might cause severe problems, because the COMPONENT instance cannot be easily PASSIVATED.

Other examples are COMPONENTS that acquire resources and never release them, or try to access GUI layers on a server that has only a text-based console. The most striking example of this kind of problem is a COMPONENT telling the APPLICATION SERVER to quit.

Therefore:

Impose IMPLEMENTATION RESTRICTIONS on the COMPONENT IMPLEMENTATION. Limit the programmer's freedom over what they can do in the COMPONENT IMPLEMENTATION by preventing them from using language or API features that might interfere with the CONTAINER.

❊ ❊ ❊

The actual restrictions depend very much on the underlying language and its features, as well as the component architecture. Some typical candidates, however, can be given generically:

- Threading is usually done by the CONTAINER, so everything that has to do with threads, synchronization or blocking is usually not allowed.
- CONTAINER and APPLICATION SERVERS can run on machines with stripped-down user interface. Everything that would access any kind of console or user interface directly is therefore usually disallowed.
- CONTAINERS can be clustered or replicated to facilitate load balancing or fail-over. Thus anything that assumes that all COMPONENTS run in one address space is not permitted, for example typical implementations of the *Singleton* pattern [GHJV94].
- A COMPONENT must be deployable on any machine with a suitable CONTAINER. As a consequence, a COMPONENT must never access external resources directly – that is, without the CONTAINER's knowledge – because this might not be possible on another machine.

An important contribution of a component architecture is that it provides a well-defined programming model for its users. The IMPLEMENTATION RESTRICTIONS are an important part of this model and should thus be well documented and well explained.

Limitations or restrictions are even more useful if you can enforce them. In the context here, the CONTAINER should be able to make sure that a COMPONENT sticks to the restrictions. Ideally, the CONTAINER should detect violations of the restrictions during COMPONENT INSTALLATION, preventing non-conforming COMPONENTS from being installed in the first place. However, the problem is that you usually cannot enforce conformance to all of these rules. By defining the IMPLEMENTATION RESTRICTIONS, however, you at least give COMPONENT developers the chance to create 'well-behaving' COMPONENTS. If they don't stick to the rules, you at least know who to blame!

In some programming languages, such as Java, the security features can help to enforce some of the limitations. For example, you can disallow a COMPONENT IMPLEMENTATION from calling *System.exit(0)*.

Of course, by limiting the freedom of the COMPONENT implementer, some well-known and proven programming techniques might not be usable in COMPONENT implementations. This might require awkward constructs in some circumstances:

- Log messages have to be output to a database, a message queue, or a file configured to be a MANAGED RESOURCE.
- Starting a thread in order to poll for a result is not possible – this has to be done by an external service that eventually invokes the COMPONENT.

❊ ❊ ❊

In EJB there are a couple of things that are not allowed in COMPONENT IMPLEMENTATIONS, including:

- Creating new threads
- Using server sockets for communication (because they would compromise server clusters)
- Accessing files directly

- Using custom class loaders for loading classes
- Blocking I/O

The EJB specification [SUNEJB] lists all these limitations. Most of the restrictions are not checked by the CONTAINER, and therefore the developer is responsible for following these rules. Nevertheless, Java security is used to disallow several of the most problematic features.

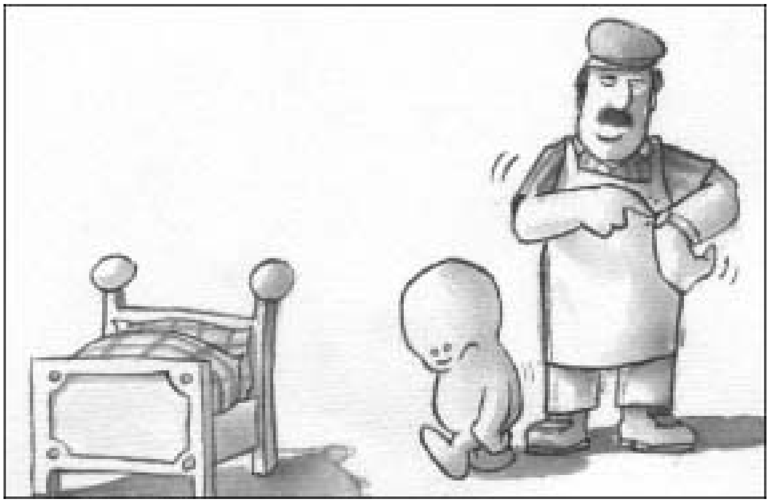
As CCM is programming-language independent, CONTAINERS for each language will impose different restrictions on the COMPONENT IMPLEMENTATIONS. However, because CCM is upward compatible with EJB, the restrictions on Java implementations will be conceptually the same as for EJB.

In COM+ there are also several restrictions. For example, threading is restricted depending on the *apartment* type[1] in which the COMPONENT lives. In addition, because a component is at least logically removed after the end of a transaction, it is not allowed to keep state between transactions.

1. Apartments are a way to define and implement different threading policies for COMPONENTS.

Lifecycle Callback

Your COMPONENTS live in a CONTAINER, and you plan to DISTINGUISH IDENTITIES. You use INSTANCE POOLING or PASSIVATION to keep resource consumption acceptable.



❊ ❊ ❊

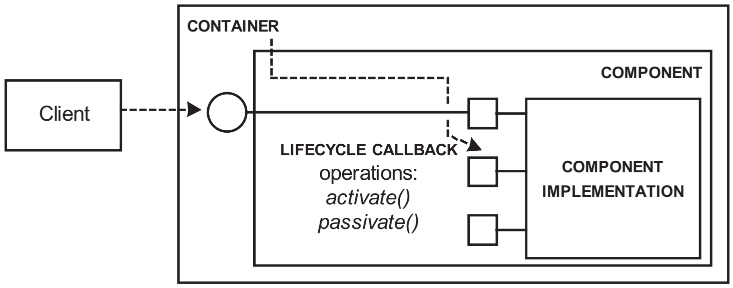
COMPONENT instances that live in a CONTAINER need to be initialized after they have been created, and they might have to be de-initialized before they are destroyed. This is commonly known as *lifecycle management*. Moreover, the CONTAINER uses advanced resource management techniques such as POOLING or PASSIVATION, which requires an even more sophisticated lifecycle management for the COMPONENT instances.

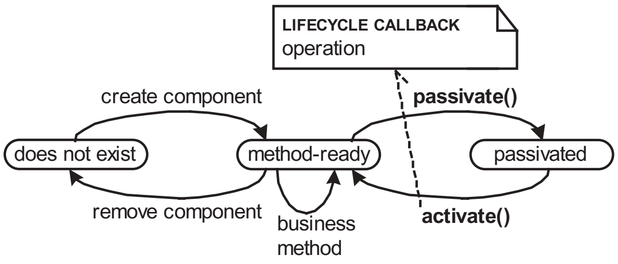
For example, if you use POOLING with ENTITY COMPONENTS, a physical COMPONENT instance will represent different logical entities over its lifetime. When the instance is created, it must be initialized at least with the COMPONENT CONTEXT that allows the instance to access the MANAGED RESOURCES it requires. It may then be placed into a pool, where it stays until the CONTAINER requires a COMPONENT instance to serve a client request. Serving a request means that the instance now has to represent a certain logical identity. This in turn means that the instance has to update its state from a database. After it has been used to serve the client request, the COMPONENT might be put back into the pool, or it might be passivated. Both activities require the release of previously-acquired MANAGED RESOURCES.

As can be seen from the discussion, a COMPONENT instance has to cooperate with the CONTAINER in order to be used efficiently.

Therefore:

Require a COMPONENT IMPLEMENTATION to implement a set of well-defined LIFECYCLE CALLBACK operations. Each of these operations is invoked by the CONTAINER at a well-defined point during the life-cycle of an instance, to make the instance do the things the CONTAINER expects. To make this work, the COMPONENT IMPLEMENTATION has to provide a correct implementation of these operations.





✲ ✲ ✲

This pattern forms the basis for lifecycle management of the COMPONENT instances through the CONTAINER. While the lifecycle has to be well-defined, and the LIFECYCLE CALLBACK operations must have clear semantics, there is still considerable freedom for the CONTAINER over when to call the operations. The pattern allows the CONTAINER to implement different policies for POOLING, activation, PASSIVATION, reuse, or load-balancing. The programmer need not care about these policies as long as he implements the LIFECYCLE CALLBACK operations semantically correctly.

Typical tasks that have to be taken care of inside the LIFECYCLE CALLBACK operations include acquisition and release of resources used by the instance and changes to the identity of a COMPONENT, which requires an update of an instance's state. Typical lifecycle phases at which LIFECYCLE CALLBACK operations have to be provided are:

- After creation – before destruction
- Before passivation – after activation
- Before being placed into a pool – after being taken from a pool
- After assignment of a new logical identity – before de-assignment of a logical identity

Note that LIFECYCLE CALLBACK operations form a contract between the CONTAINER and the COMPONENT, just as the COMPONENT INTERFACE is a contract between the client and the COMPONENT.

While the integration of technical aspects with business methods defined in the COMPONENT INTERFACE usually requires a GLUE-CODE LAYER specifically generated for a particular COMPONENT (INTERFACE), the LIFECYCLE CALLBACK operations are defined per kind of COMPONENT (SERVICE, SESSION, ENTITY) and can be invoked generically by the fixed parts of the CONTAINER.

Although the LIFECYCLE CALLBACK operations usually have to be implemented manually by the COMPONENT implementer, in some cases they can be created automatically by an IMPLEMENTATION PLUG-IN. A typical example is the persistence code for ENTITY COMPONENTS – once the mapping to the (relational) database has been defined, the implementation can be generated.

❊ ❊ ❊

EJB defines several LIFECYCLE CALLBACK operations for a Bean. For example, Entity Beans' *setEntityContext()* is used by the CONTAINER to supply the COMPONENT with the COMPONENT CONTEXT after assigning it a logical identity (see VIRTUAL INSTANCE). In the case of Entity Beans, *ejbLoad()* and *ejbStore()* are used to synchronize the COMPONENT's persistent state with the database. *ejbActivate()* and *ejbPassivate()* are used to handle activation and passivation with regard to INSTANCE POOLING. *ejbCreate()* and *ejbRemove()* handle the creation and deletion of logical identities.

Beans can only be compiled when these operations are implemented, as they are defined in the *javax.ejb.EntityBean* interface. However, the *ejbCreate()* method depends on the declaration of the *create()* methods in the HOME INTERFACE, so this method is checked during deployment using COMPONENT INTROSPECTION. It is the responsibility of the COMPONENT developer to implement the LIFECYCLE CALLBACK methods. The only exceptions to this rule are those operations that deal with database access, such as *ejbLoad()*. If you use Container-Managed Persistence (CMP) the CONTAINER generates the code for these LIFECYCLE CALLBACK operations automatically using an IMPLEMENTATION PLUG-IN.

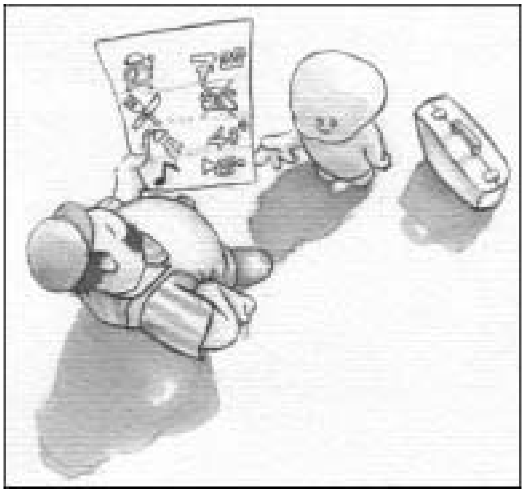
CCM provides roughly the same LIFECYCLE CALLBACK operations as EJB. The set of operations of course depends on the COMPONENT type. All COMPONENT types have operations to set the COMPONENT CONTEXT. The SESSION COMPONENT supports PASSIVATION through *activate()* and *deactivate()* – see INSTANCE POOLING and PASSIVATION for details. For ENTITY COMPONENTS, CCM provides the same operations as EJB, but with other names.

EJB and CCM provide an interesting LIFECYCLE CALLBACK interface, called *Synchronization* in CMM and *javax.ejb.SessionSynchronization* in EJB. It has two operations, *before_completion()* and *after_completion()*. These are used to notify a COMPONENT about transaction boundaries. These LIFECYCLE CALLBACK operations can be used to do an explicit in-memory roll-back for the state of the COMPONENT if an operation fails and the transaction is required to be rolled back.

In COM+, poolable COMPONENTS can be notified before they are deactivated and after they have been activated again. If they wish to be informed, they have to implement the *IObjectControl* interface. This provides three operations, of which two, namely *activate()* and *deactivate()*, are related to lifecycle management. The COM+ run-time calls them whenever necessary.

Annotations

You use COMPONENTS to implement the functional concerns and a CONTAINER to take care of the technical concerns.



❊ ❊ ❊

In conformance with the principle of SEPARATION OF CONCERNS, the CONTAINER is responsible for providing the technical concerns. To be applicable for all COMPONENTS, the CONTAINER implements technical concerns in a flexible, generic way. They have to be configured to a certain extent to ensure that they are suitable for a specific COMPONENT. However, we don't want to code this information explicitly for each COMPONENT as part of its COMPONENT IMPLEMENTATION.

Generality and reusability always imply that there is something to configure or to specify how a generic artifact should be used. This is also true for the CONTAINER. Consider transactions: it is not enough that the CONTAINER *can* handle transactions for you – you have to tell it *how* it should handle transactions for a specific COMPONENT.

There are many ways in which transactions can be handled:

- An operation can run in a transactional context
- An operation might be *required* to run within a transaction
- An operation might be required to run *in its own*, *new* transaction
- An operation might not be allowed to run in a transaction at all.

The CONTAINER can provide all these alternatives for you, but you have to help it decide which to use.

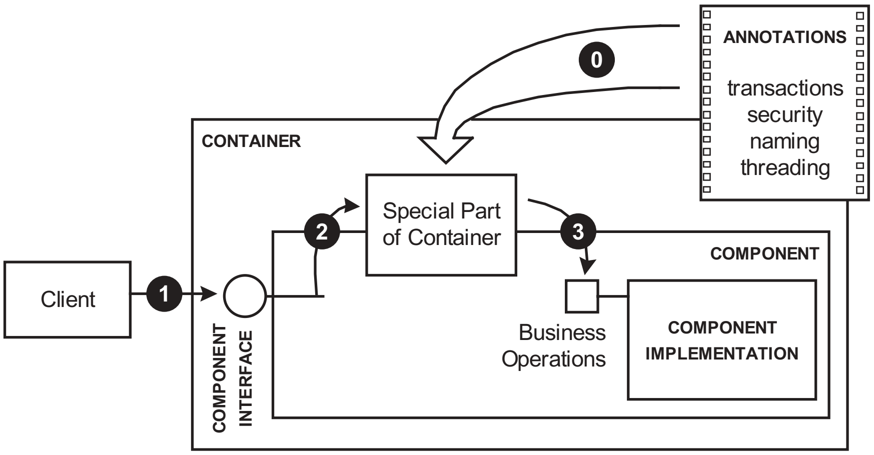
Another example is security. The CONTAINER can enforce permission checks on COMPONENT instances. But you need to tell the container, what these permissions are. Who is allowed to create a COMPONENT instance? Who is allowed to invoke which operation on which instance?

The technical concerns realized by the CONTAINER can usually be characterized by the following properties:

- There is only a limited number of possible options for each concern – for example, a transaction can be *required*, *allowed*, or *forbidden* for an operation – or the options can be defined with regard to a set of data items and options, such as: user *XY* is allowed to invoke the operation *abc()*.
- Once the above decision is made, the implementation of the concern follows simple, predefined rules or structures.

Therefore:

Provide a means for the COMPONENT developer to annotate the COMPONENTS with the technical concerns. Such ANNOTATIONS are a part of the COMPONENT, but separate from the COMPONENT IMPLEMENTATION and the COMPONENT INTERFACE. The ANNOTATIONS specify how the CONTAINER should apply the technical concerns for the COMPONENT. ANNOTATIONS are not program code, they are a high-level specification. It is the CONTAINER's job to decide on the concrete implementation.



0: ANNOTATIONS are used to "configure" the CONTAINER
1: Client invokes operation on the COMPONENT INTERFACE
2: Invocation is intercepted by the CONTAINER
3: Invocation is forwarded to actual COMPONENT IMPLEMENTATION

❊ ❊ ❊

ANNOTATIONS do not relieve the COMPONENT developer from thinking about technical concerns. He still has to make reasonable decisions about what the CONTAINER should do with his COMPONENT. But the programmer is not required to *code* these things. As described in SEPARATION OF CONCERNS, this provides significant advantages regarding people, responsibilities, performance optimizations, reuse, and more.

It is important to understand that the CONTAINER can't do wizardry and make clever decisions. Wrong specifications in the ANNOTATIONS can have consequences that are serious in the overall application. For example, transaction attributes can be very important regarding the correctness of an application.

It is usually possible to use a GUI-based tool to help programmers to specify ANNOTATIONS, providing predefined options for selection. COMPONENT INTROSPECTION can help in the creation of such a tool, and in allowing the CONTAINER to verify the specification the ANNOTATIONS. COMPONENT INTROSPECTION allows external entities such as tools or the CONTAINER to access information about a COMPONENT, such as a list of operations and their parameters.

The CONTAINER can implement ANNOTATIONS in different ways. To ensure reasonable performance, and to adapt a specific COMPONENT to the generic parts of the CONTAINER, the CONTAINER generates a GLUE CODE LAYER. This usually occurs during COMPONENT INSTALLATION. Because the programmer only specifies what the CONTAINER should do, but not how, a lot of freedom remains for optimizations on the part of the CONTAINER.

Apart from the obvious examples of transactions and security, other information is often part of ANNOTATIONS:

- The name of the COMPONENT in the NAMING system
- The names and types of MANAGED RESOURCES that are accessed by the COMPONENT
- Dependencies on other COMPONENT INTERFACES
- A particular threading model
- Information for optimizations that cannot be implemented by the CONTAINER without additional, perhaps domain-specific knowledge
- Quality of service parameters, that might for example configure the COMPONENT BUS.

In the COM-world, ANNOTATIONS are also known under the names of *declarative programming* and *attribute-based programming*.

❖ ❖ ❖

In EJB, the Deployment Descriptor is an instance of the ANNOTATIONS pattern. It contains security policies, specification of transactional properties, and other general information such as whether a Session Bean is stateful or stateless, the name of the Bean in the NAMING system (JNDI), et cetera. For Entity Beans it also contains information for Container-Managed Persistence, if used, and about relationships among Entity Beans, although these are items are partially vendor-specific.

In EJB 1.0 the Deployment Descriptor used to be a serialized Java object. This lead to many problems over portability and handling. This is why in EJB 1.1 and subsequent versions the Deployment Descriptor is an XML file.

The Deployment Descriptor must be supplied together with the COMPONENT IMPLEMENTATION and the COMPONENT INTERFACE for deployment in a J2EE APPLICATION SERVER. The Deployment Descriptor can be created manually with a normal text editor, although almost all EJB server vendors provide GUI tools to facilitate the process of setting up the Deployment Descriptors. These tools use COMPONENT INTROSPECTION to gain information about the COMPONENT. Whether it is more appropriate to use a tool or edit the Deployment Descriptors manually depends very much on the size of the project and the development process/tools used by the development team.

A CORBA component descriptor serves the purpose of ANNOTATIONS in CCM. It is XML-based, like the Deployment Descriptor in EJB. A CORBA component descriptor provides, among other things, information on a COMPONENT and all its supported and provided interfaces, as well as threading, transaction and security policies. The provided, supported and REQUIRED INTERFACES are referenced using their interface repository identifier, allowing additional details about the interfaces to be retrieved from the repository.

COM+ also uses ANNOTATIONS for several aspects of a COMPONENT's behavior. In this context the concept is known as *attribute-based programming*. The attributes are stored in a special configuration database called the COM+ *catalog*. They can be specified when the COMPONENT is installed, using a tool called the Component Services Explorer. The aspects of a COMPONENT that can be configured by attributes are transactions, synchronization, pooling, the construction string (initialization data), just-in-time activation, and security, among others.

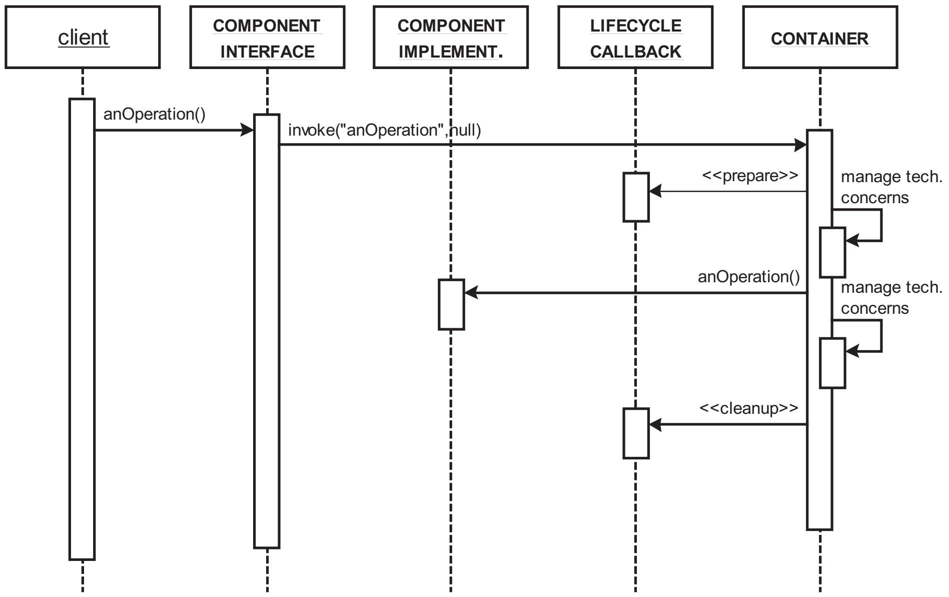
Summary

This chapter describes the building blocks of COMPONENTS from the view of the COMPONENT developer. To make a COMPONENT work, and to realize the promises made by COMPONENT technologies, COMPONENTS have to be built in a specific way.

The following illustration shows how the patterns described in this chapter fit together:

- The COMPONENT IMPLEMENTATION is responsible for implementing the required functionality without worrying about technical concerns.
- The exact configuration of technical concerns for the specific COMPONENT is specified by the COMPONENT developer in the ANNOTATIONS and implemented by the CONTAINER.
- The client, which can also be another COMPONENT, accesses a COMPONENT only through its COMPONENT INTERFACE. The COMPONENT INTERFACE is separate from the COMPONENT IMPLEMENTATION, resulting a low level of coupling.
- The COMPONENT IMPLEMENTATION must provide a LIFECYCLE CALLBACK interface that is used by the CONTAINER to control the lifecycle of COMPONENT instances in the face of optimization and resource management techniques.
- To make sure a COMPONENT IMPLEMENTATION behaves well and can be executed in the CONTAINER without interference with the CONTAINER, IMPLEMENTATION RESTRICTIONS are established and must be followed by the COMPONENT IMPLEMENTATION.

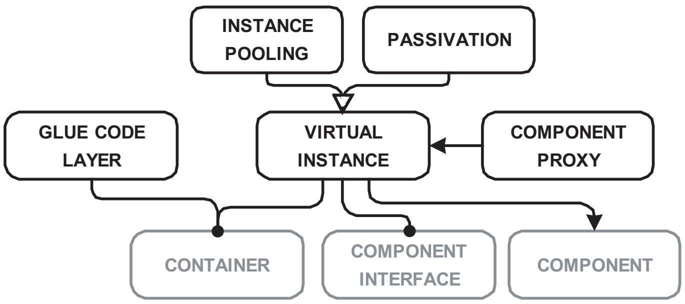
The dynamics of the collaboration among the artifacts are illustrated, to some degree, by the following sequence diagram:



When a client invokes an operation on a COMPONENT, it actually invokes the operation on the COMPONENT INTERFACE. The interface, or some generated implementation, forwards the request to the CONTAINER. This prepares a suitable COMPONENT IMPLEMENTATION and manages technical concerns, for example by starting a transaction or checking security issues. The initial operation is then executed by the COMPONENT IMPLEMENTATION. After that, the CONTAINER again handles technical concerns, for example by committing a transaction, and finally cleans up the instance.

3 Container Implementation Basics

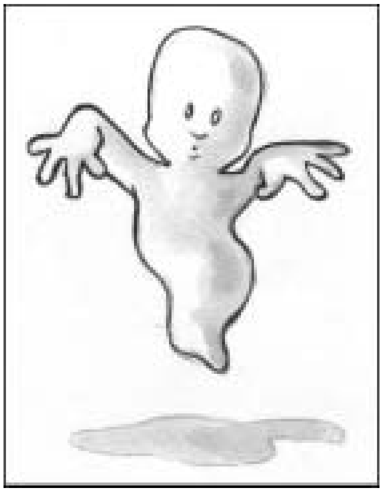
Patterns about component containers could fill a book of their own – their implementation is based on several non-trivial techniques. However, to understand component architectures, it is useful to have at least some understanding of the basic building blocks that make up a container.



The single most important technique that a CONTAINER uses is the VIRTUAL INSTANCE. Logical COMPONENT identities and the programming language objects representing them have completely unrelated lifecycles. To implement this, a COMPONENT PROXY is used to manage the relationships between these two identities. When the proxy is in place, INSTANCE POOLING and PASSIVATION are commonly used as resource optimization techniques. Last but not least, a CONTAINER usually needs a GLUE CODE LAYER to adapt its generic parts to specific COMPONENTS. The COMPONENT PROXY is usually part of the GLUE CODE LAYER.

Virtual Instance

You have separated the COMPONENT INTERFACE from the COMPONENT IMPLEMENTATION. Your system design results in potentially many COMPONENT instances, especially when using ENTITY COMPONENTS.



❊ ❊ ❊

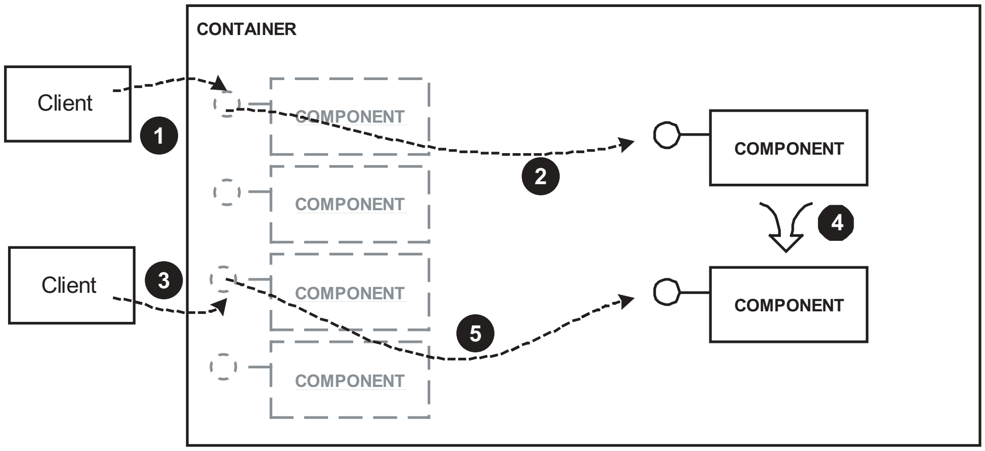
Using COMPONENTS might lead to many COMPONENT instances, especially when every business entity is represented by an ENTITY COMPONENT. Each COMPONENT instance requires memory and other CONTAINER or system resources. This creates the danger that for large numbers of instances your CONTAINER can run into resource problems, especially shortage of memory.

Imagine a customer relations management (CRM) system for a big company that uses an ENTITY COMPONENT to represent customers. If the company has 10,000 customers, this results in 10,000 customer COMPONENT instances. Logically, each customer exists all the time, so all instances must be available to clients all the time. However, it is obviously not possible actually to keep all instances in memory simultaneously.

The same problem holds for SESSION COMPONENTS. For example, in an Internet shopping application you cannot keep all currently-open shopping cart sessions in memory all the time.

Therefore:

Give clients the illusion that every COMPONENT instance is always available by using VIRTUAL INSTANCES. This requires logical and physical COMPONENT identities to be distinguished. A physical COMPONENT instance and its logical identity – the thing the physical instance represents – must be separated. When a client invokes an operation on a logical instance, the CONTAINER has to make sure that a suitable physical instance is available to handle the request.



1: A client invokes an operation on a VIRTUAL INSTANCE
2: CONTAINER forwards invocation to actual physical instance
3: A client invokes an operation on another VIRTUAL INSTANCE
4: CONTAINER makes sure a physical instance is assigned to the corresponding VIRTUAL INSTANCE
5: Invocation is forwarded to the instance

❊ ❊ ❊

While the client has the illusion that every instance is active all the time, the total number of real instances in memory at any given time will be probably much lower than the number of logically active instances. The workload and memory usage of your CONTAINER will therefore drop significantly. In practice, it is not possible to build reasonably-sized component-based systems without using this pattern.

VIRTUAL INSTANCES can come in at least two flavors. For SESSION and ENTITY COMPONENTS, the CONTAINER can use INSTANCE POOLING. It keeps a pool of prepared physical instances and assigns them to logical entities when needed. LIFECYCLE CALLBACK operations are used to prepare the physical component instances for their new assignment.

In case of SESSION COMPONENTS, PASSIVATION can be used. This means that physical instances that have not been used for a given time are removed from memory, and their state stored on disk or in a database. Again, LIFECYCLE CALLBACK operations are used to prepare instances for their upcoming PASSIVATION.

Implementation of this pattern requires that the CONTAINER can intercept incoming method invocations for such a VIRTUAL INSTANCE and prepare a physical instance for service. A COMPONENT PROXY is used for this purpose.

❊ ❊ ❊

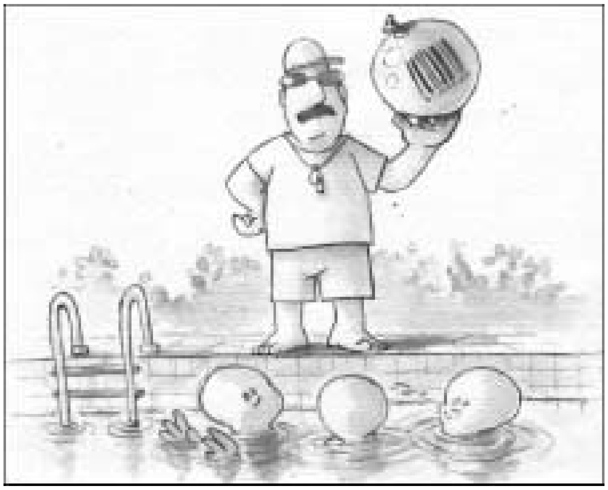
In EJB, the CONTAINER usually uses INSTANCE POOLING and PASSIVATION to implement VIRTUAL INSTANCES, depending on the Bean type. Except for the required implementation of some LIFECYCLE CALLBACK operations, the mechanics are mostly invisible to the COMPONENT developer, but crucial for the scalability characteristics of EJB systems.

In CORBA, the Portable Object Adapter (POA) allows exactly the same to be achieved for CORBA object instances. A CORBA object is the logical instance. A *servant* is a physical object that 'impersonates' the logical CORBA object. Each CORBA object is associated with a POA instance whenever it has to handle invocations. The assignment of servants to CORBA objects can be controlled by different policies, which is exactly what VIRTUAL INSTANCE mandates. Among others, POAs provide policies for activation and deactivation, to handle the swapping of identities. The *Evictor* pattern describes some possible implementations [HV99]. The same technique is also used in CCM.

In COM+, this pattern is also implemented. This is easily illustrated by the fact that COM+ allows just-in-time activation, a feature in which the physical component instance is created only when a request arrives. Pooling is also possible and can be enabled by configuration.

Instance Pooling

You run your COMPONENTS in a CONTAINER. You want to implement VIRTUAL INSTANCES for ENTITY and SERVICE COMPONENTS.



❊ ❊ ❊

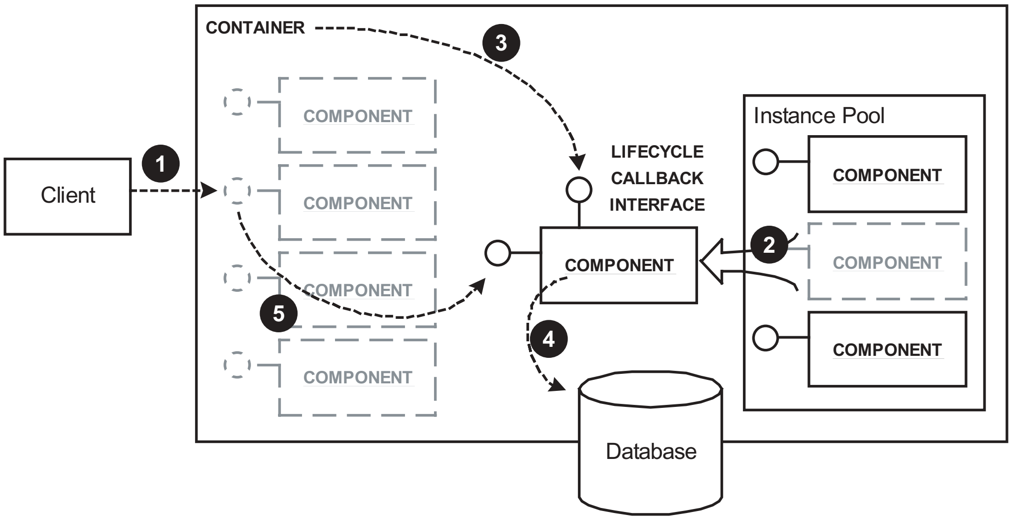
Creating and destroying physical COMPONENT instances in the CONTAINER is expensive, because a COMPONENT instance implies a significant overhead – it has technical and functional state, connections to external resources, and more. Even though this overhead is strongly related to certain points in the lifecycle of the COMPONENTS, it is necessary to reduce it as much as possible.

A physical COMPONENT instance is not just a simple object. It is connected to the CONTAINER's infrastructure, it is connected to the COMPONENT BUS, and it might be associated with *transaction contexts* and be part of a certain threading model. If we want to implement the concept of VIRTUAL INSTANCES, it is therefore certainly not an option to create a physical instance for each method invocation and discard it afterwards, because the complete environment of the COMPONENT would have to be set up and destroyed as well. Keeping physical instances for all VIRTUAL INSTANCES active all the time is also not an option, however.

Thus, a kind of instance 'reuse' must be provided. This seems quite simple for SERVICE COMPONENTS, because they are stateless and therefore all equal anyway. It is not that simple for ENTITY COMPONENTS, because they have state.

Therefore:

Create a pool of physical COMPONENT instances in the CONTAINER. When a VIRTUAL INSTANCE is accessed by a client, assign one of the pooled instances to the VIRTUAL INSTANCE to handle the request. Return the instance to the pool after handling the request. Use LIFECYCLE CALLBACKS to notify a pooled instance of upcoming state changes. The implementation of the LIFECYCLE CALLBACK operations has to take all measures to make sure it represents the VIRTUAL INSTANCE correctly.



1: A client invokes an operation on a VIRTUAL INSTANCE
2: CONTAINER takes an instance from the instance pool
3: CONTAINER invokes suitable LIFECYCLE CALLBACK operations to prepare the instance
4: Instance load its data from the database
5: Invocation is forwarded to the instance
Later: COMPONENT instance will be put back to the pool

❊ ❊ ❊

If INSTANCE POOLING is used by the CONTAINER, a specific physical instance can be in two states: *pooled* or *active*. In the pooled state an instance has no logical identity. No method invocations can reach the instance while it is *pooled*. In the *active* state, an instance represents a logical identity, which means that it has to have the functional state of the logical identity. The implementation of the LIFECYCLE CALLBACK operations must ensure that this state is set up correctly when the transition from *pooled* to *active* is made.

INSTANCE POOLING is usually used for SERVICE and ENTITY COMPONENTS. For SERVICE COMPONENTS, pooling is quite simple. All instances are equal, thus they can be exchanged and assigned randomly. No functional state must be set up – the CONTAINER can implement this very efficiently. The CONTAINER can even use several physical instances to handle requests for one VIRTUAL INSTANCE to enable load balancing – this is one of the reasons why the concept of a stateless SERVICE COMPONENT has been introduced.

For ENTITY COMPONENTS, implementing INSTANCE POOLING is not so simple, for two main reasons:

- Firstly, the LIFECYCLE CALLBACK operations need to be implemented in a way that loads the persistent state of the logical entity it represents before client requests arrive at the instance
- Secondly, synchronization must be enforced if several clients access the instance concurrently.

On the other hand, using INSTANCE POOLING for ENTITY COMPONENTS has the greatest benefit in terms of memory usage optimization, because the number of logical instances can be very high. In the worst case, one instance row in a relational database maps to one ENTITY COMPONENT.

Another opportunity exists for significant optimization – the CONTAINER can decide on its own how long logical/physical assignments should be kept. If the CONTAINER waits before putting an instance back into the pool, subsequent invocations on the same logical instance can be handled even more efficiently, because a physical instance is already assigned. The COMPONENT developer can assist the CONTAINER in this decision by specifying this behavior in the ANNOTATIONS.

Note that the physical COMPONENT instances that are subject to INSTANCE POOLING stay in memory all the time. Only the total number of physical instances necessary for the overall system is reduced – they 'impersonate' different logical identities over time. Because the physical instances are in memory all the time, a physical instance can keep its resource connections and the rest of its environment.

Usually, the COMPONENT PROXY is used to invoke LIFECYCLE OPERATIONS before the invocation itself is forwarded to the physical instance. It serves as the target for method invocations before a physical instance is assigned.

❊ ❊ ❊

EJB provides INSTANCE POOLING for Stateless Session Beans and for Entity Beans. For Stateless Session Beans, INSTANCE POOLING is simple – as they don't carry any state, they don't need to be ′filled with identity′ when the CONTAINER gets an instance from the pool to handle a request, so no LIFECYCLE CALLBACKS are necessary in this case.

For Entity Beans, the situation is not so simple. EJB requires Entity Beans to implement the LIFECYCLE CALLBACK methods *ejbLoad()* and *ejbStore()*. These operations are called by the CONTAINER to synchronize the state of an Entity Bean instance with the persistent store. The CONTAINER therefore has the ability to change a instance's identity and its associated state during the instance's lifetime. For example, when an Entity Bean is taken from the pool and it has to 'impersonate' a new logical entity, the following steps take place:

- First, the CONTAINER sets a new PRIMARY KEY in the COMPONENT CONTEXT.
- *ejbActivate()* is then called to notify the Entity Bean that it is put in the active state.
- It then calls *ejbLoad()* to make the Entity Bean load its new state, the state that is associated with the PRIMARY KEY in the context.

- The *ejbStore()* method is called to write the data back into the database. This happens at least at the end of the transaction, but might happen earlier.
- When the *container* decides to put the physical entity back into the pool, *ejbPassivate()* is called.

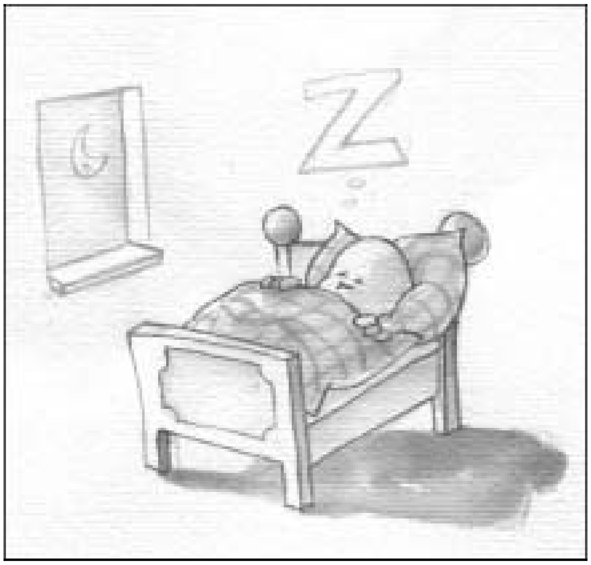
Note that INSTANCE POOLING is not intended to be used with Stateful Session Beans, whose lifecycle does not allow for that – they use PASSIVATION instead.

In CCM, INSTANCE POOLING is a feature that is already available through the POA. Programmers must develop and register suitable *servant managers*. In CCM, the CONTAINERS already provide such servant managers, leaving only the necessary LIFECYCLE CALLBACK operations to be implemented in the COMPONENT IMPLEMENTATION. CCM provides different policies to define how many servants – that is, implementation instances – are actually used to realize different logical instances over time. One option is INSTANCE POOLING, other options include one servant for all instances or a specific servant for each logical instance.

In COM+, you can either instantiate a number of instances which live until the CONTAINER is shut down, or you can use just-in-time (JIT) activation, which is a form of PASSIVATION. If JIT-activation is used, many COMPONENT instances are created and destroyed over time. To avoid potential performance problems, COM+ provides instance pooling for JIT-activated COMPONENTS. To be poolable, a COM+ COMPONENT must meet several requirements: among others, it must be stateless. Pooling can be configured with a minimum and maximum instance count for the pool.

Passivation

You run your COMPONENTS in a CONTAINER. You want to implement VIRTUAL INSTANCES for SESSION COMPONENTS.



❊ ❊ ❊

Providing VIRTUAL INSTANCES for SESSION COMPONENTS is different from ENTITY COMPONENTS, because a SESSION COMPONENT's state has no persistent representation in a database. There are nevertheless periods of inactivity when the client invokes no operations on the instance. You might want to remove these temporarily unused instances from memory to save resources.

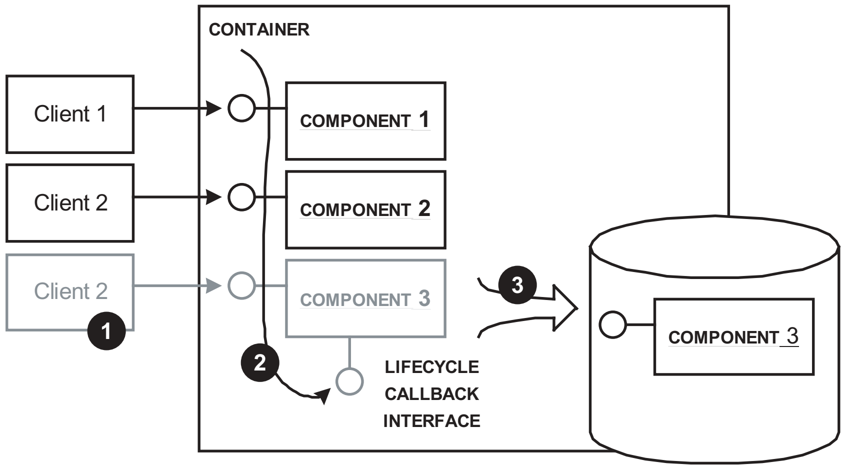
ENTITY COMPONENTS can use INSTANCE POOLING easily, because their state is just a representation of persistent database data. INSTANCE POOLING 'pages in' the required data into memory.

In the case of SESSION COMPONENTS, no persistent representation of the state is available, but in contrast to SERVICE COMPONENTS, they have state. Typical examples are a shopping cart or a long process.

Many instances of SESSION COMPONENTS might be active at the same time. As SESSION COMPONENTS often represent a client session, it is typical to have as many instances as there are concurrent clients. One example would be the shopping carts in an e-commerce application. So some kind of optimization is needed.

Therefore:

Allow the CONTAINER to remove COMPONENT instances temporarily from memory when they are not accessed for a specific period. The state of such a passivated COMPONENT must be stored persistently. When the instance is accessed again, the state must be reloaded. Make sure that this works transparently from the COMPONENT developer's viewpoint.



1: A client keeps a reference to an instance, but does not invoke any operations on it
2: The CONTAINER calls specific LIFECYCLE CALLBACK operations to prepare the instance for PASSIVATION
3: The instance is passivated by storing its state in a database
Later: client can invoke operations again, instance is reactivated from database

This pattern might look similar to INSTANCE POOLING, but it is not. Although both patterns have the same goal in trying to implement VIRTUAL INSTANCES, the techniques are different. INSTANCE POOLING uses a fixed number of COMPONENT instances that 'impersonate' different logical identities during their lifetime. The COMPONENT contains code that stores/loads its state explicitly in its LIFECYCLE CALLBACK operations.

In the case of PASSIVATION, a COMPONENT instance – the programming language object(s) – that is not used for a specific period is evicted from memory until it is needed again. This happens transparently to the programmer. The persistent representation is not defined by the COMPONENT developer, and the COMPONENT does not contain any code to store/load the state.

There is another important difference: in contrast to instances that are subject to INSTANCE POOLING, in PASSIVATION the physical COMPONENT instances are really evicted from memory, so their connections to resources must be broken before they are passivated and reconnected upon reactivation. LIFECYCLE CALLBACK operations must be provided to manage this, although the CONTAINER might manage some parts automatically.

PASSIVATION usually involves a significant overhead, comparable to paging in operating systems. The need to passivate should therefore be avoided whenever possible. To enable this, the server should have enough memory to keep instances active, at least for short-lived instances. Heavy passivation is usually a clear indication of insufficient server resources.

Note that both techniques require a COMPONENT PROXY to make the logical COMPONENT instances available when a request for them arrives.

❊ ❊ ❊

EJB uses PASSIVATION for Stateful Session Beans. Some LIFECYCLE CALLBACK operations, such as *ejbActivate()* and *ejbPassivate(),* are required to make this possible. These are used by the CONTAINER to notify the Bean that it has just been activated, or is about to be passivated, respectively. PASSIVATION is not necessary for Stateless Session Beans and EJB 2.0 Message-Driven Beans, because they do not have any state. A subsequent invocation can use a new or pooled, and therefore different, instance. State does not need to be maintained between two invocations. These types of Beans use INSTANCE POOLING.

Passivation is also not an issue in Entity Beans, because Entity Beans write their state to persistent storage. INSTANCE POOLING is used to minimize resource usage.

CCM uses the same techniques to implement PASSIVATION. The necessary functionality is already provided by the POA – see page 98 – which allows several activation and deactivation policies to be defined.

COM+ provides a feature called just-in-time (JIT) activation, which means that COMPONENTS are instantiated only when requests arrive for an instance. When the request has finished, or more correctly at the end of the current transaction, the COMPONENT instance is removed from memory.

Component Proxy

You want to implement VIRTUAL INSTANCES, for example using INSTANCE POOLING or PASSIVATION as the implementation strategy.

❊ ❊ ❊

When providing VIRTUAL INSTANCES, your clients need to hold references to logical COMPONENT instances that are not physically in memory. If the client invokes an operation on such an instance, the CONTAINER must have a way of obtaining a physical instance for the logical one. Also, before a request is forwarded to this physical instance, the CONTAINER needs to perform additional operations, such as invoking LIFECYCLE CALLBACK operations.

The problem described applies to VIRTUAL INSTANCES in general, no matter whether they are implemented using INSTANCE POOLING or PASSIVATION. In both cases the client must hold a reference to something to which it can send its requests. Only the strategies that the server uses to obtain a physical instance differ between INSTANCE POOLING and PASSIVATION.

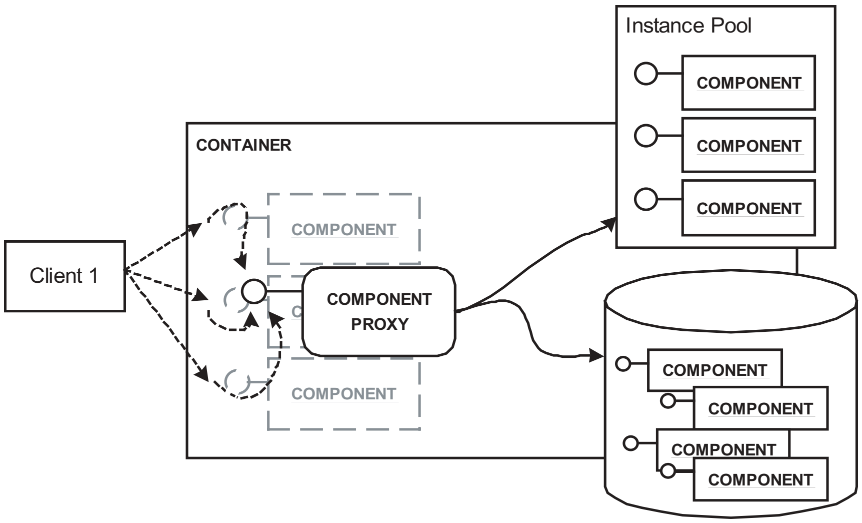
In addition to just obtaining a suitable physical instance, the instance must also be prepared. This preparation happens in two stages:

- First, the instance must be prepared to fulfill its task, for example by loading the respective state or recovering resources
- Second, the specifications given in the ANNOTATIONS have to be realized.

  Consider the situation in which a client calls an operation on a COMPONENT for which the ANNOTATIONS specify that a transaction is required. The CONTAINER must have a chance to start the transaction before the operation reaches the COMPONENT instance, and commit it or roll it back afterwards. Security has similar requirements.

Therefore:

Provide a proxy for the physical COMPONENT instances. Clients never have a reference to the physical component instance, they only talk to its proxy. The COMPONENT PROXY is responsible for implementing INSTANCE POOLING or PASSIVATION. It takes care of the enforcement of the technical requirements specified in the ANNOTATIONS. To increase performance, one proxy is usually responsible for several logical COMPONENT instances.



✲ ✲ ✲

The main goal of the pattern is to give the clients something on which they can invoke operations that is *not* the actual instance, because the actual instance may not be in memory. The COMPONENT PROXY must therefore at least implement logically the COMPONENT INTERFACE of the COMPONENT instances it represents.

To benefit from this pattern, the COMPONENT PROXY must use significantly fewer resources that the COMPONENT itself – otherwise there would be no resource savings. Thus usually one COMPONENT PROXY instance represents a set of (usually all) instances of a specific COMPONENT. To make this possible, a method invocation from a client must include the logical COMPONENT instance identifier, so that the proxy can do the forwarding correctly. This identifier must be transported by the COMPONENT BUS, and is usually part of the INVOCATION CONTEXT and supplied by the CLIENT LIBRARY. It is the responsibility of the CONTAINER to determine and use a suitable number of proxies.

It is also possible for a COMPONENT PROXY to represent instances of different COMPONENTS, but because it must implement the COMPONENT INTERFACES of all such COMPONENTS, this is not trivial.

The proxy does not implement any functional logic. It merely forwards the invocations to the actual COMPONENT IMPLEMENTATION provided by the developer. The proxy itself has a rather simple behavior pattern:

- Enforce ANNOTATIONS – check security, start a transaction if necessary
- Obtain a physical instance using INSTANCE POOLING or PASSIVATION
- Forward request to COMPONENT IMPLEMENTATION
- Finish transaction, if necessary

It can therefore easily be generated. It is usually part of the GLUE CODE LAYER, which acts as an adapter between the generic CONTAINER and the specific COMPONENT.

❊ ❊ ❊

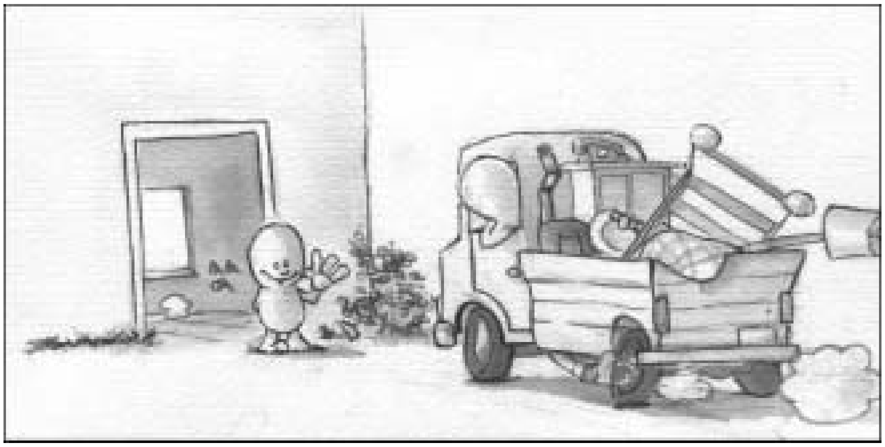
In EJB, the COMPONENT PROXY is attached to the COMPONENT BUS, which forwards invocations from the client application to the APPLICATION SERVER. The actual type of the COMPONENT PROXY is an implementation detail of the CONTAINER and thus not standardized as part of EJB. It forwards the invocations to the COMPONENT IMPLEMENTATION object provided by the Bean programmer. The proxy is generated during COMPONENT INSTALLATION.

The concept is also implemented by CCM, although concrete implementations are not yet available. Generated servants delegate operations to the COMPONENT IMPLEMENTATION.

COM+ uses the same technique. The server-side stub, a part of the marshalling DLL, plays this role.

Glue-Code Layer

You are using a CONTAINER to provide technical concerns to COMPONENTS. You want to reuse your CONTAINER for many different COMPONENTS.



❋ ❋ ❋

To reuse the CONTAINER so that it can support the technical concerns of many different COMPONENTS, the CONTAINER's functionality must be generic. However, each COMPONENT has a unique COMPONENT INTERFACE and features a specific configuration for the technical concerns in the ANNOTATIONS. Usually, it is not possible to host specific COMPONENTS in a completely generic CONTAINER, especially if you want to do this with reasonable performance.

The CONTAINER's job is to *integrate* the functional concerns provided by the COMPONENT with the required technical concerns. To achieve this integration as completely as possible from a client's point of view, the CONTAINER must take COMPONENT-specific aspects into account:

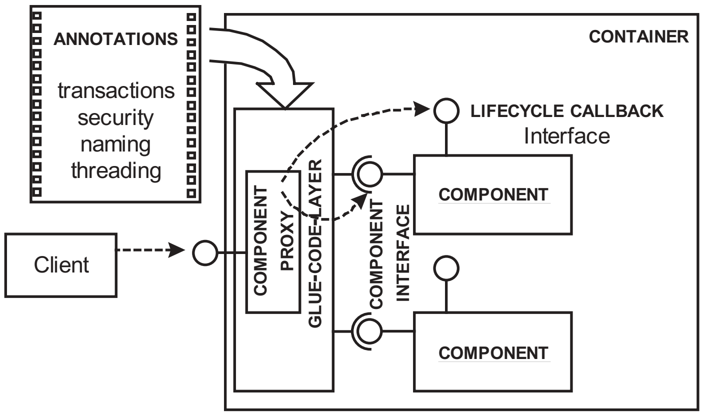
- Each COMPONENT has a unique COMPONENT INTERFACE. This interface must be 'wrapped' by something that has the same interface to ensure that it can be accessed by the client.

- The CONTAINER needs to provide VIRTUAL INSTANCES in order to minimize resource consumption. A PASSIVATION or INSTANCE POOLING strategy has to be implemented. The LIFECYCLE CALLBACK operations have to be called at appropriate times.
- The CONTAINER has to integrate all the specifications made in the ANNOTATIONS with the COMPONENT INTERFACE.

Most but not all of this work is done by the COMPONENT PROXY. The proxy, however, should impose as little overhead as possible – performance is definitely an issue here, because every method invocation 'passes' this proxy.

Therefore:

Use a layer of generated code as an adapter between the generic parts of the CONTAINER and the specific COMPONENT IMPLEMENTATION. This layer adapts the generic CONTAINER to a specific COMPONENT. Generate a specific GLUE-CODE LAYER for each COMPONENT running in the CONTAINER. In particular, the COMPONENT PROXY is part of the GLUE-CODE LAYER.



❋ ❋ ❋

The need for a GLUE-CODE LAYER depends heavily on the implementation language in use, and in particular on its type system. A lack of generic types, such as in Java, usually requires code generation as a consequence.

There are of course ways around this problem, especially in interpreted or semi-interpreted languages. For example, when Java is used as the implementation language, reflection can be used. Reflection allows developers to discover and invoke the operations a class or an interface provides dynamically.

The need to generate code could be minimized or even removed by using the reflective features of the underlying language, together with run-time interpretation of the ANNOTATIONS. However, this conflicts with performance requirements. Reflection and interpretation are always much slower than custom-generated and compiled code, which is why they are often not viable options.

The CONTAINER needs an opportunity to generate the GLUE-CODE LAYER. COMPONENT INSTALLATION is used to allow the CONTAINER to do everything that is necessary to host a specific COMPONENT. This is also the time at which the GLUE-CODE LAYER is generated.

Code generation in general is often regarded as undesirable, because the generated code is hard for developers to understand and hard to maintain. In this case, however, as with CORBA stubs, this generated code never has to be modified and almost never has to be read. From the client *and* the COMPONENT developer's viewpoints the appearance of the generated code is irrelevant, as long as it corresponds to the semantics defined by the component architecture. In the case of a standardized COMPONENT architecture, this generated code and its characteristics provide the means by which a vendor's CONTAINER can be adapted to the interfaces of the COMPONENTS as standardized in CCM, EJB and COM+.

The combination of CONTAINER and GLUE-CODE LAYER can share their work in different ways:

- The CONTAINER may already provide a lot of functionality and the GLUE-CODE-LAYER act merely as an adapter.
- The CONTAINER may basically be just a collection of hooks to which the GLUE-CODE is added, and the CONTAINER can be generated completely. This code generation is then done by the APPLICATION SERVER during COMPONENT INSTALLATION.

If you want to make the CONTAINER/GLUE-CODE LAYER flexible at run-time, the CONTAINER provider could offer INTERCEPTION. INTERCEPTION allows you to integrate custom behavior into the invocation chain of an operation in the CONTAINER.

❊ ❊ ❊

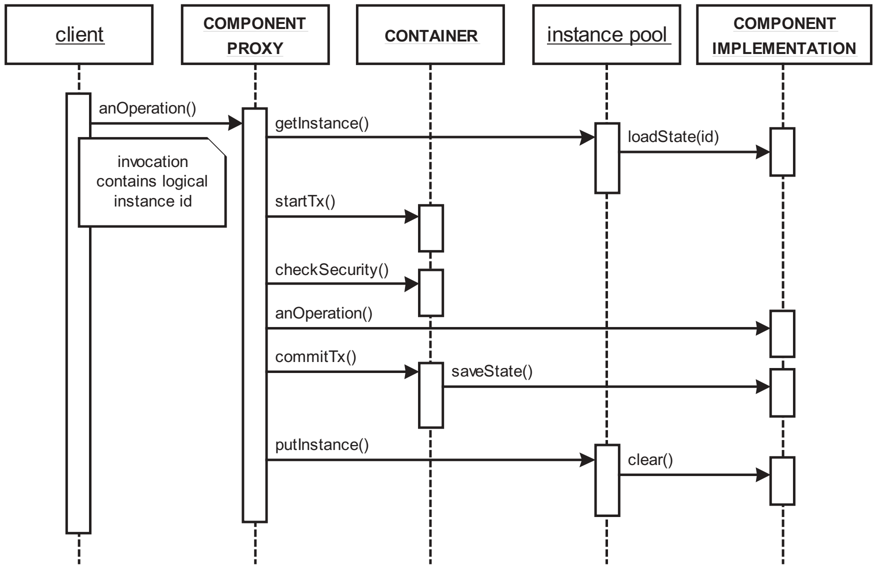
In EJB, the APPLICATION SERVER usually creates a set of classes that often begin with an underscore, end with *Impl* or have an other generated names. These classes usually contain the GLUE CODE LAYER, used to adapt to specific protocols used by the COMPONENT BUS and to manage transactions, security, pooling, et cetera.

In CCM, code generation is used in the same way as in EJB. While in EJB, Java's reflection could be used to overcome some of the requirements for code generation, this is not true for some CCM implementations for other languages that do not offer reflective features. Automatically-created DLLs or other kinds of libraries are used here.

In COM+ the GLUE CODE LAYER is also used. For example, the marshaling DLL is created as part of the GLUE CODE LAYER, and contains the server-side and the client-side proxies.

Summary

This chapter provides some details of the implementation of the CONTAINER. Central to this implementation is the concept of VIRTUAL INSTANCES, which is necessary to keep resource usage at an acceptable level. The COMPONENT PROXY serves as a placeholder for physical COMPONENT instances, giving the client the illusion of interfacing directly with the desired logical instance. To adapt the generic implementations of technical concerns provided by the CONTAINER to specific COMPONENTS and their COMPONENT INTERFACES, a GLUE-CODE LAYER is generated by the CONTAINER.



With all these structures in place, the CONTAINER can implement different strategies to minimize resource usage, mainly by keeping only those parts – COMPONENT instances – of the application in memory that are really needed:

- INSTANCE POOLING uses a set of pre-instantiated COMPONENT instances and changes their logical identity by storing and loading the respective state to/from a persistent store. This is especially suitable for ENTITY COMPONENTS and SERVICE COMPONENTS.
- PASSIVATION really removes temporarily-unused instances from memory and stores their state on some kind of persistent storage. This is especially useful for SESSION COMPONENTS.

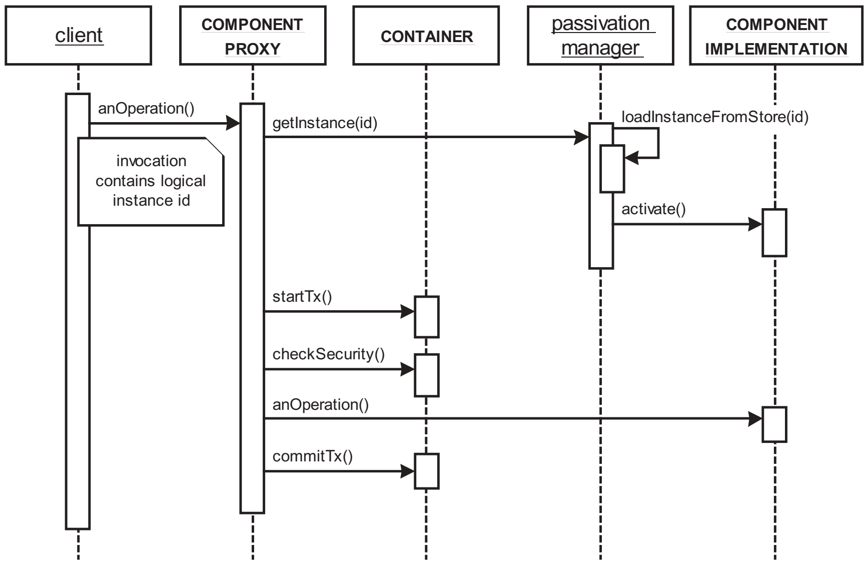
The following sequence diagram shows the collaborations among the different parts of the CONTAINER and the COMPONENTS. First, we look at the case of INSTANCE POOLING.

A client invokes an operation on a COMPONENT. The invocation reaches the COMPONENT PROXY of the respective COMPONENT. The proxy contacts the instance pool and requests a currently-unused physical instance. The pool finds a suitable COMPONENT IMPLEMENTATION instance and calls a LIFECYCLE CALLBACK operation to load the state that represents the required logical instance. The COMPONENT PROXY, together with the CONTAINER, then takes care of the technical concerns (*startTransaction()* and *checkSecurity()*). Finally the operation originally issued by the client is forwarded to the previously-prepared COMPONENT IMPLEMENTATION instance.

After the execution of the operation, the COMPONENT PROXY once again handles technical concerns (here *commitTransaction()*) and the CONTAINER calls a LIFECYCLE CALLBACK operation on the instance, which saves the state of the instance back to persistent storage. The instance is then put back into the pool and the state of the instance is cleared. The instance is now ready to serve another request.

In case of PASSIVATION, things are different. There are two more or less unrelated phases, the passivation phase and the reactivation of an instance.

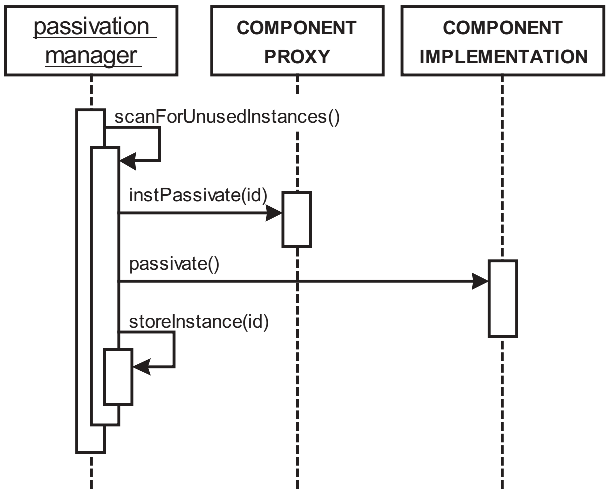
Starting with the latter, a client invokes an operation on a passivated instance:



The invocation reaches the COMPONENT PROXY. It calls the *Passivation-Manager*[1] to activate the instance on which the operation should be invoked. Once it is reloaded, its *activate()* LIFECYCLE CALLBACK operation is called. Then, just as in the case of INSTANCE POOLING, the COMPONENT PROXY handles the technical concerns and forwards the invocation to the reactivated instance.

Note that in the case in which an invocation is intended for an instance that has not yet been passivated, the *PassivationManager* simply returns the already available instance.

Later, if no invocations have arrived for a predefined period, the instance is passivated to a persistent store. This happens without any invocation from a client – it is a CONTAINER policy to determine when to passivate a physical instance:

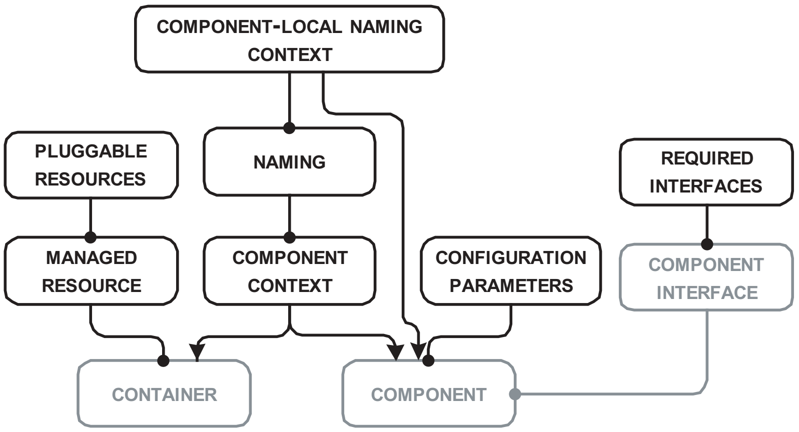


Whenever the *PassivationManager* finds an instance that has not been used for a specified period, it notifies the COMPONENT PROXY that it is about to passivate the respective instance. It then calls the *passivate()* LIFECYCLE CALLBACK operation to allow the instance to prepare for passivation. In a final step, the *PassivationManager* stores the instance in a persistent store.

1. The *PassivationManager* is the part of the CONTAINER that manages the PASSIVATION of SESSION COMPONENT instances.

4 A Component and its Environment

Although a COMPONENT's task is to implement specific functionality completely, independent of other COMPONENTS, a COMPONENT cannot fulfill this task without access to some parts of its environment.



The basis for access to a COMPONENT's environment is the COMPONENT CONTEXT. This allows the COMPONENT to control some aspects of its CONTAINER and to access external resources. Resources cannot be managed by the COMPONENT itself, for reasons of portability and efficiency, which is why the CONTAINER or the APPLICATION SERVER provide MANAGED RESOURCES to the COMPONENTS it hosts.

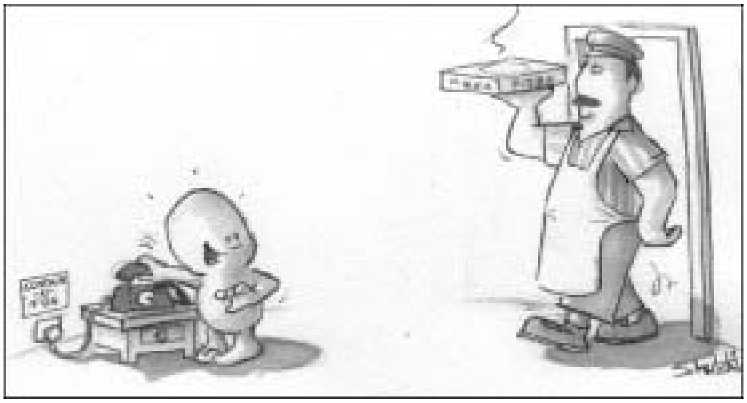
A more sophisticated replacement for MANAGED RESOURCES is PLUGGABLE RESOURCES. These allow third-party vendors to integrate any kind of resource seamlessly with the CONTAINER or APPLICATION SERVER, including technical concerns such as security or pooling.

A global authority that registers all available COMPONENTS and resources must be available to 'glue' the system together. A NAMING system is usually used for this. To make sure that a portable COMPONENT does not depend on specific names in the NAMING system, you can use a COMPONENT-LOCAL NAMING CONTEXT. Entries in this context are mapped to the real NAMING system when the COMPONENT is installed in a CONTAINER.

To give COMPONENT programmers a means of implementing the principle of variability, it must be possible to pass CONFIGURATION PARAMETERS to COMPONENTS when they are deployed. To track and manage the dependencies of a specific COMPONENT on other COMPONENT INTERFACES, REQUIRED INTERFACES can be specified as part of COMPONENT ANNOTATIONS.

Component Context

Your application uses COMPONENTS that run in a CONTAINER.



❊ ❊ ❊

Your COMPONENTS cannot exist completely on their own – they have to be given access to external resources. Moreover, although they are not responsible for the implementation of technical concerns, they might need to control *some* aspects of them at run time. All this has to be done without compromising the integrity of the CONTAINER.

Imagine an ENTITY COMPONENT that stores its state in a database. To do this, the COMPONENT needs a connection to the database. The database will of course not be part of the COMPONENT, thus the COMPONENT needs to access the database connection as an external resource. Note that such resources are usually controlled by the CONTAINER, which provides them in the form of MANAGED RESOURCES. Other external resources are for example the COMPONENT INTERFACES that the COMPONENT needs to use in order to perform its own job.

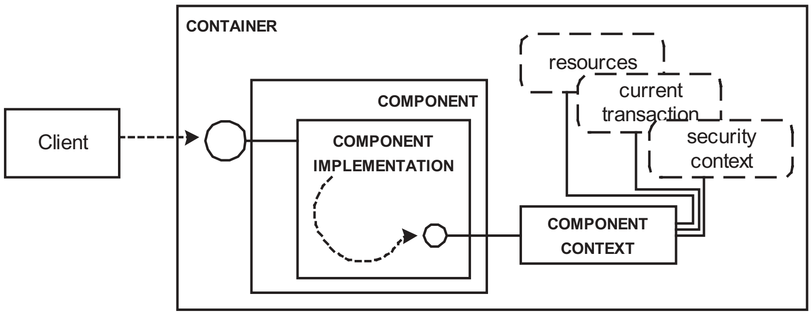
COMPONENTS sometimes need to control some technical concerns handled by the CONTAINER. The most common example is transactions. Of course, it is the CONTAINER's job to coordinate transactions that involve several COMPONENTS and integrate them with potentially many transactional resources. But a COMPONENT that is involved in a transaction must be able to mark the transaction for rollback if it decides this is necessary for reasons of the business logic. For example, an account might decide that the current transaction must be rolled back because too much money is to be withdrawn from it.

Access to security information might also be necessary. This is the case if the COMPONENT needs to implement more sophisticated security policies that cannot be specified declaratively in the ANNOTATIONS.

There is an additional issue that arises when the CONTAINER uses resource optimization techniques such as POOLING in the case of Entity Beans. Because an instance's logical identity changes during its lifecycle, the instance might not always know what identity it currently has. The CONTAINER thus has to have a way to tell the instance about its identity.

Therefore:

Let the CONTAINER provide a COMPONENT CONTEXT to each COMPONENT instance. This context object's interface provides operations for accessing resources, security information, the current transaction or instance-specific information about the COMPONENT instance itself. It can also include the possibility of accessing other parts of the COMPONENT's environment, such as NAMING or the COMPONENT-LOCAL NAMING CONTEXT.



❋ ❋ ❋

Using this pattern allows COMPONENTS instances to access selected aspects of their environment, the CONTAINER. The COMPONENT CONTEXT interface is well defined as part of the component architecture. It serves as a contract between the COMPONENT and the CONTAINER, just as the LIFECYCLE CALLBACK interface is a contract between the CONTAINER and the COMPONENT. The CONTAINER is of course free to implement the COMPONENT CONTEXT interface in any way it likes that is compatible with its internal structure.

Usually, the context object is passed to the instance at the very beginning of its lifecycle. It remains valid until the instance is destroyed. Note however that the COMPONENT CONTEXT object is available to the COMPONENT *and* to the CONTAINER. The CONTAINER is free to update or change any information in the context whenever it likes. In fact, this is absolutely necessary, because the current transaction or the called identifier of a method invocation change for every invocation made on the COMPONENT. The COMPONENT CONTEXT must reflect these changes. As a consequence, the COMPONENT must never store information obtained from the COMPONENT CONTEXT locally.

As mentioned above, the COMPONENT CONTEXT is used frequently to access invocation-specific data such as the security identity of the caller. Such data is transmitted to the COMPONENT with every method invocation as part of the INVOCATION CONTEXT. On the client side, the CLIENT LIBRARY provides 'magic' to add this data to each method invocation.

❊ ❊ ❊

In EJB, the *javax.ejb.EntityContext* and j*avax.ejb.SessionContext* interfaces are implementations of the COMPONENT CONTEXT pattern. A Bean has to implement LIFECYCLE CALLBACK operations, which the CONTAINER can call to provide the context object to the Bean instance (*setEntityContext()* and s*etSessionContext()*). This will be done at well-defined points during the lifecycle of a Bean. So the context is available to a COMPONENT whenever it is active and may need to access the context.

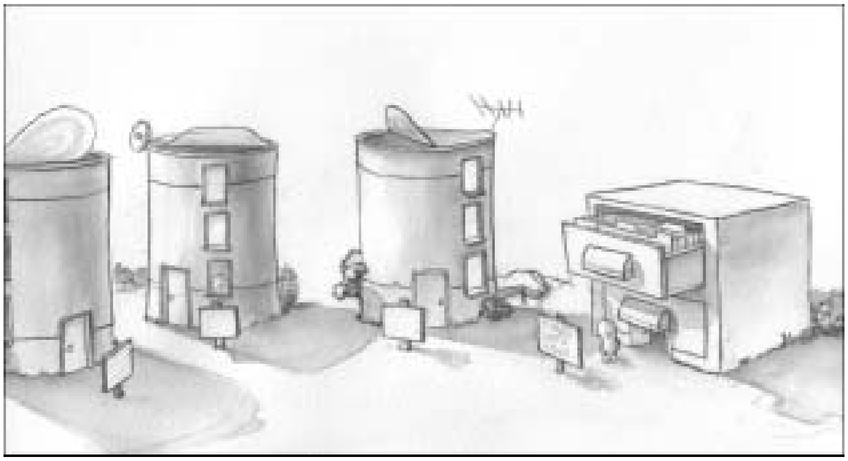
CCM defines a set of interfaces to implement the COMPONENT CONTEXT pattern. Some are called *internal* and define the communication between the COMPONENT and the CONTAINER. Examples are *Transaction*, *Security*, and *Storage*.

One interface, the *ComponentContext* interface, serves as the bootstrap interface to access the others. Different specialized contexts exist, depending on the type of COMPONENT. The CONTAINER passes the COMPONENT CONTEXT to a COMPONENT when it is instantiated. The mechanisms are basically the same as in EJB.

In COM+, a COMPONENT can obtain its context using the *CoGetObject-Context()*. From a COMPONENT's point of view, the context behaves just like another COMPONENT, and provides several interfaces. Among others, it provides *IContextState*, used to manage deactivation and transaction voting, and *ISecurityProperty* to determine the security identifier of the caller, for example a user name. Information that is specific to a particular invocation can be obtained using a different context, the *call context*, which is available via *CoGetCallContext()*. Using *ISecurtityCallContext*, the COMPONENT can access fine-grained security and caller/role information. *IServerSecurity* allows the COMPONENT to access DCOM security information.

Naming

Your COMPONENTS can access their environment via the COMPONENT CONTEXT. You want to assemble an application consisting of several COMPONENTS and other resources, typically hosted in several CONTAINERS in one APPLICATION SERVER.



❊ ❊ ❊

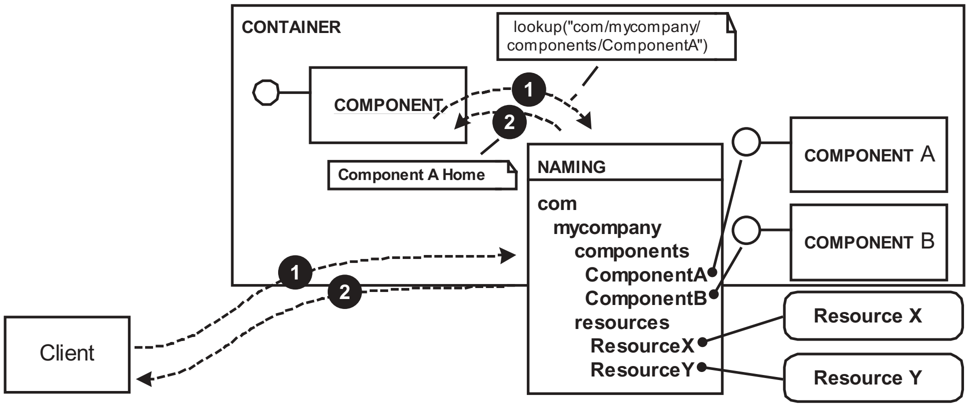
Accessing COMPONENTS from clients, or accessing other COMPONENTS or resources from within a COMPONENT IMPLEMENTATION, requires a well-defined means of obtaining references to the respective items. Location and distribution over CONTAINERS, APPLICATION SERVERS and machines should be transparent to the client.

Consider a COMPONENT IMPLEMENTATION. As part of its functionality, it needs to access another COMPONENT instance. The COMPONENT needs a way of obtaining a reference to the other instance, or more specifically, to its COMPONENT HOME. Because distribution is a technical concern, it should not have to involve the COMPONENT developer, who should not have to worry about the location of a COMPONENT. The developer therefore needs a *central* point from which references to other COMPONENTS can be obtained.

The same is true for resources such as database connections. Again, it is the CONTAINER's job to manage such resources by providing MANAGED RESOURCES – see below. But a COMPONENT IMPLEMENTATION needs a way to obtain a resource reference.

Therefore:

Provide a central NAMING service that maps structured names to resource references. NAMING can be used uniformly to look up any kind of COMPONENT or resource reference. It can be accessed by clients and by COMPONENT IMPLEMENTATIONS using commonly-known object references provided by the APPLICATION SERVER or the COMPONENT BUS.



1: A client calls lookup on the NAMING service, providing the name as a parameter
2: The NAMING service returns the reference

❊ ❊ ❊

NAMING provides a *white pages* service by mapping names to references. These references can either be resources such as database connections, message queues, or other COMPONENTS. Note that there is often another level of indirection – the NAMING service usually provides references to *Factories* [GHJV94].

- In the case of MANAGED RESOURCES, a resource factory is returned that in turn provides references to the resources themselves. However, the factory can use optimizations such as connection pooling.

- For COMPONENTS, the COMPONENT HOME is returned. Again, the COMPONENT HOME is a factory that allows clients to create new instances, locate old instances or delete instances.

The names used in the naming service are usually structured, just like file names in a file system. A name is normally represented as a string; a valid naming name for a COMPONENT HOME could therefore be */myCompany/mySystem/myComponentHome*, for example. Instead of directories, NAMING uses *contexts*. A *context* can contain reference bindings or other *contexts,* resulting in a hierarchy of contexts.

You should enforce a uniform naming scheme to make look-up for COMPONENT HOMES and MANAGED RESOURCES simple. For example, for each COMPONENT called *X*, the HOME could be called *XHome*. To avoid name clashes, you should also standardize a context structure, such as */myCompany/mySystem/…*

The NAMING service is usually provided by the APPLICATION SERVER, which also contains the CONTAINERS for the different COMPONENTS. Client access to the NAMING service is provided using the COMPONENT BUS. COMPONENT implementations access NAMING using the COMPONENT CONTEXT.

NAMING usually provides two sets of operations. One set is responsible for registering and unregistering objects. These operations are usually called *bind(name, objectRef)* or *unbind(name).* The other set is used to do the actual look-up. Such operations are usually called *lookup(name)*, and return the reference to the COMPONENT HOME or MANAGED RESOURCE. Listing the content of a context is usually also possible through an operation called *list(contextName)*.

NAMING services can usually be *federated*. This means that one NAMING context can be embedded in another, as in UNIX file systems, where you can mount different physical hard drives or partitions in one global directory tree. This is especially interesting for clustered APPLICATION SERVERS. It was originally intended to provide a common access to all NAMING systems in an enterprise.

If a suitable non-standard NAMING implementation is used, it can also provide a simple form of load balancing. This works by returning different references for a look-up if the NAMING service knows about several COMPONENTS that provide the same COMPONENT INTERFACE. Ideally for load-balancing they would be located on different machines.

There is a related service called *trader*. A trader also returns references, however the look-up is not based on a unique name but on a set of attributes. Attributes can include the reference type, performance data, location and more. A client can query for these attributes based on a search specification. Just as NAMING provides a white pages service, a *trader* thus provides a yellow pages service. This pattern is also known as 'Lookup'.

❊ ❊ ❊

As mentioned above, an EJB CONTAINER is usually part of a J2EE APPLICATION SERVER. The APPLICATION SERVER provides an implementation of NAMING accessible through JNDI, the Java Naming and Directory Interface that is also part of J2EE. Every COMPONENT HOME, every service, and every MANAGED RESOURCE has to be registered within the NAMING service. Each COMPONENT must therefore be given a name that will be used for the registration. This name is specified in the ANNOTATIONS. A COMPONENT instance (running in the CONTAINER) or any other client can then access the NAMING context and use it for look-up.

In CCM, CORBA Naming is used for this purpose. A client looks up the COMPONENT HOME of a COMPONENT or other services using normal CORBA Naming look-up operations. The reference to the NAMING service can be obtained using CORBA's built-in *resolve_initial_references()* operation, which is generally used to look up basic services under commonly-known names.

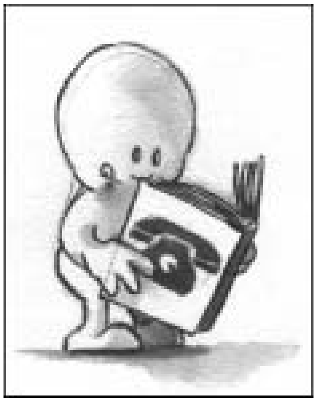
Another method exists for connecting with Home Interfaces: a *home finder* is a CORBA object that can be used to retrieve references to COMPONENT HOME objects, either by specifying the repository identifier of the COMPONENT or by passing the repository identifier of the factory.

In COM+ there is no real NAMING service, so you cannot define arbitrary names and arrange them in hierarchies of contexts. However, it is of course possible to locate the factory for a specific COMPONENT.

There is a COM+ library operation called *CoGetClassObject()* to which you pass the GUID (Globally Unique IDentifier) of the COMPONENT whose factory you want to get. The Service Control Manager, which is responsible for getting this object, uses the Windows Registry to find the server machine that hosts the object. This server machine then uses the same mechanism to find the server DLL that implements the COMPONENT, and calls a standard operation in the DLL that returns the class object. This object's reference is then returned.

Component-Local Naming Context

Your COMPONENTS access other COMPONENTS or resources. They use the COMPONENT CONTEXT and the NAMING system for this purpose.



❊ ❊ ❊

The NAMING system is a part of the environment in which a COMPONENT is used, usually part of the APPLICATION SERVER. By using names from the NAMING system in a COMPONENT IMPLEMENTATION, you make your COMPONENT dependent on these names. It is not possible for these names to be the same in all environments in which the COMPONENT should be used.

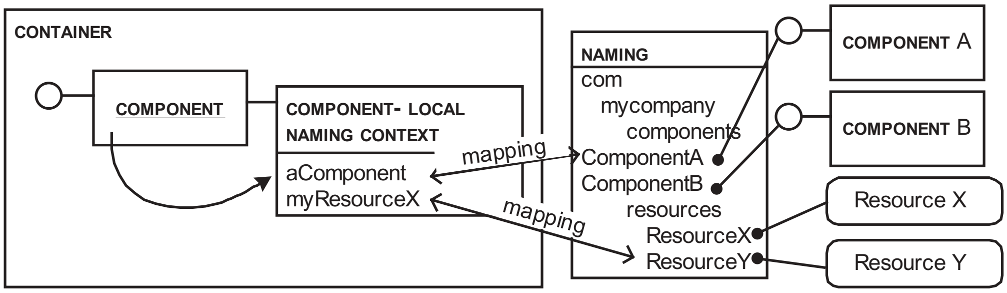
One of the goals of using COMPONENTS is to provide *reusable* blocks of functionality for use in different systems or applications. This means that a specific COMPONENT will be used in several different environments. Consider an ENTITY COMPONENT. To store its persistent state, it needs to look up a database connection in the NAMING service. The database must be looked up using a specific name. This name must be coded in the COMPONENT IMPLEMENTATION, making the COMPONENT dependent on the name. Of course, the same is true when another COMPONENT is looked up in NAMING.

Installing the COMPONENT in another environment requires either a change of the name of the resource in the code, or a renaming of the resource in the NAMING system. Changing code is not possible because it is usually not available. Changing the name in the NAMING system is very hard, because other COMPONENTS might reference the name that is currently used.

Making a COMPONENT dependent on the names in its environment limits its portability and is thus undesirable.

Therefore:

Provide a COMPONENT with its own local NAMING context. The COMPONENT IMPLEMENTATION accesses only this local name. When a COMPONENT is installed in a CONTAINER in a specific environment (APPLICATION SERVER), map the names in the local context to the names of the actual resources in the global NAMING system.



❊ ❊ ❊

Using the indirection introduced by this pattern, deploying the COMPONENT in another environment only requires a change to the mapping during COMPONENT INSTALLATION. No changes in code or changes in the global NAMING service are necessary.

However, changing the mapping is not always enough. For example, in the case of a database, the interface to a database, usually SQL, may contain vendor-specific extensions. The COMPONENT therefore needs to know the type of database to enable it to create the correct SQL code. To overcome this problem, use CONFIGURATION PARAMETERS to tell the COMPONENT which SQL code has to be used.

Note that in component architectures where a COMPONENT-LOCAL NAMING CONTEXT is not available, this behavior can easily be simulated with CONFIGURATION PARAMETERS. Such a parameter, which can also be changed at COMPONENT INSTALLATION, can contain a mapping from a local name to the global name in the NAMING context. Of course it is the COMPONENT IMPLEMENTATION's responsibility to do the mapping manually.

❊ ❊ ❊

EJB provides a COMPONENT-LOCAL NAMING CONTEXT. From a COMPONENT's point of view it is accessed in the same way as the global NAMING CONTEXT and can be found at *java:comp/env*. The entries in this context are further structured into other subcontexts, depending on the type of the entry. The mapping to the real names in NAMING is done during COMPONENT INSTALLATION. Actually, the COMPONENT-LOCAL NAMING CONTEXT is just another view of the CONFIGURATION PARAMETERS, as the entries in the local JNDI context are defined in the Bean's Deployment Descriptor.

CCM, being a superset of EJB, provides a similar mechanism.

COM+ does not provide this feature, therefore CONFIGURATION PARAMETERS must be used.

Managed Resource

You run your COMPONENTS in a CONTAINER. Using the COMPONENT CONTEXT and NAMING, the COMPONENT IMPLEMENTATION can access resources and COMPONENTS. The CONTAINER provides VIRTUAL INSTANCES for your COMPONENTS.

❊ ❊ ❊

Because the CONTAINER provides VIRTUAL INSTANCES, you do not know how many physical COMPONENT instances exist at any given time. Thus, you don't know how many resources you actually have to provide, and to whom. The CONTAINER must also be able to integrate the resources with its security and transaction policies.

A resource in the context of this pattern is an external entity that the COMPONENT accesses. Typical examples include database connections, message queues, and files. It is not very useful to let a COMPONENT manage these resources, for several reasons.

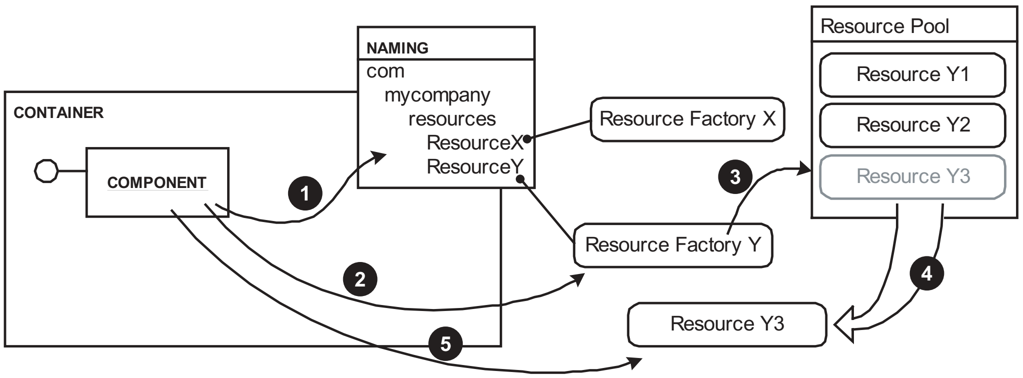
As a consequence of VIRTUAL INSTANCES, the number of physical COMPONENT instances varies. It is thus hard to determine the number of resources you will need over time. You cannot easily assign them to specific physical instances, because in the case of PASSIVATION, passive instances don't need the resources, and in the case of INSTANCE POOLING a pooled instance might not need the same resources as active instances. Repeated re-acquisition of resources during the lifecycle of an instance is also not an option, because of the performance penalty it incurs.

There are other problems with letting a COMPONENT instance manage its own resources. To access a resource, you usually need a driver, some kind of resource identifier, a user to log in with, or a password. If you install the COMPONENT in another CONTAINER in a different APPLICATION SERVER, there might be other types of resources available, they might have a different identifier or you might need to use another driver or user/password combination. You do not want to limit your COMPONENT's portability by hard-coding these things in your COMPONENT IMPLEMENTATION's source code.

There is an even more critical aspect. Resources must be integrated with the CONTAINER's handling of technical issues, especially transactions and security. If the COMPONENT just accesses arbitrary resources outside of the CONTAINER, the CONTAINER will not know about it, and therefore will not be able to integrate the resources seamlessly.

Therefore:

Let the CONTAINER manage resources for the COMPONENTS. This includes pooling of resource instances for faster acquisition by the COMPONENTS. All resources available to COMPONENTS have to be registered with the CONTAINER and published in the CONTAINER's NAMING system. COMPONENTS access resource factories through the COMPONENT-LOCAL NAMING CONTEXT.



Before: CONTAINER sets up a resource pool
1: Client looks up a resource factory in the NAMING service
2: Client accesses factory to request a resource
3: Resource factory accesses the ressource factory, and…
4: …a resource is taken from the pool, passed to the client
5: Client accesses the resource, integrated with the technical concerns such as security and transactions

❊ ❊ ❊

The management of resources by the CONTAINER usually includes the following:

- First, the CONTAINER creates a resource pool. It acquires a predefined number of instances of each resource and places them in a pool.
- When a COMPONENT instance needs a resource, it looks up a resource factory in NAMING and uses it to acquire a resource from the pool.

Getting a resource from a pool is much faster than actually acquiring the resource completely. The resources can also be shared among multiple COMPONENTS. This probably leads to less resource consumption. The pool definition also includes driver, user and password. When the COMPONENT accesses an instance, it does not need to worry about these.

Note that the programming style changes if resources are pooled. Without pooling, you would acquire a resource once at the beginning of the lifecycle of a COMPONENT instance and keep it until the end. This is because it takes finite time to acquire the resource. With pooling, however, you retrieve the resource from the pool only when you really need it and return it immediately afterwards. Because retrieving a resource from a pool takes almost no time, this is efficient. Note that this reduces the time for which a resource is locked by a client, thus reducing the total number of resources needed for the overall system.

Because all resources are registered in the CONTAINER, and because resources are accessed through a CONTAINER-provided resource factory, the CONTAINER can intercept any call to a resource and thereby manage transactions and security as specified in the ANNOTATIONS.

In most cases, several CONTAINERS run together in an APPLICATION SERVER. In this scenario the APPLICATION SERVER provides MANAGED RESOURCES for use by all CONTAINERS.

Resource pooling also has some disadvantages. Take a database connection, for example. A COMPONENT may not be able to specify with which user to log into a database, because all connections use the same user and therefore all connections in the pool are created equally. Alternatively, if the COMPONENT specifies user and password, then the pools are usually very small and the benefits of pooling are reduced.

❊ ❊ ❊

EJB APPLICATION SERVERS allow you to define database connection pools. They are configured using proprietary tools and syntax. However, from the COMPONENT's point of view they are all accessed using so-called *data sources*. A data source serves as a factory for the resources it provides. It is the clients' interface to resource pools. Pools can also be defined for other resources, especially those coming from PLUGGABLE RESOURCES defined using the J2EE Connection API. While not pooled, JMS queues or topics are also MANAGED RESOURCES and are accessed the same way.

All configured data sources are registered in the NAMING service, allowing the COMPONENTS to access them uniformly via JNDI. More specifically, they are accessed though the COMPONENT-LOCAL NAMING CONTEXT.

Persistence is usually handled by the Container in CCM. CCM provides quite a sophisticated persistence service definition – connection pooling is managed by the APPLICATION SERVER. However, because persistence support might be provided by a different vendor than the APPLICATION SERVER itself, an interface between the APPLICATION SERVER and the persistence provider is also part of the standard. A COMPONENT can access the persistence provider using the *Storage* interface. APPLICATION SERVERS are expected to manage pools of connections to persistence providers.

COM+ provides resource pooling through its OLEDB service. OLEDB is the successor to ODBC, a standard for accessing relational database systems. However, this is not specific to COM+, it is a general feature provided by OLEDB that can also be used from non-COM+ applications.

Pluggable Resources

You run your COMPONENTS in a CONTAINER and use MANAGED RESOURCES to provide pooling and technical integration of specific resources, such as database connections, message queues or files.

❊ ❊ ❊

COMPONENTS often need to access resources the CONTAINER or APPLICATION SERVER of which developers have incomplete knowledge, or are not concerned about, such as legacy systems or proprietary databases. These resources should be accessible in the same way as 'standard' MANAGED RESOURCES. You also want to integrate these resources in your CONTAINER's transaction and security management.

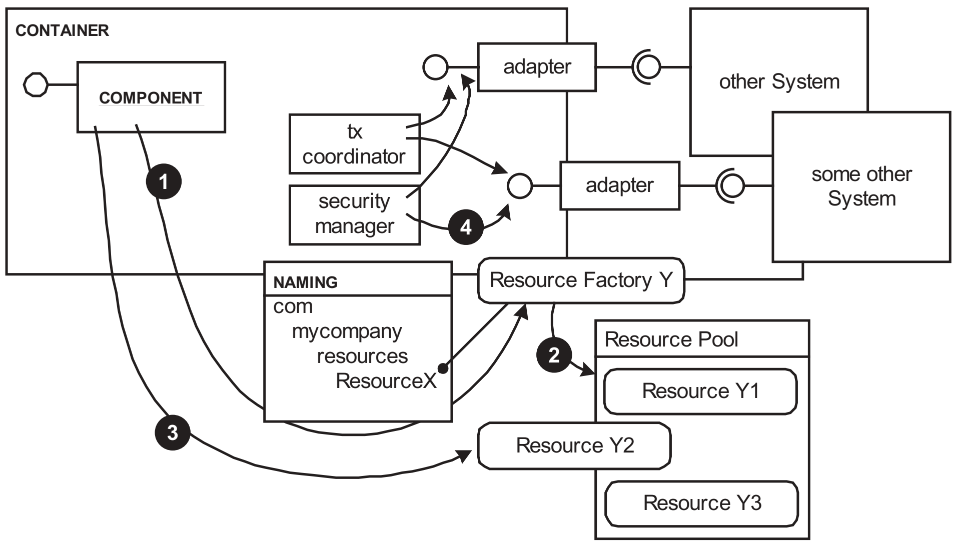
Consider the common case in which you want to integrate an Enterprise Information System into your system. You will usually use SERVICE COMPONENTS to provide a COMPONENT *Facade* around the system, so that the clients do not notice the 'foreign' system in the application and can use the current programming model.

However, real integration also includes technical concerns, in particular transactions and security. This means that the CONTAINER's transaction coordinator must be able to span transactions over the integrated system, for example to ensure synchronized roll-back, and that either the CONTAINER's security credentials must be accepted by the integrated system, or that some kind of mapping must be performed.

The issues described above all provide foreign system integration from the view of the COMPONENT user, the client. But, in addition to this, we also want to provide seamless integration for the COMPONENT programmer. He should be able to use foreign systems just as any other MANAGED RESOURCE – pooling should be available, and the resources should be usable in the same 'get-use-release' programming model.

Therefore:

Define resource adapters that allow any kind of resource to be plugged into the APPLICATION SERVER. Define a generic interface between the resource adapter and the APPLICATION SERVER that defines operations that allow the APPLICATION SERVER to build and manage pools of connections to the integrated system, and to integrate with the CONTAINER's technical concerns such as security and transaction coordination. As with MANAGED RESOURCES, these also provide a resource factory available through NAMING for use by COMPONENTS to obtain connections.



Before: CONTAINER sets up a resource pool
1: Client access resource factory and requests a resource
2: Factory accesses the pool and returns a resource
3: Client accesses the resource
4: The transaction and security manager in the CONTAINER use the resource adapters to coordinate the technical concerns with the plugged in resource

❊ ❊ ❊

Using this pattern has two significant advantages. Custom resources can be used efficiently, and they can be used based on the same programming model as built-in standard resources. 'Custom resources' can be anything in this context, including:

- Third-party Enterprise Information Systems (EIS)
- Proprietary databases
- Legacy systems
- Gateways to code in other languages

The idea is to create a suitable resource adapter only once, especially for mainstream third-party systems. Without pluggable resources, it becomes the application developer's responsibility to integrate technical concerns with the container, such as transaction handling and security of third party systems. This is a very complex and sometimes impossible task, as the developer has no access to the container's internal implementation.

❊ ❊ ❊

J2EE provides the Connector API in Version 1.3. This API implements this pattern, including pooling, transaction integration, et cetera. In addition to integrating these resources with transaction and security management of the CONTAINER, the connector architecture also provides a generic interface to them. It is based on command objects and result sets.

CCM features pluggable persistence providers to provide persistence for COMPONENTS, as specified in the COMPONENT definition, using Persistent State Definition Language (PSDL) definitions.

In COM+, the database connection pools and other pooled resources are based on the concept of *resource dispensers*, something that manages a pool of resources in close coordination with the transaction manager. There is a published API that lets you create your own resource dispensers, that is, you can create pools for your own resources.

Configuration Parameters

You use COMPONENTS to build reusable building blocks of functionality. To achieve reuse, you need a certain degree of functional variability (see page 22) in your COMPONENT IMPLEMENTATION.

❋ ❋ ❋

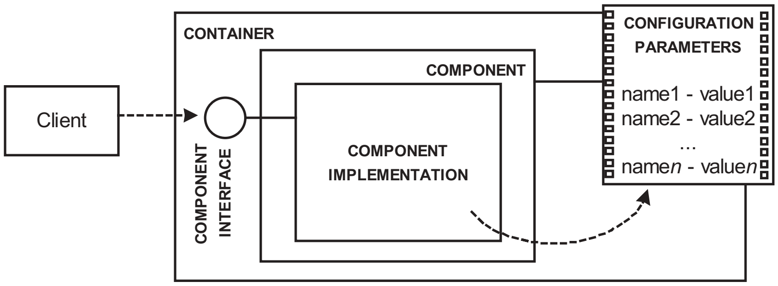
Functional variability must be available when the COMPONENT is used, which means during COMPONENT INSTALLATION. This variability must be achieved without modification to the COMPONENT IMPLEMENTATION code. You therefore need a means of passing configuration information into the COMPONENT during COMPONENT INSTALLATION, or later.

For an example of a functional configuration, consider a COMPONENT that manages addresses. One part of an address is the ZIP code. In each country the ZIP code has a different format. The format can be validated easily by using a regular expression. In Germany, for example, ZIP codes are five-digit numbers, which can be validated by an expression of the form '^99999'. To make the format of the ZIP code flexible, the regular expression must be configured during COMPONENT INSTALLATION.

Technical configuration is also necessary if a relevant issue is not covered by the ANNOTATIONS. For example, consider the initial size of a list. You don't want to hard-code this into your COMPONENT if the optimal size depends on the system in which the COMPONENT is used.

Therefore:

Allow the COMPONENT to access configuration parameters that are defined for the COMPONENT during COMPONENT INSTALLATION. Such parameters are usually name-value pairs, both of a string type. Make this information accessible from within the COMPONENT IMPLEMENTATION, for example by using the COMPONENT-LOCAL NAMING CONTEXT or the COMPONENT CONTEXT.



❊ ❊ ❊

This pattern allows the user – the installer or deployer – of a COMPONENT to change parts of the structure or behavior without changing its implementation. Of course, it is only possible to configure the aspects of the COMPONENT that the developer has made accessible through a CONFIGURATION PARAMETER.

Usually, the COMPONENT developer has to provide the names and suitable default values for the CONFIGURATION PARAMETERS, as well as some documentation about the function of each parameter. CONFIGURATION PARAMETERS are therefore often part of the ANNOTATIONS of a COMPONENT, and the values can (or must) be adapted during COMPONENT INSTALLATION, the process by which the COMPONENT is installed in a CONTAINER.

It is, of course, usually not possible to force the installer to provide correct values for such parameters. One way to help solve this problem is to provide useful default values as part of the ANNOTATIONS. These default values are set by the COMPONENT developer, who usually knows which values are sensible. LIFECYCLE CALLBACK operations can also be used. For example, the operation invoked at COMPONENT instantiation can verify the configuration and can signal an error in the case of problems.

❊ ❊ ❊

In EJB, CONFIGURATION PARAMETERS can be defined in the Deployment Descriptor (see ANNOTATIONS). They are essential name-value pairs. While the name is always of type *String*, the value can be of any primitive Java types including *String*. The Bean can access them using NAMING through JNDI. The parameters are part of the COMPONENT-LOCAL NAMING CONTEXT, which can be found at *java:comp/env/*.

CCM is a little more sophisticated – it supports the concept of COMPONENT configuration explicitly. The lifecycle of a COMPONENT instance has two parts: the configuration phase and the operational phase. Once the configuration phase has completed, signaled to the instance by calling *configuration_complete()*, the configuration cannot be changed.

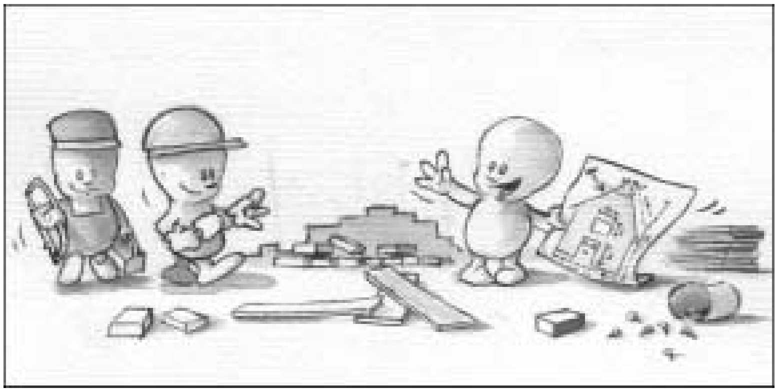
The configuration itself is expressed as a set of values for the COMPONENT's attributes (those defined directly on the COMPONENT, not on *facets*, which are additional interfaces that can be requested from a component). Thus the configuration is executed as a series of calls to attribute setters. The COMPONENT can raise an exception, if it considers the configuration invalid. The configuration is done by a *Component Configurator*, an interface that provides an operation *configure(Component)*. Specific configurators can be associated with creation operations (factories) in the COMPONENT HOME.

COM+ also provides a standard way to pass configuration information to a newly-created COMPONENT instance based on the use of the *IObjectConstruct* interface. The deployer can define a configuration string to be passed to the new instance, usually in the form:

name=value;name=value;… (*et cetera*)

Required Interfaces

You are using COMPONENTS to separate your business logic into separate, reusable entities. Now you want to reassemble applications from these COMPONENTS.



❊ ❊ ❊

To make COMPONENT applications manageable, you must know which COMPONENTS depend on other COMPONENT INTERFACES. This is usually not expressed by the COMPONENT INTERFACES alone, because they only specify what a COMPONENT provides, not what it requires in terms of other COMPONENTS.

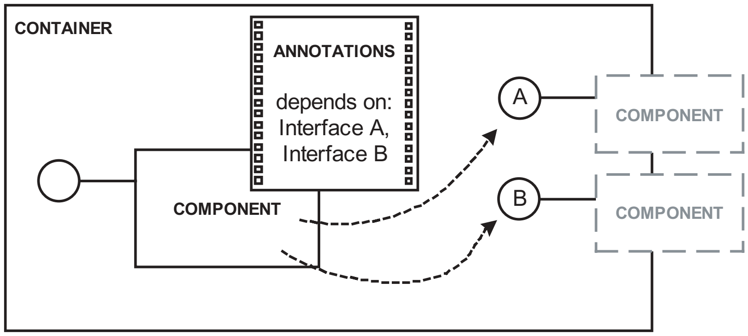
In general, a COMPONENT should provide a complete functionality. In most cases, of course, it cannot work without collaboration with other COMPONENTS – the summary section of Chapter 1 shows a typical layering scheme. While a COMPONENT must not depend on specific COMPONENT IMPLEMENTATIONS, it might very well expect to collaborate with other COMPONENTS that are accessible via a specific COMPONENT INTERFACE. Note that this is not a drawback – it is the whole purpose of a COMPONENT to provide a specific functionality and collaborate with other COMPONENTS that provide different, though perhaps related, functionality.

This does not mean we recommend that COMPONENTS should reference other COMPONENTS in an application, of course. Good software development practices still apply, and patterns such as *Facade* or *Mediator* can and should be used to reduce the number of dependencies.

However, if you have such dependencies on the interface of other COMPONENTS in the context of a larger application, you will want to make them explicit.

Therefore:

Allow a COMPONENT to specify which other COMPONENT INTERFACES it uses to perform its task. Either include it in the COMPONENT INTERFACE or add it to the ANNOTATIONS. Note that such dependencies *can* depend semantically on a COMPONENT's IMPLEMENTATION, not just on its abstract specification.



❊ ❊ ❊

Using this pattern allows you to check whether a deployed application that consists of a set of collaborating COMPONENTS can work without error. To avoid run-time errors, this is usually determined during COMPONENT INSTALLATION. Note, however, that this requires facilities not provided by programming languages. If you use a separate interface definition language for the COMPONENTS, you can make REQUIRED INTERFACES part of it. Otherwise, the ANNOTATIONS can contain the relevant information.

Note also that there is a semantic difference between including the REQUIRED INTERFACES specification in the COMPONENT INTERFACE and including it in the ANNOTATIONS:

- Including it in the ANNOTATIONS means that a particular COMPONENT requires another INTERFACE in its implementation. This is because ANNOTATIONS are specific to a COMPONENT's specific implementation, not to all COMPONENT implementations.
- Including the specification in the COMPONENT INTERFACE means that all COMPONENTS that implement the interface depend on another specific COMPONENT INTERFACE. This can be interpreted as a prescription that says "all COMPONENTS that implement this interface *have to use* the referenced one".

❊ ❊ ❊

In EJB, the Deployment Descriptor allows a Bean to specify which other Beans it requires. The COMPONENT-LOCAL NAMING CONTEXT, the instance-specific part of JNDI, is used to store the name of the required Beans and map them to the global name in JNDI. Note that EJB 2.0's Entity Bean relationships use a similar mechanism, but are not an implementation of this pattern – the dependencies are primarily related to persistence mapping issues.

CCM uses the concept of *connections*. A COMPONENT is able to connect to another COMPONENT if it provides a receptacle for the other interface as part of its own COMPONENT definition. A receptacle is thus a specification of a REQUIRED INTERFACE for a COMPONENT. The CIDL compiler creates a set of operations to connect and disconnect COMPONENTS.

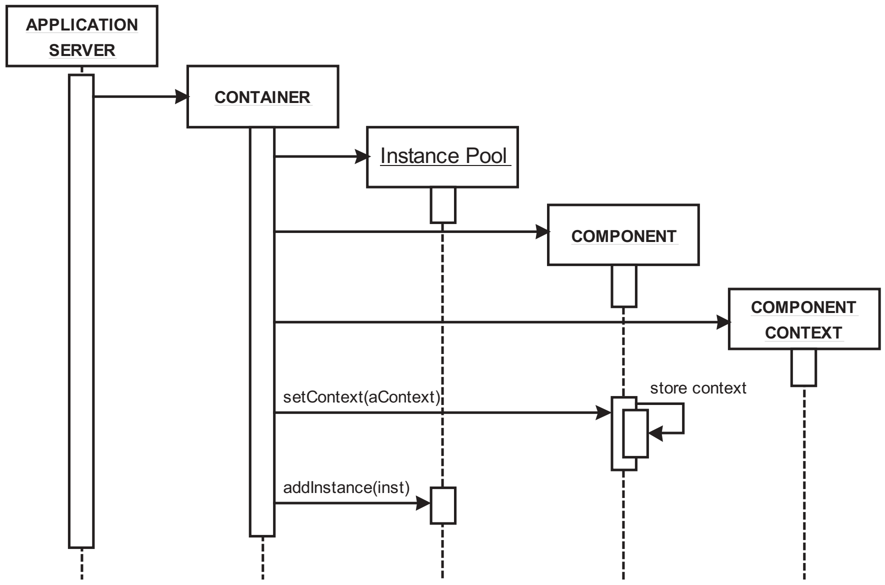
This pattern is not implemented in COM+.

Summary

This chapter shows how a COMPONENT communicates with its environment, usually the CONTAINER. The initial access point is usually the COMPONENT CONTEXT, which also allows the instance to control some aspects of the CONTAINER. It is provided by the CONTAINER to the COMPONENT when it is instantiated. The COMPONENT-LOCAL

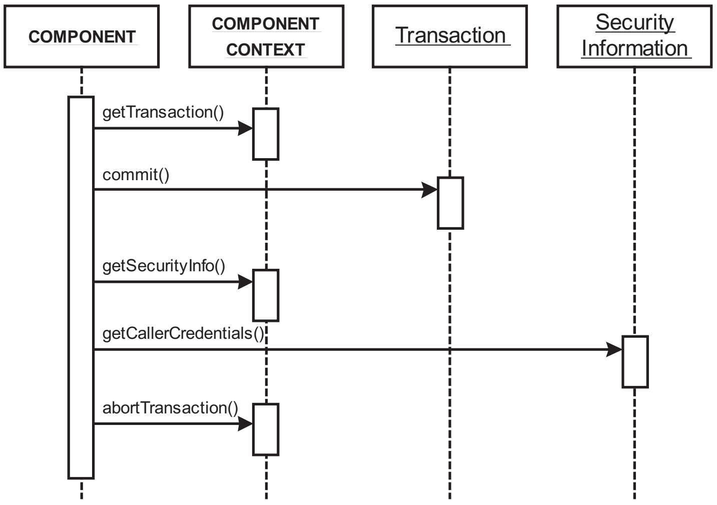
NAMING CONTEXT allows a COMPONENT to access references in NAMING based on a name scoped only to the COMPONENTS. MANAGED RESOURCES provide pooling of resources and their integration into the CONTAINER's management of technical concerns.

The following sequence diagram shows how the CONTAINER passes the COMPONENT CONTEXT to the COMPONENT when it is created. We use an ENTITY COMPONENT in the example.



When the APPLICATION SERVER is started, it instantiates the CONTAINERS for the COMPONENTS it has to run. The CONTAINER, in turn, instantiates an instance pool, then instantiates a COMPONENT and a suitable COMPONENT CONTEXT. It provides the COMPONENT CONTEXT to the just-created COMPONENT instance, which stores it for later use. Last but not least, the CONTAINER adds the new COMPONENT instance to the instance pool. The process of creating a COMPONENT and a COMPONENT CONTEXT, as well as the addition of the instance to the pool, is repeated until the predefined number of instances is available in the instance pool.

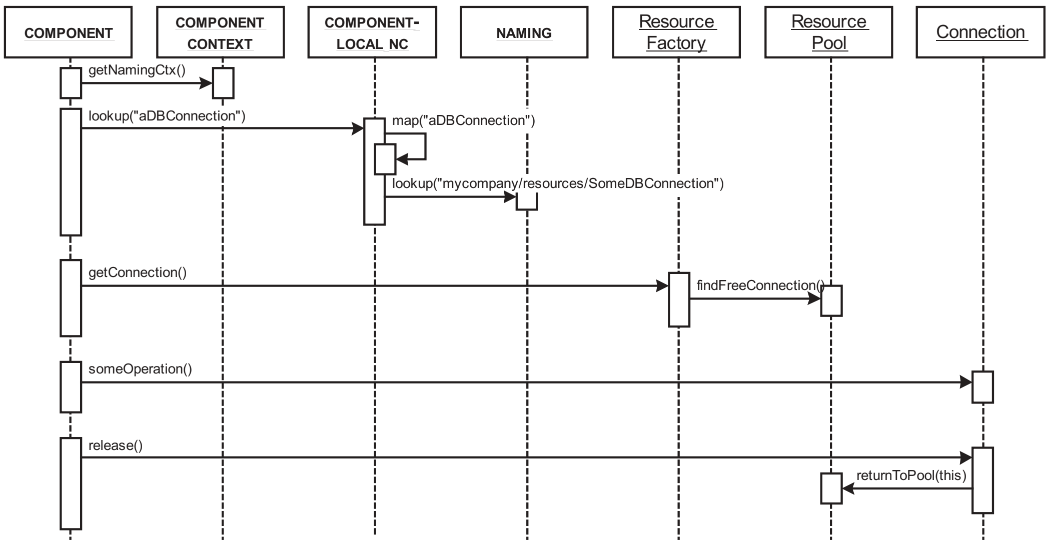
If the COMPONENT needs to influence technical concerns directly, it can call the respective operations on the COMPONENT CONTEXT. Examples include committing or aborting a transaction or obtaining security information about the caller or the current operation:



This sequence diagram contains three different things:

- Firstly, the COMPONENT uses the COMPONENT CONTEXT to look up the *Transaction* object associated with the current operation. It then calls *commit()* on this object.
- Secondly, it retrieves the *SecurityInfo* object to get information about the credentials of the caller of the current operation.
- As illustrated in the third case, the COMPONENT CONTEXT might also contain short-cut operations for the most typical requests. Here, the COMPONENT calls *abortTransaction()*, which is basically a shorthand for *getTransaction().abort()*.

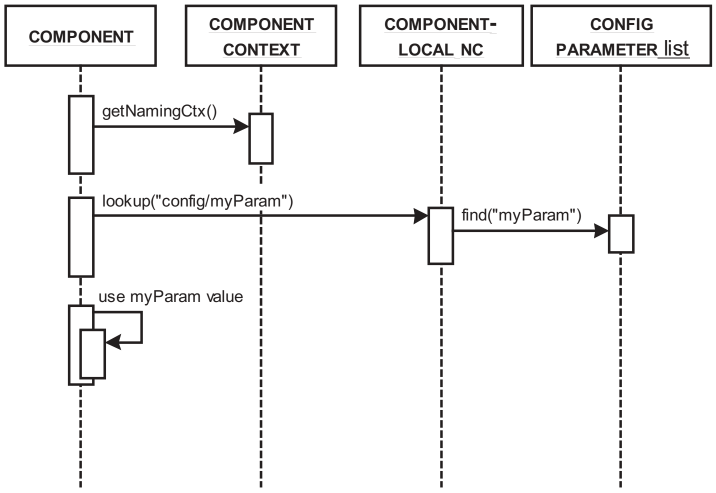
When a COMPONENT instance wants to use an external resource such as a database connection, it uses the COMPONENT CONTEXT to look up the COMPONENT-LOCAL NAMING CONTEXT and get the resource factory, which in turn creates the resource:



- Firstly, the instance uses the COMPONENT CONTEXT to access the COMPONENT-LOCAL NAMING CONTEXT. On this context it calls *lookup()*, supplying the logical name of the required resource.
- The COMPONENT-LOCAL NAMING CONTEXT uses information provided in the ANNOTATIONS to map the logical name to the actual name of the resource in the global NAMING context.
- The instance then looks up the resource in the global NAMING context, using the respective mapped name.

As mentioned above, the look-up actually returns a resource factory for the required resource. The COMPONENT then uses this factory to get a resource. Usually, the resources are pooled, and the resource factory accesses the pool to return an actual resource. The COMPONENT uses the resource, and, when finished, releases the resource. This results in the return of the resource to the pool.

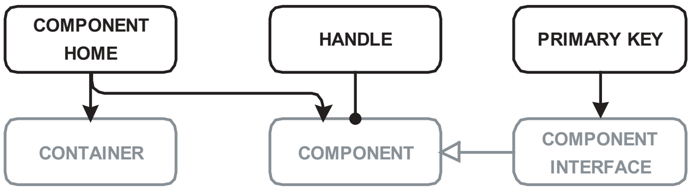
Access to CONFIGURATION PARAMETERS is usually also done through the COMPONENT-LOCAL NAMING CONTEXT. The next sequence diagram shows how this typically works:



The COMPONENT uses the COMPONENT CONTEXT to get access to the COMPONENT-LOCAL NAMING CONTEXT, as before. It then calls *lookup()*, passing the name of the required CONFIGURATION PARAMETER. The parameter is retrieved from the local list of CONFIGURATION PARAMETERS, as specified in the ANNOTATIONS. No mapping to the global NAMING context is performed, because the required information is COMPONENT-specific, or even instance-specific.

5 Identifying and Managing Instances

This chapter describes three patterns that are commonly used to manage and identify COMPONENT instances.



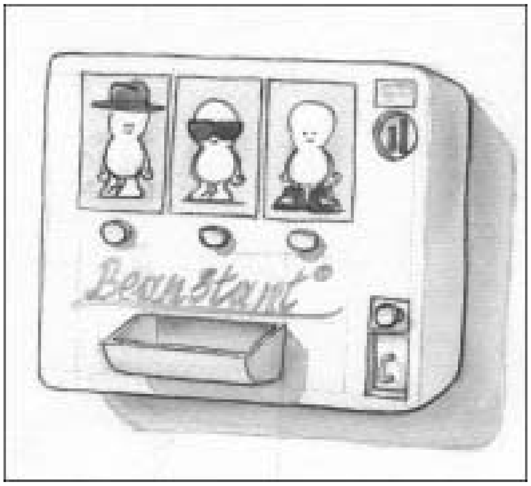
We start with the COMPONENT HOME. It is usually a non-trivial task to create a (logical) COMPONENT instance in the CONTAINER. The COMPONENT HOME provides a simple interface to clients to achieve this and to manage the instances during its lifetime.

For ENTITY COMPONENTS, the situation is more complex, as they have persistent state and a logical identity. You therefore need additional operations in the COMPONENT HOME, for example to look up instances. To achieve this, you need to identify such an instance. PRIMARY KEYS are used for this purpose.

For all other COMPONENTS you might also need a way to store references to an instance persistently, although they have no logical identity. HANDLES do precisely that.

Component Home

You have decomposed the functionality into COMPONENTS. Your application is assembled from collaborating, loosely-coupled COMPONENTS.



❊ ❊ ❊

The procedure for creating or finding COMPONENT instances in the CONTAINER is not trivial. It depends on the way in which the CONTAINER has been implemented and on the behavior required in the ANNOTATIONS. But clients still need to create or find instances, independent of the concrete way how this is done.

A client that wants to invoke operations on a specific COMPONENT instance must first obtain a reference to the instance. A new instance may be created or an older one located and re-used. The exact operations involved depend on the kind of COMPONENT:

- For SERVICE COMPONENTS, the client can connect with *any* instance, because all instances of SERVICE COMPONENTS are stateless and thus equal. The client just needs to access an arbitrary instance from the pool. Locating a specific SERVICE COMPONENT from the pool, and locating a specific instance is not necessary.

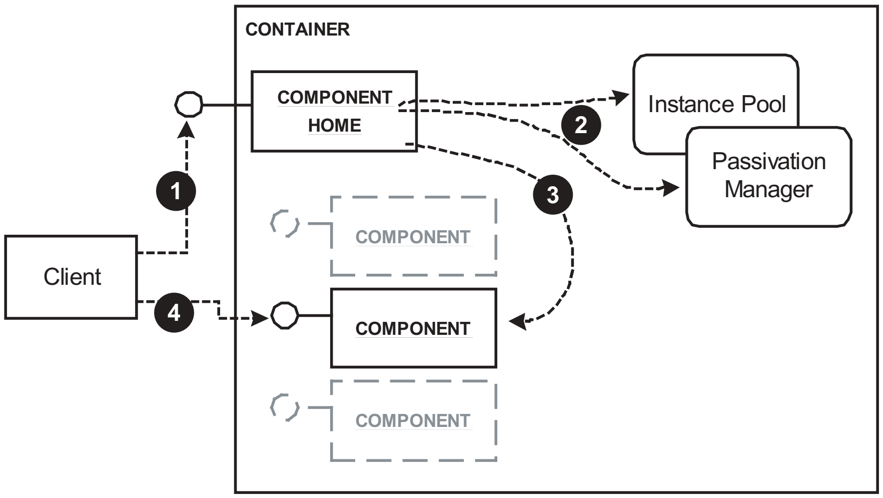
- For SESSION COMPONENTS, creating a new instance really means creating a new physical instance. Locating a specific instance is not necessary, because SESSION COMPONENTS have no identity and are not shared between clients.
- For ENTITY COMPONENTS, creating an instance means that new persistent data is created. As more than one client in more than one session needs to access the persistent data, locating specific entities is necessary. This can be done using the PRIMARY KEY of an instance.

As can be seen from the list above, creating or finding an instance may involve processing logic that depends on the technical concerns, and should thus be handled by the CONTAINER. The use of VIRTUAL INSTANCES makes the situation yet more complex.

The situation is even more interesting in the case of clustered APPLICATION SERVERS, as the concrete CONTAINER in which the instance should be created is also unknown.

Therefore:

For each COMPONENT, provide a COMPONENT HOME that serves as an interface for clients to manage instances of the respective COMPONENT. Depending on the kind of COMPONENT, the COMPONENT HOME provides different operations for creating and locating instances.



Before: Client looks up the **COMPONENT HOME** in **NAMING**
1: Client accesses **COMPONENT HOME** requesting a (new) **COMPONENT** instance
2: Depending on the **COMPONENT** kind, the **COMPONENT HOME** accesses the instance pool or the passivation manager
3: The **COMPONENT HOME** provides an instance to the client
4: The client accesses the instance

❊ ❊ ❊

Note that the COMPONENT HOME is just an interface. This means that the CONTAINER is free to implement it in a way that suits its particular internal structure. The steps required to implement the interface can be arbitrarily complex. This implementation is usually created during COMPONENT INSTALLATION.

The COMPONENT HOME is an implementation of the *Factory* pattern [GHJV94]. As such, it is the clients' only access point to control the lifecycle of logical COMPONENT instances. Note that, because the container uses the concept of VIRTUAL INSTANCES, this has nothing to do with the lifecycle management provided by the CONTAINER for the physical instances coordinated using the LIFECYCLE CALLBACK interface. So if, for example, a client creates a logical instance of a SERVICE COMPONENT with a call to the COMPONENT HOME, in fact no new physical instance is created. Instead, a physical instance from the pool is taken and returned to the client.

To make the programming model simpler, the COMPONENT HOME should have the same 'look and feel' to the programmer as do other COMPONENTS.

The operations provided in the COMPONENT HOME vary depending on the kind of COMPONENT. For all kinds of COMPONENTS, an operation to create new logical instances must be provided. What such a creation operation actually means depends on the way in which VIRTUAL INSTANCES are implemented. In addition, ENTITY COMPONENTS need to be located (based on the PRIMARY KEY and probably other attributes), because they have a logical identity and persistent state. The COMPONENT HOME therefore needs to provide 'finder' operations.

These finder operations often cannot be implemented completely by the CONTAINER. Consider a finder that allows you to locate all *Person* entities with a first name 'Joe'. The business logic for how these entities are found usually has to be implemented by the COMPONENT programmer, for example using an SQL *select* statement. The implementation can be provided as part of the COMPONENT IMPLEMENTATION or as a separate artifact. Actually creating instances from these entities is the job of the CONTAINER and is part of the GLUE-CODE LAYER.

The COMPONENT HOME provides a well-defined way to connect with COMPONENT instances. Of course, now you need a way get in touch with the COMPONENT HOME. To solve this problem, NAMING contains a COMPONENT HOME instance for each COMPONENT.

❊ ❊ ❊

When creating a COMPONENT (Bean) in EJB, the programmer has to define a Home Interface (COMPONENT HOME) in addition to the Bean's Remote Interface (COMPONENT INTERFACE). Depending on the COMPONENT's kind, the developer has to specify several *create()*, *remove()*, and *find...()* operations. For EJB 2.0 and later, the Home Interface of Entity Beans can also be used to define business operations. These operations will be called on Beans in the pooled state and serve as a way of implementing operations that are not bound to a

Bean instance. They might be seen as the equivalent to static methods in Java.

The *create* and *find* operations of the Home Interface can be overloaded with different signatures and serve as factory operations for the Bean: they return one (create operations, find operations) or more (find operations) instances. Note that the programmer never really implements the Home Interface. If Bean-Managed Persistence is used, the developer has to provide implementations for the Home's operations in the COMPONENT IMPLEMENTATION class. If Container-Managed Persistence is used, he does not need to implement the operations at all – they are automatically implemented by the CONTAINER or by a special IMPLEMENTATION PLUG-IN. Some additional specifications, such as an SQL string for the finders, usually have to be specified in the Deployment Descriptor.

In both cases, the actual implementation of the Home Interface itself is generated by the CONTAINER as part of the GLUE-CODE LAYER, to allow for INSTANCE POOLING and PASSIVATION. EJB 2.0 introduced a performance optimization for local invocations (from a co-located Bean). These Beans can use the Local Home and the Local Interface. Conceptually, they are the same as their remote counterparts (Home Interface and Remote Interface) but they can be used only locally.

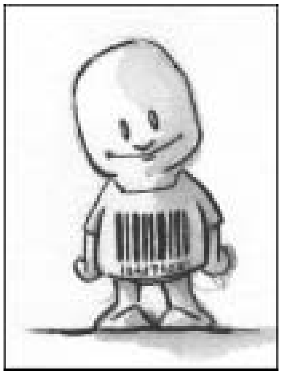
In CCM, declaration of Home Interfaces for COMPONENTS is also necessary. As in EJB, a home manages exactly one COMPONENT type. Home interfaces are by definition normal IDL (Interface Definition Language) interfaces. However, a CIDL shorthand exists. A home can work with PRIMARY KEYS or not, depending on the type of COMPONENT it manages.

Homes provide at least one *create* operation. Homes for COMPONENTS with primary keys also have a *find* and a *destroy* operation. The find operations are based on the PRIMARY KEY. In addition to the default create and find operations, the programmer can declare additional factory and finder operations. Only if such operations are declared does the Home Interface have to be implemented explicitly in a home executor. Otherwise, the home implementation is automatically generated.

In COM+ a COMPONENT (a *CoClass* in COM+ terminology) is the implementation of one or more interfaces. For each COMPONENT, the programmer has to create a factory. Programmatically, the factory itself is a COMPONENT, and its interface must inherit from *IClass-Factory*. It provides an operation *CreateInstance* to create a new, non-initialized instance of the COMPONENT. Upon creation, the programmer can specify which of the COMPONENT INTERFACES implemented by the COMPONENT he would like to receive. The programmer must then initialize the un-initialized instance in some suitable way.

Primary Key

You are using ENTITY COMPONENTS that have a logical identity. A COMPONENT HOME is used to manage instances.



❊ ❊ ❊

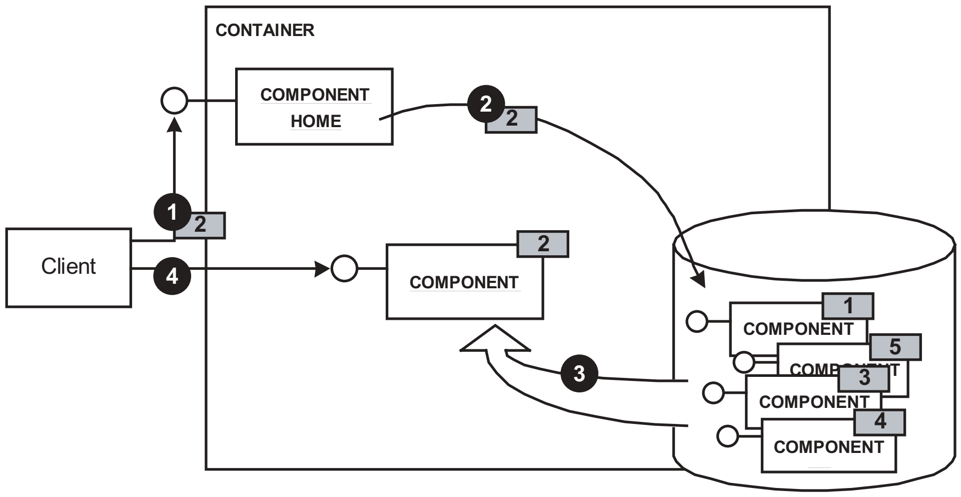
ENTITY COMPONENTS represent business entities, i.e. usually persistent data in some kind of database. You therefore need a way to identify each logical COMPONENT instance, independent of the underlying physical instance, because VIRTUAL INSTANCES make this physical instance change over time. It must also be possible to store references to store references to such logical entities to allow persistent entity relations.

In the case of ENTITY COMPONENTS, the COMPONENT HOME provides operations to locate specific logical entities. This process can be based on different sets of attributes, just as in an SQL *select* statement. However, there must be a way to look up an instance based on a unique identifier. This is necessary, for example, to store references from one COMPONENT instance to another, as in a typical company–employee relationship.

As such a relationship is part of the persistent business model, it must be possible to make the references persistent. Also, to support VIRTUAL INSTANCES (be it PASSIVATION, INSTANCE POOLING, or any other strategy) the references must not lose their semantics. Remote object references cannot therefore be used as a solution here.

Therefore:

Provide a PRIMARY KEY that uniquely identifies an ENTITY COMPONENT instance. The class used as the PRIMARY KEY should be user-definable. Make sure that objects of this class can be passed by value and that the PRIMARY KEY can be easily stored in persistent storage.



Before: Client looks up a **COMPONENT HOME** in **NAMING**
1: Client accesses the **COMPONENT HOME**, providing the **PRIMARY KEY** as a parameter
2: **COMPONENT HOME** accesses the database to locate the data for the instance for the **PRIMARY KEY**
3: Instance is created/taken from the pool/activated. Instance is associated with the **PRIMARY KEY**
4: Client accesses the instance

❊ ❊ ❊

Using this pattern allows your client applications or COMPONENTS to keep a reference to a particular logical ENTITY COMPONENT instance in any way they choose. They can store it in a file or in a database as part of their persistent state, or they can pass the PRIMARY KEY to any other part of the system. The PRIMARY KEY can then be used to look up a specific instance of a COMPONENT using the COMPONENT HOME. ENTITY COMPONENTS have persistent state, and therefore a logical identity, so their COMPONENT HOME provides operations to return a reference to the COMPONENT instance when a PRIMARY KEY is passed to it.

Note that references based on PRIMARY KEYS are 'unidirectional'– the logical COMPONENT instance you reference with a particular PRIMARY KEY object does not know that it is referenced. There is therefore no automatic way to know whether there are references to a particular instance: the instance can be removed without any effect on the PRIMARY KEY or clients or COMPONENTS using it, unless you ensure the opposite explicitly, for example by using constraints in the underlying database.

If a relational database is used to store the persistent state of ENTITY COMPONENTS, the PRIMARY KEY class is typically a holder for the values of the primary key fields in the database – often just a simple string.

You must ensure that the PRIMARY KEY classes really just identify a logical COMPONENT instance, and don't assume anything about the host, CONTAINER, or other technical aspect. This is in contrast to a HANDLE, which is a shortcut to recovering a specific COMPONENT instance. Note that it is a good idea to make the PRIMARY KEY immutable. A modification of the PRIMARY KEY can lead to problems, because it is hard to decide which logical instance was actually modified – its unique identifier, the PRIMARY KEY, is changed and therefore it cannot be used in this case.

❊ ❊ ❊

In EJB, PRIMARY KEY classes are mandatory for Entity Beans. PRIMARY KEY classes can be of any user-defined type as long it is *Serializable*. This ensures pass-by-value semantics for remote invocations. As the state of Entity Beans is usually stored in a relational database, it is important for the performance of the application to use a PRIMARY KEY class that can be mapped easily to one or more database columns. This is especially important, to allow efficient searching in the database.

For APPLICATION SERVERS offering only a simple persistence layer, *integers* or *strings* are commonly used as PRIMARY KEYS, because they can simply be mapped to a *char* or *integer* column in the database.

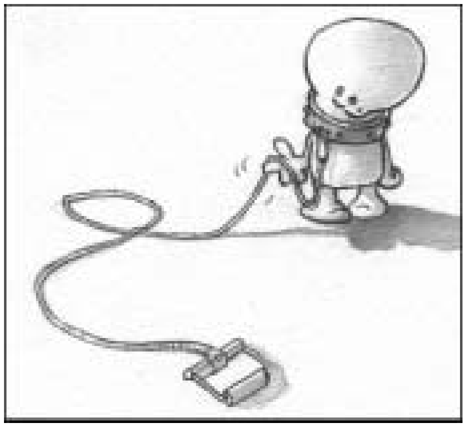
APPLICATION SERVERS that don't offer sophisticated mapping tools often just serialize the PRIMARY KEY and store the resulting byte array as *blobs* (binary large objects) in the database. This is of course very inefficient.

In CCM, basically the same mechanisms are used. ENTITY COMPONENTS and process components, which both have persistent state, are associated with a PRIMARY KEY class. The primary key is a CORBA value type to provide pass-by-value semantics, as in EJB. In CCM, the persistent state of COMPONENTS is specified abstractly using PSDL (Persistent State Description Language). It is then mapped to a concrete persistent store by tools provided by the persistence provider. Specifying how the PRIMARY KEY is represented in the database is thus not part of the CCM standard. For relational databases, the same constraints apply as for EJB.

In COM+, PRIMARY KEYS do not exist because ENTITY COMPONENTS are not supported explicitly. There is a somewhat related mechanism in COM+ – the state of a COMPONENT can be stored locally in a file, et cetera, and can be identified using a *Moniker.* A Moniker can be dereferenced, returning a reference to the instance from which the associated file was created. Internally, a new COMPONENT instance is created and the reference returned to the client, but this has the same state as the original instance. Although not PRIMARY KEYS, Monikers can be used for similar purposes – the file name of the Moniker can be used as the name or the PRIMARY KEY of the instance.

Handle

You are using SESSION, ENTITY or SERVICE COMPONENTS. Clients have references to instances of these COMPONENTS.



❊ ❊ ❊

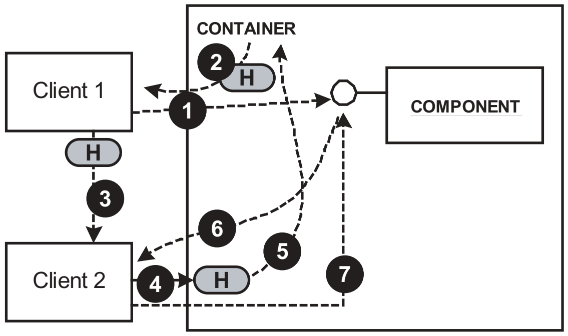
There are situations in which you need to store a reference to a COMPONENT instance persistently although the target instance has no logical identifier, as in the case of SERVICE or SESSION COMPONENTS.

For a SESSION COMPONENT to be of any use, a client has to use the same instance repeatedly over a long period. This period might even span the shutdown of the client-side application. When the application is restarted, the client might want to continue work with the same instance. There is therefore a need to store something locally that is able to return the same instance again.

Even in the case of ENTITY and SERVICE COMPONENTS, such a mechanism is useful. Obtaining an ENTITY COMPONENT instance reference involves several steps – get in touch with NAMING, look up the COMPONENT HOME, find the instance based on the PRIMARY KEY – so a shortcut to re-acquire a specific instance is useful.

Therefore:

Provide HANDLES for each COMPONENT. A HANDLE created by a specific COMPONENT instance knows how to re-acquire the reference to the instance from which it was created. HANDLES can be passed by value or stored persistently. The actual implementation of a HANDLE is the responsibility of the CONTAINER.



1: Client invokes an specific operation on an instance to request a HANDLE
2: HANDLE is created by the CONTAINER and returned to the client
3: HANDLE is stored, or passed to another client
4: Other client accesses HANDLE and requests the original instance reference
5: HANDLE accesses the CONTAINER, which
6: returns the instance from which the HANDLE was created
7: Client accesses original instance

❊ ❊ ❊

From an API point of view, a HANDLE provides an operation such as *getInstance()*, which returns the instance from which it was created. While the interface of HANDLES – the above operation – is predefined, the implementation of this operation depends on the implementation of the CONTAINER.

As a HANDLE is always created by the instance it will return upon invocation of *getInstance()*, you have to have a reference to a COMPONENT instance to be able to obtain a HANDLE to it.

Note that a HANDLE is not the same as a PRIMARY KEY – there are several differences. A PRIMARY KEY identifies a logical COMPONENT instance. It can thus only be used for COMPONENTS that have a logical identity, ENTITY COMPONENTS. A HANDLE can be used for any kind of

COMPONENT, because it cooperates directly with the CONTAINER to re-get the instance from which it was created.

It is interesting to consider what a HANDLE actually returns when it is asked for the COMPONENT instance from which it was created. For SERVICE COMPONENTS, it returns any instance of the COMPONENT – remember that all instances are equal. For SESSION COMPONENTS, it returns the same instance, and for ENTITY COMPONENTS it returns an instance that represents the same logical entity.

A PRIMARY KEY can still be used to look up a COMPONENT instance, even if the CONTAINER and APPLICATION SERVER type change or the location of the COMPONENT HOME has moved, because it identifies a logical business entity. This is not true for HANDLES, because they usually encapsulate the look-up process, and contain information that is no longer valid if the COMPONENT HOME has moved or the CONTAINER implementation changes.

As with PRIMARY KEYS, there is of course no guarantee that the referenced instance still exists. HANDLES are also unidirectional – an instance can be removed even though clients might still have HANDLES for the instance.

❊ ❊ ❊

In EJB, you can call *getHandle()* on any Bean instance. The returned object implements *javax.ejb.Handle*, which provides the functionality to return a reference to the instance that originally created the HANDLE. The method to do this is *getEJBObject()*. The implementation of the HANDLE object is up to the CONTAINER, and is provided to the client as part of the CLIENT LIBRARY. Because it is the CONTAINER's job to implement the functionality, a HANDLE is not portable between APPLICATION SERVERS – migrating your system from one server vendor to another might invalidate the HANDLES. HANDLES are required to be valid after a restart of an APPLICATION SERVER. However, HANDLES to deleted logical instances of Entity Beans, or to Stateful Session Beans that have been removed due to a timeout, are of course invalid.

In CCM, the IOR (Interoperable Object Reference) is basically a HANDLE to the COMPONENT. The IOR can be serialized to a string and converted back to a real reference later on. It can thus easily be stored for reuse after a client restart.

In COM+, HANDLES are also available in the form of *Monikers*, an object that allows you to locate and activate other COMPONENT instances. Technically, the Moniker must implement the *IMoniker* interface. As described in HANDLE, the concrete implementation of Monikers, that is, the way in which other instances are looked up, depends on the type of the Moniker. COM+ provides a set of standard Monikers, for example the file Moniker, used to create an instance based on the contents of a local file. Custom implementations are also possible.

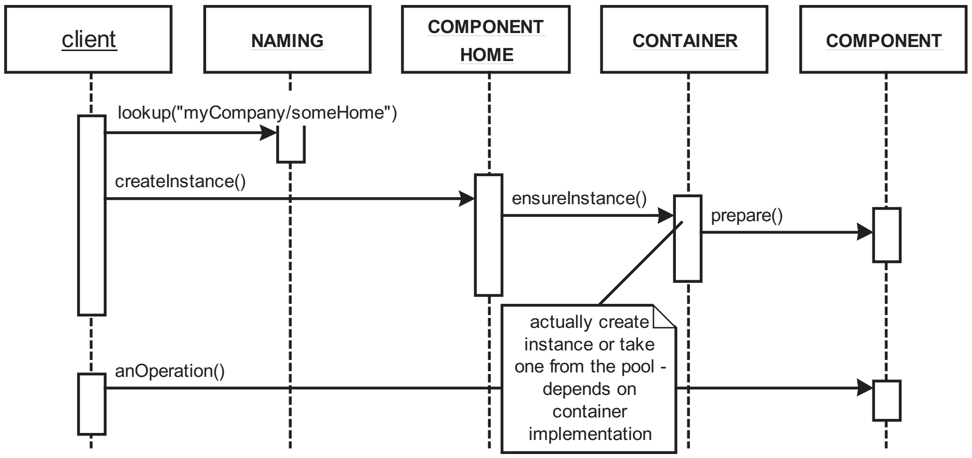
After a Moniker has been created, it has to be bound manually to the COMPONENT instance it is supposed to represent by the programmer.

Summary

This chapter explains how to identify instances and how to manage them. The COMPONENT HOME is used to manage instances. What this management actually entails depends on the kind of COMPONENT.

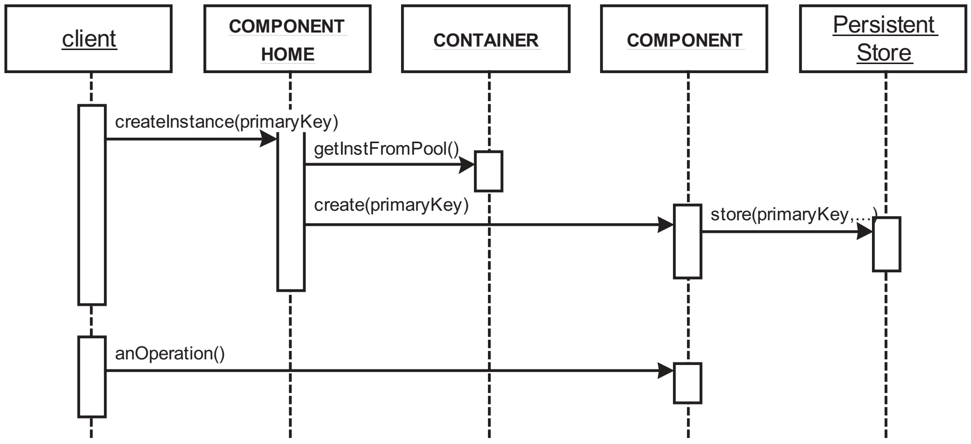
For all COMPONENT types, the COMPONENT HOME provides an operation to create a new instance. For ENTITY COMPONENTS, which have a logical identity represented by a PRIMARY KEY, the home provides an operation to map the PRIMARY KEY to the respective instance. In addition to that, HANDLES provide a kind of 'persistable reference', which, upon request, returns the instance from which it was originally created.

The following sequence diagrams show how these patterns go together. First, we show how a client obtains a reference to a SERVICE COMPONENT.


client
NAMING
COMPONENT HOME
CONTAINER
COMPONENT
lookup("myCompany/someHome")
createInstance()
ensureInstance()
prepare()
actually create instance or take one from the pool - depends on container implementation
anOperation()

The client obtains a reference to the NAMING service – we assume that the client has a reference to the NAMING service of the respective APPLICATION SERVER. It then uses NAMING to look up a reference to the COMPONENT HOME of the respective COMPONENT. The COMPONENT HOME provides a *create* operation, which the client uses to obtain a reference to a suitable instance. Because SERVICE COMPONENTS are stateless, the client need not pass information to the COMPONENT HOME that identifies a particular instance – they are all equal. Thus, it is the CONTAINER's decision which instance is returned to the client. Here, the CONTAINER takes an instance from a pool, initializes it, and returns it to the client.

For ENTITY COMPONENTS the situation is a little different, because the PRIMARY KEY of the instance is involved in most operations. The following sequence diagram shows how a client creates a new ENTITY COMPONENT instance.

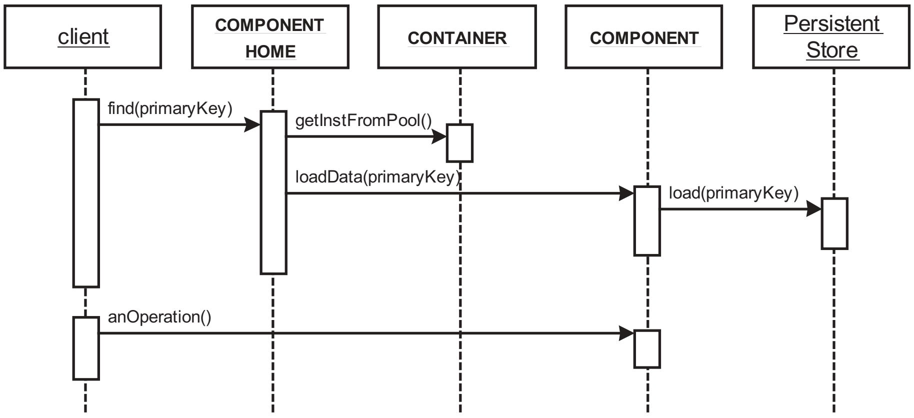


When the client calls the create operation for an ENTITY COMPONENT, the PRIMARY KEY that should be associated with the new instance has to be passed to the COMPONENT HOME[1]. The COMPONENT HOME then contacts the CONTAINER to retrieve an instance of the ENTITY COMPONENT from the pool (assuming the CONTAINER uses INSTANCE POOLING). At this stage, the instance is still 'anonymous' – it does not yet represent a logical identity. In the next step, the COMPONENT HOME calls a suitable *remove* operation on this instance, again passing in the PRIMARY KEY. The implementation of this operation creates a new record in the underlying database, thus creating a persistent representation of the ENTITY COMPONENT's state. The instance, which at this point represents the logical identity identified by the respective PRIMARY KEY, is then returned to the client, ready to receive method invocations.

Because ENTITY COMPONENTS have a logical identity and are persistent, they can be found later in another session or from another client.

1. Alternatively, a *create* operation can also take other parameters from which the PRIMARY KEY can be inferred, for example, a first name, a family name and a birth date might be used to construct a (hopefully) unique PRIMARY KEY. The client can then obtain the PRIMARY KEY from the newly-created instance. If the PRIMARY KEY is opaque and can be generated independently of the attributes of the instance, it is also possible to call the *create* operation with no parameters.

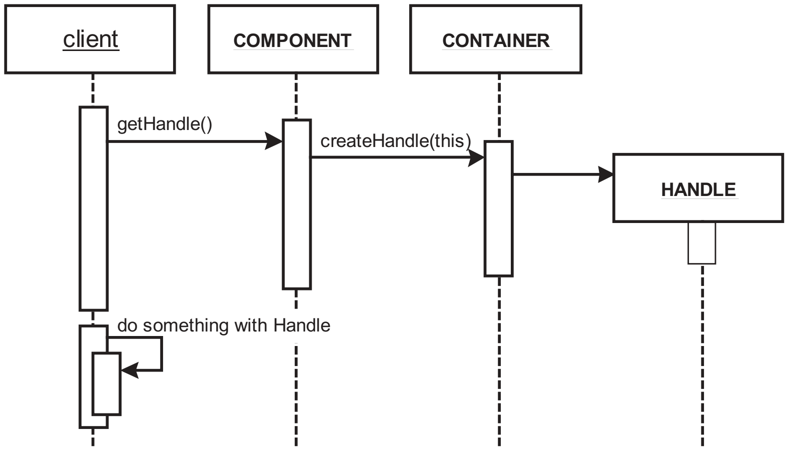
To make this possible, the client needs to know something that identifies the instance. The COMPONENT HOME provides finder operations to map this information to one or more COMPONENT instances. The simplest way to do the look-up, of course, is to use the PRIMARY KEY, as illustrated in the next sequence diagram.



The client calls *find()* on the COMPONENT HOME, passing in the PRIMARY KEY of the instance it wishes to find. Again, the COMPONENT HOME uses the CONTAINER to obtain an anonymous instance of the COMPONENT. The CONTAINER then calls a *load* operation that retrieves the state of the logical identity defined by the PRIMARY KEY from the database. The instance now represents the requested logical identity and is returned to the client for method invocations. At some later time, for example at the end of the current transaction or after the invocation has been handled by the instance, the CONTAINER calls a suitable LIFECYCLE CALLBACK operation to save the state back to the persistent store and return the instance to the pool.

Last but not least, let's have a look at how HANDLES are used. Using HANDLES involves two steps. Obtaining a HANDLE, and using the HANDLE to obtain the instance from which it was created.

Obtaining a HANDLE is straightforward, as the next diagram shows.



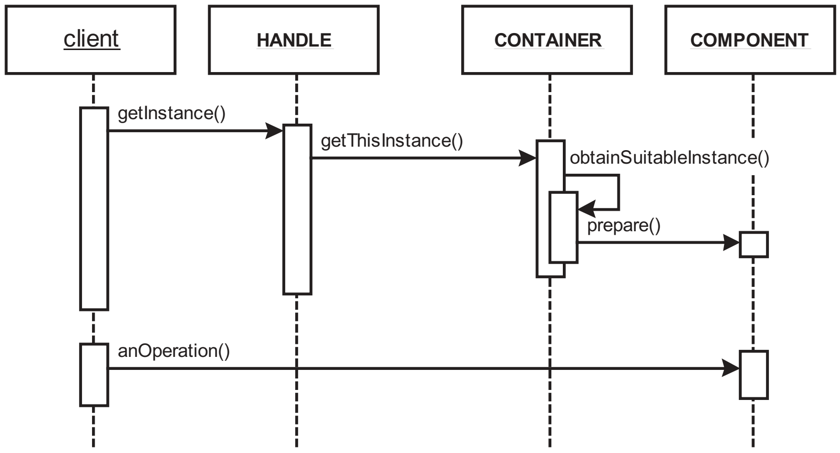
A client has a reference to a particular COMPONENT instance for which the client wants to a HANDLE. It calls a *getHandle()* operation on the instance. Internally, the instance contacts the CONTAINER, which creates the HANDLE. The HANDLE contains some logical identifier that identifies the instance from which it was created. What this identifier actually contains and how it is structured depend on the CONTAINER. However, it must contain enough information to locate the instance. This includes even on which specific APPLICATION SERVER the COMPONENT is located.

Later, a client uses the HANDLE to connect to the instance from which it was created. This is really only useful if one of the following conditions is true:

- Considerable time elapses between getting the HANDLE and using it
- The HANDLE has been stored persistently in the meantime
- The HANDLE is received over a network

If not, the reference to the instance could merely have been retained.

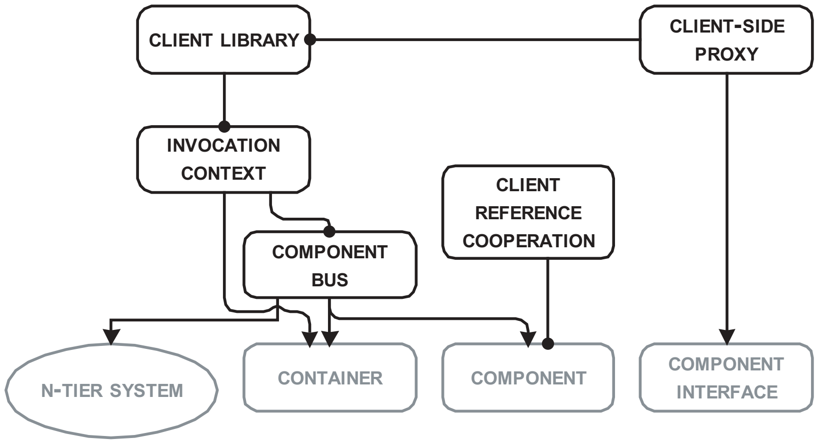
Assume that you have got in touch with a HANDLE and you now want to get recover the instance from which it was created.



To do this, the client calls a *getInstance()* operation on the HANDLE. This results in the HANDLE contacting the CONTAINER, passing it the internal identifier that the CONTAINER passed to the HANDLE when it was created. The CONTAINER uses this identifier to fetch the respective instance and return it to the client. The details of how it does this depend on the implementation of the CONTAINER.

6 Remote Access to Components

This chapter deals with the problems involved in remote access to COMPONENTS. Note that we do not cover general remote access patterns, as they are described in the literature, for example *Broker* [BMRSS96] or *Reactor*/*Proactor* [SSRB00].



As COMPONENTS run in a CONTAINER, possibly on a separate server, access to COMPONENTS really means remote access to the CONTAINER. A COMPONENT BUS achieves this, hiding the details of the underlying low-level network protocols. Inside the APPLICATION SERVER, the CONTAINER serves as the adapter between the COMPONENTS and the COMPONENT BUS. However, clients outside the APPLICATION SERVER also need to access the COMPONENT BUS. A CLIENT LIBRARY that contains all the necessary files and configuration is used for this purpose. In particular, it contains a CLIENT-SIDE PROXY that acts as a local receiver for method invocations on remote COMPONENTS, forwarding the invocation to the COMPONENT BUS.

To allow the CONTAINER to handle technical concerns correctly, it needs more information about a method invocation than just the target instance and the method's name and parameters. It also needs transaction and security data. Each invocation therefore carries an INVOCATION CONTEXT, which contains all this additional information.

In some circumstances, the CONTAINER needs a client's help with managing the lifecycle of COMPONENT instances. CLIENT REFERENCE COOPERATION shows when this might be necessary and how it can be implemented.

Component Bus

You use a CONTAINER to host your COMPONENTS. To implement an N-TIER ARCHITECTURE, remote access to these COMPONENTS is necessary.

❊ ❊ ❊

In an N-TIER ARCHITECTURE, your COMPONENTS can reside on other machines than the client applications, or within the same process (CONTAINER or APPLICATION SERVER). Client programming should be the same no matter whether the accessed COMPONENTS are located locally or remotely. It is also critical that remote access performs well with many COMPONENT instances and clients – quality of service requirements (QoS) must be realized.

Providing remote access to objects is a problem that has essentially been solved by object-oriented middleware such as CORBA [OMGCORBA], DCOM [MSDCOM] or Java's RMI [SUNRMI]. Patterns like *Broker* [BMRSS96] or *Remote Proxy* [GHJV94] describe extensively how such systems are implemented.

However, the solutions provided by these patterns are not enough for component architectures:

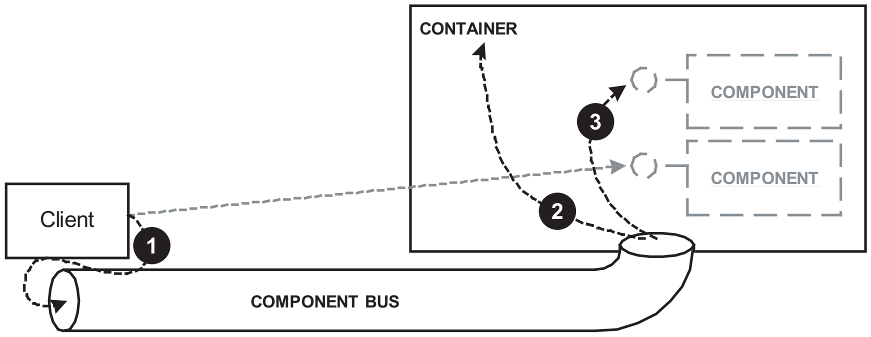
- First, clients or client COMPONENTS should not need to be changed when the underlying transport protocol changes, for example from RMI's JRMP to IIOP or DCE/RPC. A COMPONENT or client developer should not have to deal with any protocol-specific aspects – everything should be handled from the CONTAINER as part of the technical concerns.
- Second, clients should not need to care about the actual location of a specific COMPONENT instance, because this is a technical concern. This is even more true in the face of relocation of COMPONENTS for reasons of fail-over or load-balancing.
- Third, the remote access technology must be integrated with the implementation of VIRTUAL INSTANCES. Depending on how they are implemented, there might be varying numbers of physical COMPONENT instances in the CONTAINER. Remote access must still perform well for very large numbers of physical (or even logical) instances and many clients. For example, it is usually not feasible to use a separate network connection or socket for each physical instance.

When the caller and the invoked instance are located in the same process, the invocation should be optimized. The overhead compared to a normal method invocation should be as low as possible. This scenario is quite typical – it happens whenever a COMPONENT invokes an operation on another COMPONENT that is located in the same CONTAINER or APPLICATION SERVER.

There should also be a way to change the transport protocol, ideally without effect on client programs and COMPONENT IMPLEMENTATIONS. It should be possible to transport COMPONENT invocations synchronously or asynchronously. QoS parameters such as guaranteed delivery for asynchronous invocations or request timeouts should also be configurable on the CONTAINER side by the administrator or COMPONENT developer, for example, by using ANNOTATIONS, but the *client* programmer should not need to care worry about these issues.

Therefore:

Access COMPONENTS only via a COMPONENT BUS, a generic communication infrastructure for remote and local COMPONENT access. Because the COMPONENT BUS is integrated with the CONTAINER, it can provide efficient access to a large number of COMPONENT instances. Because it knows the location of the caller and the invocation target, it can optimize local invocations efficiently. The COMPONENT BUS hides the underlying low-level transport protocol, its semantics and its QoS attributes. Its configuration is done with the help of ANNOTATIONS.



1: Client invokes an operation on the remote VIRTUAL INSTANCE
2: COMPONENT BUS accesses the CONTAINER to manage the technical concerns
3: Invocation is forwarded to the VIRTUAL INSTANCE

❊ ❊ ❊

A generic COMPONENT BUS into which clients and COMPONENTS can be plugged allows you to relocate COMPONENTS and their clients in anyway you like. This builds the basis for load-balancing and failover.

Because of its generic nature, the communication protocol used by the COMPONENT BUS can be switched according to technical requirements such as QoS, performance, standards conformance or the underlying physical transport layer. The standard distributed object communication technologies (such as CORBA, RMI) or messaging middleware are usually used.

To control technical aspects of the underlying transport protocol, a specific form of ANNOTATIONS can be used. Here, the QoS attributes, the selected protocol, or some performance optimizations can be specified.

While the primary task of the COMPONENT BUS is of course to transport method invocation requests and their results, this is not enough – transaction and security information must also be communicated. To allow the CONTAINER to take care of technical requirements, it needs to know things like the current transaction or the security context. An INVOCATION CONTEXT is used for that purpose. The CLIENT LIBRARY is used to furnish this information on the client side.

The COMPONENT BUS is also responsible for optimizing resource usage, especially for sockets or other network I/O. In most cases, one remote connection is used to communicate with many, sometimes even all, instances of a specific COMPONENT. To be able still to deliver the invocations to the correct instance, the COMPONENT BUS needs to cooperate with the COMPONENT PROXY. As a minimum, the COMPONENT BUS has to transport the identifier of the logical invocation target. On the server side, the COMPONENT BUS usually collaborates with the COMPONENT PROXY to access VIRTUAL INSTANCES. For details about building efficient remote communication servers, see [SSRB00].

It is not always possible to hide the underlying protocol completely. This is especially true in the case of asynchronous communication based on message-oriented middleware (MoM). For example, to exploit the benefits of asynchronous communication, the client programming model has to be aware of the asynchronicity. The same is true inside COMPONENT IMPLEMENTATIONS – because MoM does not invoke business methods on the target (it just delivers messages), a method to handle incoming messages must be provided. This might even have consequences for the COMPONENT INTERFACE.

Note that in the case of local invocations, it is not possible to use ordinary method invocations – the INVOCATION CONTEXT has to be propagated and technical concerns have to be enforced by the CONTAINER.

❃ ❃ ❃

In EJB, the COMPONENT BUS is implemented using Java's Remote Method Invocation (RMI) feature. RMI can work with different low-level communication protocols, for example JRMP, which is RMI's native transport protocol, or IIOP, CORBA's TCP/IP-based transport protocol. Up to EJB 1.1, the COMPONENT BUS only supports synchronous invocations. In version 2.0, asynchronous invocations based on the Java Messaging Service (JMS) are available. This is not transparent to the Bean developer, however, because a specific kind of Bean has to be used, *Message-Driven Beans*. Message-Driven Beans (MDBs) are basically Stateless Session Beans with a specific, predefined interface that can handle incoming messages. An MDB is attached to a specific JMS queue or topic. To access an MDB, a client has to post a message to the respective queue or topic. Thus, asynchronous delivery is also not transparent to the client.

EJB's COMPONENT BUS does not support additional QoS configuration. Depending on the underlying protocol, the INVOCATION CONTEXT can be transported directly (as in IIOP) or manually (as in JRMP).

EJB 2.0 introduced a feature called *Local Interfaces*. With these, the developer specifies a separate COMPONENT INTERFACE and COMPONENT HOME for use in local invocations. Because Local Interfaces can only be invoked locally, the CONTAINER can significantly optimize these invocations. In this respect, this is a 'manual' optimization of the COMPONENT BUS.

CCM, being CORBA-based, uses CORBA as its COMPONENT BUS. CORBA itself supports 'pluggable' protocols – these can be native to the ORB or they can be one of the standard communication protocols such as GIOP or IIOP. Asynchronous delivery can be achieved in two ways, either by declaring an operation *oneway* (which basically uses a separate thread for method invocations), or by using more sophisticated methods based on callback interfaces or polling objects, as introduced with CORBA 3's AMI (Asynchronous Messaging Interface) feature. Note that the latter two possibilities are not transparent either for the COMPONENT developer or for the clients. COMPONENT IMPLEMENTATION is simplified, because a lot of the necessary code for the COMPONENT BUS is generated during COMPONENT INSTALLATION.

COM+ uses the same communication protocol as DCOM, Microsoft's version of DCE RPC. It is possible to use asynchronous communication based on the Microsoft Message Queue (MSMQ). In this case, the CLIENT-SIDE PROXY records the method invocation and posts a corresponding message to a predefined message queue. On the server side, the COMPONENT PROXY listens to the queue, decodes the message and invokes the operation on the COMPONENT. Except for some limitations to the method signature (no return values, no parameters passed by reference), asynchronicity is just a matter of configuration during COMPONENT INSTALLATION. Semantically this mechanism is similar to CORBA's one-way operations – however, it uses a real MoM for transport.

Invocation Context

You are using a CONTAINER to take care of the technical requirements. A COMPONENT BUS provides remote access to COMPONENTS.

❊ ❊ ❊

To be able to manage technical concerns such as transactions or security, the CONTAINER needs to know more than just the name of the invoked method and its arguments when an operation is called. This additional information cannot be supplied with a normal method call, because typical programming language syntax and semantics do not allow such additional information to be passed.

Consider transactions. For transactions to work correctly, the *transaction context* must be transported. The transaction context identifies the transaction that is active when an operation is invoked. Only then can the CONTAINER make sure that the operation that is invoked runs in the context of the current transaction.

Note that a COMPONENT PROXY alone does not help here. The necessary information has to be transported between clients and the CONTAINER, or between different CONTAINERS when client COMPO-

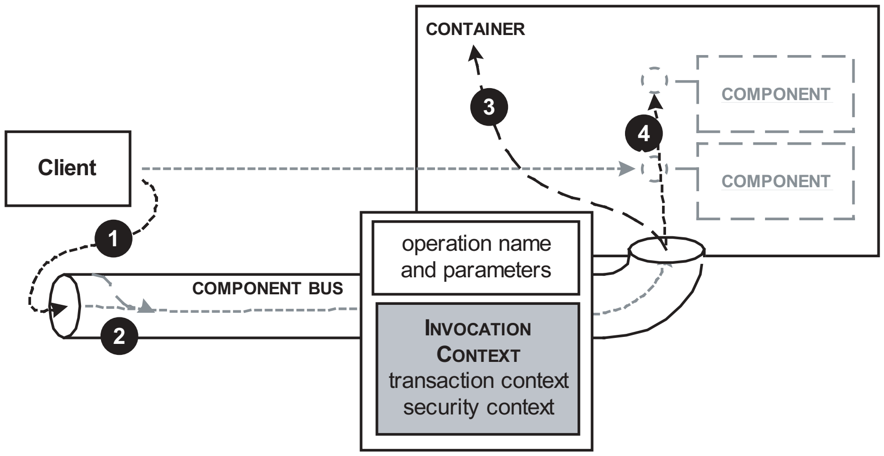
NENTS reside in another CONTAINER, or even APPLICATION SERVER, to the invocation target. It may even have to be transported between different APPLICATION SERVERS, if the source and target containers reside in different APPLICATION SERVERS.

In the case of security, the same problem exists. The security context, which includes the caller identifier and its security privileges, must be available to the CONTAINER, which includes the caller identifier and its security privileges. Here it is obvious that this information must be transported from the client to the APPLICATION SERVER, because security data is inherently client-specific.

Note that it is usually not a good idea to add the required information as additional parameters to the operation invocation, for two reasons. First, you don't want the developer to be concerned with such things when he programs functional COMPONENTS, as they clutter the operation signatures. Second, the kind of information required may not be known at build or compile time, because it depends on the implementation of the CONTAINER.

Therefore:

Include an INVOCATION CONTEXT with each operation invocation. This context contains all the information necessary for the CONTAINER to handle the technical requirements. The concrete structure of this context depends on the CONTAINER. It is the job of the COMPONENT BUS to transport this additional data. The CLIENT-SIDE PROXY must insert the data on the client side.



1: Client invokes an operation on a remote VIRTUAL INSTANCE
2: COMPONENT BUS adds the data for the INVOCATION CONTEXT
3: COMPONENT BUS forwards the data from the INVOCATION CONTEXT to the CONTAINER
4: It forwards the invocation to the VIRTUAL INSTANCE

❊ ❊ ❊

To make this pattern work, the COMPONENT BUS must be able to transport the INVOCATION CONTEXT. There are two ways in which this can be achieved. Either the underlying transport protocol must support such context transfers directly, or you need to use some kind of *gateway object*. This provides an operation *invoke(target, operation, params[], context)* that passes the information about the method to be invoked explicitly, including the INVOCATION CONTEXT.

Independent of how the context propagation is implemented, it is only present 'under the hood'. The real functional COMPONENT instance is not required to be able to handle such contexts. If parts of the information are necessary for the COMPONENT itself, make it accessible using the COMPONENT CONTEXT.

A disadvantage of using this pattern is that this context information makes remote messages larger and thus impacts performance. It is therefore not suitable for applications that are overly constrained in these respects.

Note also that if the CONTAINER relies on the presence of such an INVOCATION CONTEXT, direct local method invocations – for example from one COMPONENT instance to another – are no longer possible. They need to include the INVOCATION CONTEXT, and local method invocations that do not use the COMPONENT BUS do not include it, as discussed above[1].

The proposed solution is called *Implicit Context Propagation*, as opposed to *Explicit Context Propagation,* in which context information is explicitly handled by the application programmer. For the reasons outlined above, explicit context propagation is not suitable for component architectures.

❊ ❊ ❊

In EJB, security and transaction information is transported transparently from the client to the COMPONENT, as described above. In most cases the Bean implementation never uses this information directly. Instead, the COMPONENT PROXY uses the information to allow the CONTAINER to handle technical concerns. The Bean only needs to access the information in some cases, for example to query the state of the actual transaction, or to obtain information about the caller in order to implement advanced security. The information from the INVOCATION CONTEXT is made available to the COMPONENT through the COMPONENT CONTEXT. Depending on the protocol used in the COMPONENT BUS, the INVOCATION CONTEXT can be transported directly (as in IIOP) or manually (as in JRMP).

CCM uses IIOP as a transport protocol. IIOP allows context information to be attached directly in invocations. As before, transaction and security information are the most prominent data items transported in the INVOCATION CONTEXT. In the cases in which the handling of technical concerns by the CONTAINER is not enough, the COMPONENT can access the required data through the various interfaces provided by the COMPONENT CONTEXT.

1. Depending on the threading policies of the CONTAINER, the INVOCATION CONTEXT can be propagated to local invocations by attaching the data to the current thread. See the *Thread-Specific Storage* pattern in [SSRB00].

COM+ transport protocol also allows context information, such as security and transaction state, to be attached. The affected COMPONENT can obtain this information using the COMPONENT CONTEXT, as well as several specialized interfaces that can be accessed through the COMPONENT CONTEXT.

Client-Side Proxy

Using a COMPONENT BUS, you remotely access your COMPONENTS in the CONTAINER.

✲ ✲ ✲

The client programming model should be as simple as possible. In particular, the remote location of the COMPONENTS and access to the COMPONENT BUS should have as little impact on the client programming model as possible. The necessary data for the INVOCATION CONTEXT also need to be provided somehow. This might even include the remote instance's identity, depending on the way in which the CONTAINER implements VIRTUAL INSTANCES.

Client programming should be as intuitive as possible. As in every distributed, object-oriented environment, client method invocations should happen in the same way as local operations[2]. The conversion of the invocations into remote method invocations and the attachment to the COMPONENT BUS should happen automatically.

However, in a COMPONENT environment, things are a little more complicated than in a purely distributed object situation. This is because things like the transaction context or the client's security data

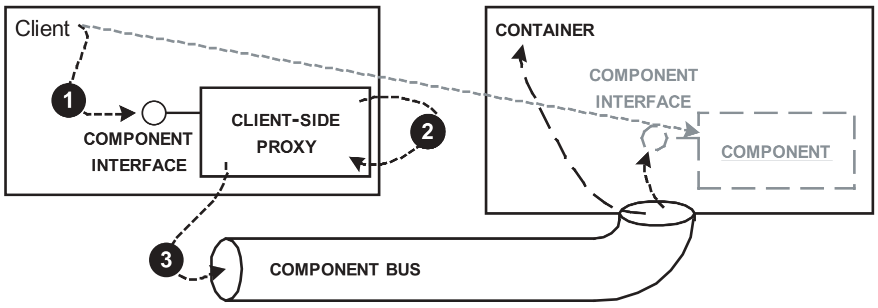
2. Remoteness is never really transparent, of course. Performance is worse, and new kinds of errors are typically introduced – see the SYSTEM ERRORS pattern.

have to be transported with every method invocation. This is done using an INVOCATION CONTEXT, but the necessary data has to be provided to the COMPONENT BUS with every invocation, to allow the COMPONENT BUS to insert the necessary data into the INVOCATION CONTEXT.

There is another problem that is related to the implementation of VIRTUAL INSTANCES and the COMPONENT BUS. Depending on how VIRTUAL INSTANCES are implemented in the CONTAINER, and how the COMPONENT BUS manages remote connections, the INVOCATION CONTEXT might also need to include an identifier of the VIRTUAL INSTANCE for which the invocation is intended.

Therefore:

Provide a CLIENT-SIDE PROXY for remote COMPONENTS that implements the respective COMPONENT INTERFACE and accepts local method invocations. The proxy forwards the invocation data to the COMPONENT BUS and also supplies the necessary data for the INVOCATION CONTEXT.



1: Client invokes an operation on a VIRTUAL INSTANCE, in reality it reaches the CLIENT-SIDE PROXY
2: Proxy adds the data for the INVOCATION CONTEXT
3: Proxy hands over the invocation to the COMPONENT BUS

❊ ❊ ❊

The CLIENT-SIDE PROXY is typically a generated artifact. It has to implement the COMPONENT INTERFACE of the remote COMPONENT that it represents, but its implementation is highly stereotypical, thus it can be generated easily. The generation is usually done during COMPONENT INSTALLATION.

The generated proxy is then packaged into the CLIENT LIBRARY. This library contains additional artifacts necessary for the client and is created during COMPONENT INSTALLATION.

How the CLIENT-SIDE PROXY works internally depends very much on how the COMPONENT BUS and the CONTAINER work. They all have one thing in common – for each remote reference to a COMPONENT instance, there is one local proxy object. The proxy objects are usually very small and just forward the invocation to the COMPONENT BUS. Doing this, they supply the COMPONENT type, the instance identifier and information about the invoked operation, as well as the data for the INVOCATION CONTEXT. In most cases, all the CLIENT-SIDE PROXIES use one remote connection provided by the COMPONENT BUS. If this was not the case, there would be potentially a lot of remote connections, risking a performance bottleneck.

Instances of the proxy are created when the COMPONENT BUS returns references to COMPONENT instances as a consequence of calling a method on a COMPONENT HOME.

Whether COMPONENT-specific code is required for the CLIENT-SIDE PROXY, or whether it can be implemented generically with only definitions and parameters, depends on the component architecture, and in particular on the COMPONENT BUS. If the programming language supports reflection a generic implementation is possible. But this is not always desirable from a performance point of view. Typically, the bare minimum usually consists of header files or other programming language artifacts that define the COMPONENT INTERFACE of the COMPONENT that should be accessed.

Note that this pattern is also used in many distributed object systems and has been described several times in, for example in [GHJV94] (the *Remote Proxy*) and in [BMRSS96]. We include it here for completeness and because it has the additional responsibility of providing the INVOCATION CONTEXT data to the COMPONENT BUS.

❊ ❊ ❊

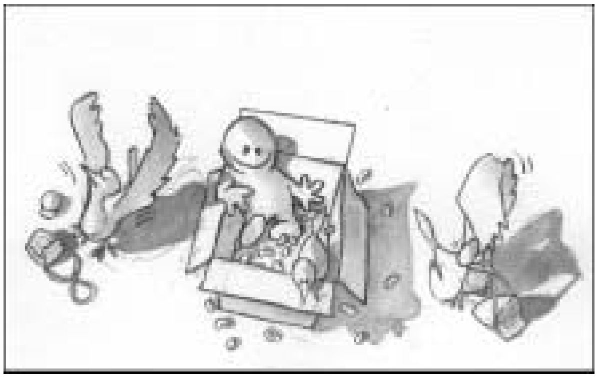
In EJB, the CLIENT-SIDE PROXIES are called *stubs*. They are usually part of the CLIENT LIBRARY and have names like *_MyComponentStub*. Instead of providing them as part of the CLIENT LIBRARY, they can also be downloaded from the server if the underlying RMI subsystem is configured appropriately. The internal implementation varies greatly – in many cases, the mechanisms of the Java reflection API are used.

In CCM, references to remote COMPONENT instances are basically remote object references. Remote objects are accessed in CORBA using generated stubs, which is an implementation of this pattern.

In COM+, the same mechanism is used. The proxy is part of the marshalling DLL.

Client Library

Using a COMPONENT BUS and a CLIENT-SIDE PROXY, you remotely access your COMPONENTS in the CONTAINER.



❊ ❊ ❊

Clients use a CLIENT-SIDE PROXY to access remote COMPONENTS as if they were local objects. But the proxy class has to be made available to clients. Clients also need access to several other helper classes, as well as to the types of operation parameters and their return types. Configuration data must be made available to clients to allow them to bootstrap and contact the APPLICATION SERVER. Finally, the code to attach clients to the COMPONENT BUS itself must be available to client programs.

The CLIENT-SIDE PROXY is usually a programming language class. This class must be available to the client programs to allow them to use it in their process and to access the instances locally.

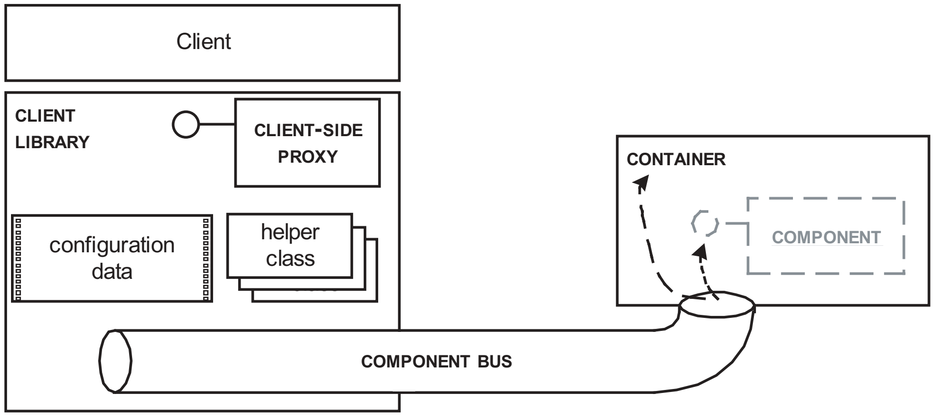
If a COMPONENT uses certain types of parameters that are not part of the programming-language specific standard libraries, these types must also be supplied to the client so that it can be compiled and linked successfully.

A certain amount of configuration data is always necessary to run a client application successfully. For example, the network address of the APPLICATION SERVER has to be defined at the client side.

Code must also be available on the client site that attaches the CLIENT-SIDE PROXIES to the COMPONENT BUS and that implements the COMPONENT BUS itself. This implementation must be compatible with the implementation of the CONTAINER itself, because it cooperates with the CONTAINER to ensure the correct semantics of VIRTUAL INSTANCES.

Therefore:

Create a CLIENT LIBRARY upon COMPONENT INSTALLATION on the server. Most importantly, this contains the CLIENT-SIDE PROXY that is the placeholder for any remote COMPONENTS in the client processes. It also contains any other parametrization or initialization parameters that are necessary for the client to access the CONTAINER and APPLICATION SERVER remotely, as well as a suitable implementation of the COMPONENT BUS.



✼ ✼ ✼

Using this pattern allows the client application developer to access COMPONENTS relatively transparently, while technical concerns are handled by the CONTAINER. Note that the CLIENT LIBRARY is under control of the CONTAINER. It is up to the CONTAINER to provide all the information the client requires, and to generate the CLIENT-SIDE PROXY in a way that suits its needs.

Not everything is hard-wired in the CLIENT LIBRARY. There are usually initialization parameters that can be set when the application is started to prevent the CLIENT LIBRARY having to be recreated all the time. For example, the network address of the APPLICATION SERVER is usually specified when the application starts.

If the client programming environment supports access to the COMPONENT BUS directly, the CLIENT LIBRARY does not need to contain specific code to access it. However, the COMPONENT BUS usually has some CONTAINER-specific characteristics to support its implementation of VIRTUAL INSTANCES, so some COMPONENT BUS code is almost always required.

❊ ❊ ❊

In EJB, most deployment tools create a client *.jar* file automatically during COMPONENT INSTALLATION. This needs to be installed in the client's *classpath* and is used by the client to access the COMPONENT on the server. The infrastructure aspects, such as security classes, and the COMPONENT BUS access itself, is provided in the form of a *.jar* file that is generic for the CONTAINER and must be made available separately to the clients. Optionally some of these classes can also be downloaded from the APPLICATION SERVER 'on the fly'.

In CCM, because of the lack of implementations, there are no concrete examples of CLIENT LIBRARIES or a running CCM system, but it is expected that this will work in the same basic way as in EJB. CORBA, the underlying transport layer, also uses generated classes that need to be made available to the client, so in CCM it will be the same.

In the case of COM+, a DLL is generated, called the *marshaling* DLL, that contains proxies and stubs to allow the client to contact the server object. It is dynamically linked to the client application.

Client Reference Cooperation

You are using COMPONENTS that run in a CONTAINER. Clients access the COMPONENTS remotely.

❊ ❊ ❊

Although the CONTAINER uses VIRTUAL INSTANCES to save memory, it still needs to know when instances are no longer required. However, because clients are located in different address spaces, traditional garbage collection techniques cannot be used. The CONTAINER does not know, for example, how many clients use a specific COMPONENT instance, and thus it does not know when an instance is no longer needed. The situation is further complicated by unreliable network connections and crashing clients.

In a purely local environment it is fairly easy to find out when an instance is no longer required. In languages that support garbage collection, the normal garbage collector can be used. In environments that do not, the mechanisms provided by the programming language for deleting unused instances can be used.

For distributed environments the situation is more complicated. Distributed garbage collection is often inefficient or technically unfeasible. In particular, you must make sure that the algorithms for instance deletion still work correctly – although perhaps less efficiently – in the face of unreliable network connections or crashing clients. For example, a crashed client cannot explicitly free a reference to an instance, so memory leaks might result.

The importance of solving the problems mentioned above depends on how the CONTAINER is implemented, and in particular whether INSTANCE POOLING or PASSIVATION is used. In the case of ENTITY COMPONENTS there is no problem, because they are (logically) alive until they are explicitly destroyed. Destruction (or deletion) is part of the business logic and not part of the technical concerns of distributed object lifecycle management. Before COMPONENT deletion, the CONTAINER can use INSTANCE POOLING or PASSIVATION to keep the system load reasonably low.

For SERVICE COMPONENTS there is also no issue. They have no state, so if an instance that is referenced is destroyed, the CONTAINER simply creates a new one when one is requested. INSTANCE POOLING can be used to keep the creation and destruction overhead minimal.

For SESSION COMPONENTS the situation is different. A COMPONENT must only be destroyed in the CONTAINER if it is no longer needed by any client. You could provide a method *destroy()* to signal that the SESSION COMPONENT is no longer needed by the client. However, if it is legal to access a SESSION COMPONENT from multiple clients and a reference to the SESSION COMPONENT is passed, this must also be taken into account somehow – the SESSION COMPONENT must not be destroyed unless all clients have called *destroy()*. Basically this implements a garbage collector based on reference counting. Note that the real problem is that a client might crash and therefore never call *destroy()*. In this case the SESSION COMPONENT would never be garbage collected, because the CONTAINER assumes that the client still holds a reference.

Therefore:

Make sure that the clients cooperate in distributed garbage collection. This can be implemented in different ways, for example heartbeats, pinging, or *leases* (see below). In all cases the CONTAINER has a means of determining when it can destroy the

instance. Each strategy has different consequences in the case of unreliable networks and clients.

❊ ❊ ❊

Using this pattern allows the CONTAINER to know when a COMPONENT instance is no longer needed by any part of the system. In more detail, the three possibilities are:

- *Reference counting*. Whenever the client obtains a reference to a COMPONENT instance, the client calls a specific operation on the instance to increment a reference count. It calls another operation when it has finished using the COMPONENT to decrement the count. When the reference count has reached zero, the CONTAINER can safely remove the COMPONENT. However, this has drawbacks in practice. If the reference count is incremented or decremented at the wrong points, the distributed garbage collection will be erroneous. A crashed client will also leave a dangling reference, because it cannot decrement the reference counter correctly.
- *Heartbeats or pinging*. This solution uses regular signals from the client to the CONTAINER (*heartbeat*) or from CONTAINER to the clients (*pinging*). If the heartbeat does not arrive or the ping fails, the client is assumed to be dead and all references are considered void, decrementing the reference count automatically. Heartbeats or pinging functionality can be implemented automatically as part of the CLIENT LIBRARY and the GLUE CODE LAYER/CONTAINER. It does of course incur a network overhead.
- *Leases*. A *lease* is a temporarily-restricted 'license' to use a COMPONENT instance. When the lease expires, the reference is considered void, and the respective reference count is decremented. The client must update the lease regularly if it wants to continue using the instance. Jini uses this pattern extensively, and the *Leases* pattern is described in the *Jini Pattern Language* [JINIPL]. Note that leases and heartbeats are really the same: a signal that is repeated from time to time to ensure that the referenced object is not garbage collected.

The complexity and performance overhead of all these solutions is the reason why some COMPONENT technologies, such as EJB, do not use this pattern at all. They 'solve' the problem by disallowing situations in which it would otherwise be needed.

❊ ❊ ❊

In EJB, a Stateful Session Bean must only be accessed by one client. When this client is finished with the Bean, it calls *remove()* and the server can then delete it. Reference counting is therefore not explicitly defined. To avoid problems with crashed clients, EJB uses a very minimal version of leases – each Session Bean has a timeout associated with it. If the Bean is not accessed for the defined period, it is removed by the CONTAINER. Invoking an operation implicitly extends the lease.

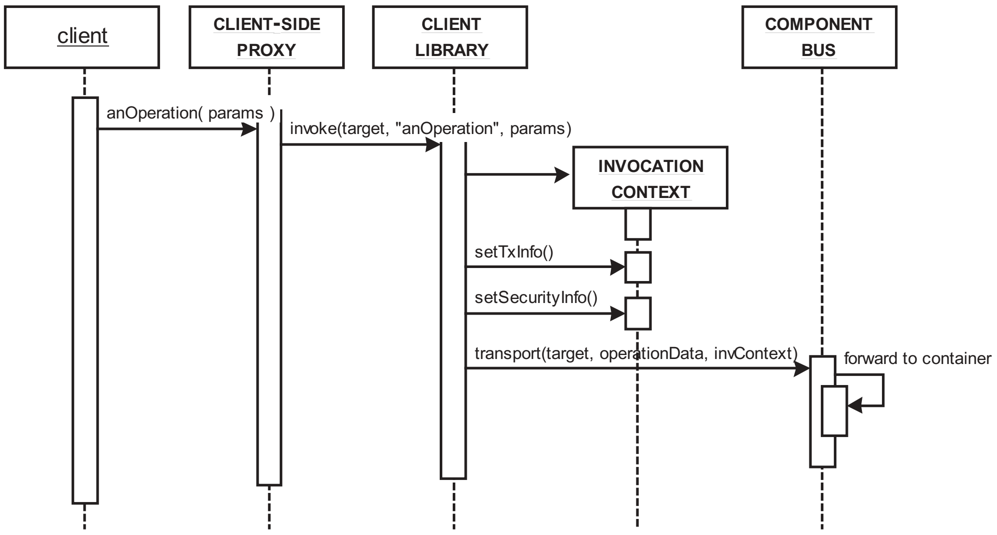
In CCM, as in CORBA, this pattern is not implemented, although Session Beans also have timeouts, as in EJB.

COM+ implements reference counting. The *IUnknown* interface, the base interface for all other interfaces, provides an operation *AddRef()* and *Release()* to increment and decrement the reference counter respectively. The client has to manage the reference count explicitly by invoking the respective operations. To facilitate this, libraries usually provide smart pointers for COM+ and COM objects that automatically take care of the reference count.

Summary

This chapter provides details about how remote clients can access the COMPONENTS inside a CONTAINER. The COMPONENT BUS can only work if clients can attach to it, so a CLIENT LIBRARY is necessary. A CLIENT-SIDE PROXY represents the COMPONENT INTERFACE in the client's process. The data for realization of technical concerns is attached to a method invocation using an INVOCATION CONTEXT. CLIENT REFERENCE COOPERATION is a pattern that will not always be used, depending on the type of COMPONENTS the architecture provides and the resource management strategy on the server.

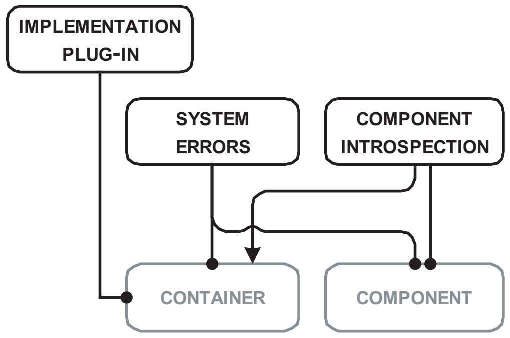
Consider a method invocation from the client to a COMPONENT:



The client invokes a 'local' operation on the CLIENT-SIDE PROXY. The proxy then forwards the call to the CLIENT LIBRARY, which creates an INVOCATION CONTEXT object. This object contains required information about security and transactions. The CLIENT LIBRARY then forwards the invocation – including target instance, operation name, parameters, and the INVOCATION CONTEXT – to the COMPONENT BUS, which is responsible for forwarding this data to the APPLICATION SERVER. On the server, the COMPONENT BUS contacts the COMPONENT PROXY to handle the request.

7 More Container Implementation

Chapter 3, *Container Implementation Basics*, provides information about how a CONTAINER is usually structured internally. This chapter goes into more detail by discussing a further three patterns.



SYSTEM ERRORS define a special class of errors (or exceptions) that denote technical problems, as opposed to application-level problems. Explicitly marking these types of errors allows the CONTAINER to take corrective action, possibly without involving the client.

COMPONENT INTROSPECTION provides a way in which a CONTAINER or its support tools can obtain information about the COMPONENT and its COMPONENT INTERFACE. This is necessary to allow tools to be provided to create or check the ANNOTATIONS of the COMPONENT.

Finally, an IMPLEMENTATION PLUG-IN is an optional part of the CONTAINER that automatically implements certain parts of the COMPONENT IMPLEMENTATION that would otherwise have to be implemented manually by the developer. A typical case for IMPLEMENTATION PLUG-INS is the implementation of persistence code.

System Errors

You are using COMPONENTS to implement your functional requirements, and you use a CONTAINER to take care of technical concerns.

❁ ❁ ❁

Clients of COMPONENTS should not have to worry about technical concerns, because they are handled by the CONTAINER. However, errors can be raised in the CONTAINER. While clients usually know how they have to react to errors resulting from application-level problems, they cannot always know how to fix problems related to technical concerns. Clients might not even know which errors can potentially be signaled by a CONTAINER.

Imagine a bank-teller COMPONENT. This might have an operation *transfer()* that transfers a given amount of money from one account to another. There are several things that can go wrong when calling this operation.

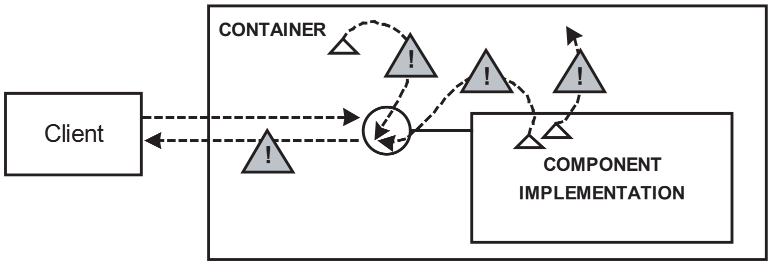
First, the balance of the account might be too low. As a consequence, the transfer is not allowed. This is a typical application-level, or functional, error that must be handled at the application level, that is, at the level at which the client calls the COMPONENT. The client can easily handle this, because the reaction to a low balance is specified in the functional requirements of the application. For example, a message could be output, requiring the operator to enter a lower amount.

However, there are other kinds of errors. The *transfer()* operation will probably involve a transactional bracket around several database operations. Debiting and crediting the respective accounts must run in one transaction. A database operation might fail, or it might not be possible to commit the transaction because of a technical problem in one of the resource managers. Alternatively, the network connection might break in the case of a remote database, or the server might be overloaded and might respond very slowly, causing a timeout.

These are completely different kinds of errors. The client cannot react to them sensibly, because there is nothing the client can do about such events except notifying the user that a technical problem has occurred on the server. As a consequence, the client should not be required to handle such errors explicitly, and, more importantly, the COMPONENT IMPLEMENTATION should not need to care about them, because they are consequences of the technical aspects implemented by the CONTAINER.

Therefore:

Define two different kinds of error: application errors and system errors. All operations on COMPONENTS can cause specific system errors at any time. System errors can also be raised by the CONTAINER without the COMPONENT IMPLEMENTATION's involvement or knowledge. Ensure that application errors are handled by the client and the CONTAINER does not have to be concerned with them. System errors should be handled by the CONTAINER and optionally forwarded to the client.



❊ ❊ ❊

Using this pattern allows the CONTAINER to react appropriately to technical errors. Only as a last resort, when there is no other possible course of action, does the CONTAINER notify the client by raising a specific error, basically signaling "I give up". For example, if a transaction cannot be committed because of a locking conflict, the CONTAINER can wait for a specific period and try again. In case of success, the client never needs to know about the problem. If the retry does not work, the CONTAINER notifies the client.

On the other hand, the COMPONENT IMPLEMENTATION can raise its own functional errors that the CONTAINER does not care about, so that clients have to deal with them. Ideally, such possible errors are specified as part of the operation signatures in the COMPONENT INTERFACE.

❊ ❊ ❊

In EJB every operation declared in the Remote Interface – *not* its implementation - has to throw the *javax.ejb.RemoteException*. This exception is the superclass of all exceptions that the CONTAINER can throw when it encounters technical problems. The client application is required to catch this exception and handle it. Note that this exception is never directly thrown by COMPONENT IMPLEMENTATIONS.

If you want to want to signal a SYSTEM ERROR explicitly, or if one of your used resources needs to throw one, you have to throw an *EJBException*. This is a subclass of *RuntimeException*, and therefore does not have to be declared in the throws-clause of the method. However, if the CONTAINER notices a thrown *EJBException* in a COMPONENT, it catches it and reacts to it, for example by aborting the current transaction. To notify the client, it then throws a *RemoteException* to the client.

EJB defines a number of additional standard exceptions, for example the *DuplicateKey* exception, which is thrown when a COMPONENT instance is created with a PRIMARY KEY that is already present.

Note that operations defined in EJB 2.0's Local Interface must not declare the *RemoteException*. This is logically consequent – a Local Interface can only be invoked by a co-located Bean. If a SYSTEM ERROR is raised in the invoked Bean, it reports this as an *EJBException*, as always. This exception 'bubbles up' to the calling Bean, is not caught there, and is eventually handled by the CONTAINER, which can then throw the usual *RemoteException* to the client.

Application exceptions are of course not subtypes of *RemoteException*, they are ordinary *Exception*-(sub)classes. They must be declared in the operations that may throw them in the Remote Interface, the Local Interface, and in the implementation. They must be caught just as usual, although the CONTAINER does not react to them, it just forwards them to the calling client. The CONTAINER therefore does no automatic transaction rollback when an application exception is thrown.

CCM also defines a set of exceptions that are thrown and handled by the CONTAINER. CCM uses the standard CORBA exceptions to signal communication and other technical problems. Exceptions for several standard cases are also available, such as *Components::CreateFailure*, *Components::FinderFailure*, and *Components::RemoveFailure.*

In COM+, every operation in an interface has a specific return type, *HResult*. This is basically an integer that serves as a uniform way of reporting errors back to the client. Some specific ranges are defined that are reserved for use by the CONTAINER. The developer is free to define his own application errors in the non-reserved ranges. The errors do not have to be checked (although of course they should). COM+ provides some operations that allow the simple check of whether, for example, an *HResult* denotes success or some kind of failure.

Component Introspection

You are using ANNOTATIONS to specify technical requirements for a COMPONENT.

❊ ❊ ❊

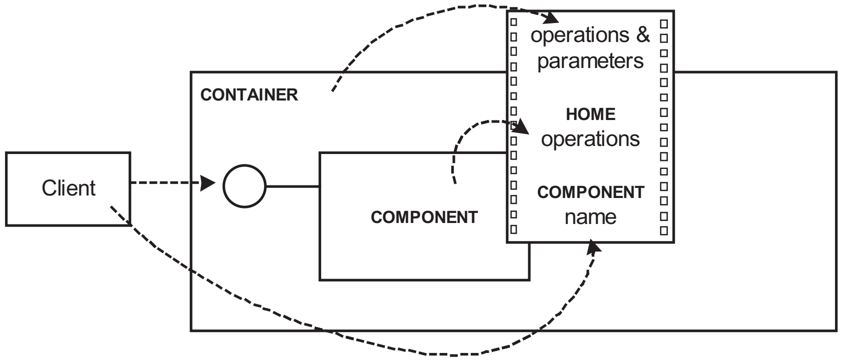
A CONTAINER has to verify the ANNOTATIONS for a COMPONENT in order to create a suitable GLUE CODE LAYER and implement the ANNOTATIONS correctly. It might also want to detect illegal combinations of certain specifications in the ANNOTATIONS. The CONTAINER therefore has to be able to learn about the features of the COMPONENT, such as the operations defined in the COMPONENT INTERFACE. You might also want to create a user-friendly tool to help create the ANNOTATIONS for a COMPONENT.

Consider transaction specifications in the ANNOTATIONS for a COMPONENT. The CONTAINER must be sure that the ANNOTATIONS contain a transaction attribute for every operation defined in the COMPONENT INTERFACE. To make this possible, the CONTAINER must be able to discover which operations the COMPONENT provides. It must also make sure that all the types used as parameters to the operations are available later at run-time.

In many cases, your CONTAINER should provide helper tools that assist users in creating the ANNOTATIONS for a COMPONENT, for example by providing suitable defaults for novice users. Such a tool should allow the user to select the transaction attribute for each operation in the COMPONENT INTERFACE. The tool therefore needs to discover the operations provided by the COMPONENT.

Therefore:

Provide information about the structure of a COMPONENT to the CONTAINER and to support tools. If the underlying language supports reflection, you can use this mechanism directly. Otherwise, you might need to generate the necessary information from the source code. In any case, make sure that this meta-information is always consistent with the actual COMPONENT.



❊ ❊ ❊

Typical information about a COMPONENT includes:

- Operations in the COMPONENT INTERFACE, their parameters and exceptions
- Operations in the COMPONENT HOME
- The name of the COMPONENT in NAMING

Java, for example, provides real reflection and the COMPONENT class or interface can be introspected its operations, et cetera, without requiring access to the source code. If reflection is not available, you can generate additional classes or add additional information to your COMPONENT classes during development using a suitable tool.

In addition, you can also make the meta-information available to the COMPONENT itself at run-time using the COMPONENT CONTEXT, thus providing reflective capabilities for COMPONENTS even if the language does not allow it directly. This meta-information about COMPONENTS might also be interesting for the client of a COMPONENT, to build suitable wrapper classes or to invoke operations dynamically. This can be useful for the implementation of IDEs or scripting languages.

See the *Reflection* pattern in [BMRSS96].

❊ ❊ ❊

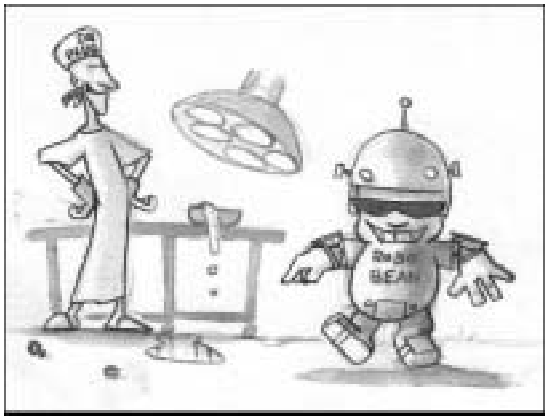
In EJB, Java's reflection API can be used for this purpose. The tools used to create the Deployment Descriptor (ANNOTATIONS) can query the interface for its operations and, if necessary, parameters. This functionality is not specific to EJB, it is a core feature of the Java language. At run-time, a Bean can obtain information about itself or other Beans by acquiring the Bean's instance of *EJBMetaData*.

CORBA relies on an *interface repository* to store its metadata about COMPONENT INTERFACES. The meta-information on an interface has to be inserted manually into the interface repository – deployment tools will automate this process. Once this is done, a client can query the interface repository about the operations of the interfaces, their parameters, et cetera. CCM uses an interface repository implementation based on the OMG MetaObject Facility (MOF).

In COM+, a COMPONENT is not really reflective, but each COMPONENT is delivered with a type library. This is a binary file that contains all the information present in the IDL file that defines a component and its interfaces. A client application or a tool can use the type library to obtain information about the COMPONENT, its GUID, the interfaces it supports, and about all operations and their parameters. OLE automation uses these features extensively.

Implementation Plug-In

You specify a COMPONENT INTERFACE and provide a COMPONENT IMPLEMENTATION. In addition, you use ANNOTATIONS to specify the way in which the CONTAINER should behave.



❊ ❊ ❊

In many projects or scenarios, certain parts of a COMPONENT IMPLEMENTATION are always implemented in a specific, more or less similar way. It is not possible to treat these aspects as technical concerns handled by the CONTAINER, because the CONTAINER does not have enough information to be able to generate the required code automatically. However, you want to automate the implementation of these parts of a COMPONENT as much as possible.

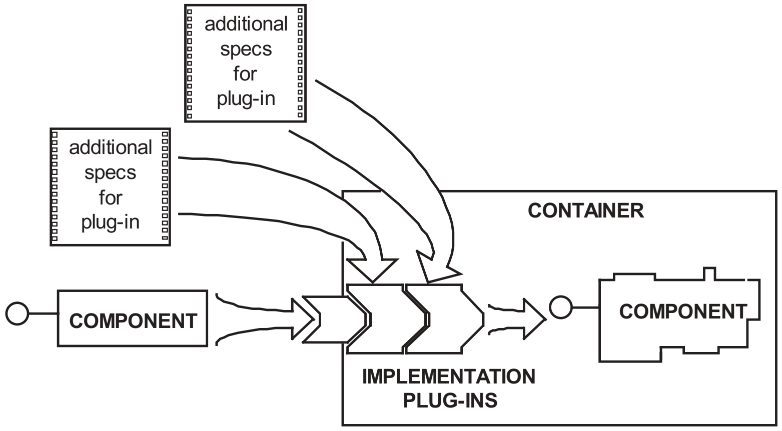
A typical example of such a situation is persistence code in ENTITY COMPONENTS. There are usually several LIFECYCLE CALLBACK operations that are used to persist a COMPONENT's state to a database or to load it from a database. Usually in a specific project there is one type of database, for example an RDBMS system, and the persistence code for COMPONENTS is always the same and typically rather simple, once you specify the mapping from attribute to database column.

It is possible for a fairly sophisticated tool to generate all the code required to access the database. However, the programmer has to write additional specifications that are not part of a COMPONENT definition or the ANNOTATIONS to make this possible. The code can then be created automatically as part of the COMPONENT development process.

The same is true for project-specific interceptors that are used to enforce additional, project-wide policies like logging, profiling or permission checking. In this situation, the reason for automatically providing the code is not to relieve the programmers from coding repetitive code. The programmers might not even know or care about such code, but others in the project team must make sure that this code is always in place, and in a specific way.

Therefore:

Provide a means to plug in code generators into your CONTAINER for certain aspects of a COMPONENT IMPLEMENTATION. Developers might have to write additional specifications to allow the tool to generate such code correctly. The code is generated as part of the GLUE CODE LAYER. Provide a standardized interface for these IMPLEMENTATION PLUG-INS.



❊ ❊ ❊

This pattern relieves your COMPONENT programmer from tedious, repetitive tasks. For example, for persistence, the user only specifies the mapping from attributes to database tables, while all the code for storing the COMPONENT's state and loading it again is generated automatically based on these specifications.

This also allows you to create additional code that 'wraps' a COMPONENT and provides functionality of which the COMPONENT developers, who might only be concerned with functional issues, are not aware.

Such plug-ins are usually executed when the COMPONENT is installed in the CONTAINER during COMPONENT INSTALLATION. The generated code is either part of the GLUE-CODE LAYER, or a modified COMPONENT IMPLEMENTATION is generated.

For CONTAINERS that should be reuseable, make sure that there is a well-defined interface for such IMPLEMENTATION PLUG-INS to allow third-party vendors to create and integrate their own specific generators.

❊ ❊ ❊

EJB provides a feature called *Container-Managed Persistence*. This implements the lifecycle operations that take care of persistence (*ejbLoad(), ejbStore(), ejbCreate(), ejbRemove()*) automatically upon COMPONENT INSTALLATION. The APPLICATION SERVER provides ways to do this by providing a plug-in interface by which persistence providers can integrate their code. Unfortunately this interface is not standardized, however, so persistence implementations cannot be exchanged seamlessly between different APPLICATION SERVERS.

CCM includes the concept of declarative persistence. An extension of IDL is used to define logical data stores and map component state to these data stores. The CONTAINER is free to implement these declarations. However, because persistence is a complex task of its own, it is expected that plug-ins will be used to handle this in the future.

In COM+, persistence is not addressed by the component model, so no plug-ins are available to directly implement it. On a lower level, (OLEDB, ADO, ODBC), the concept of pluggable drivers for database access is available.

Summary

The patterns described in this chapter are not as closely related as those in previous chapters. However, they are essential building blocks for creating a usable component architecture.

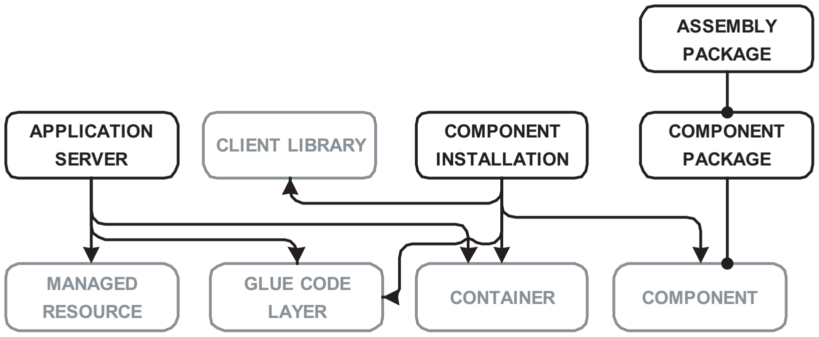
SYSTEM ERRORS are essential to make sure the right kind of error is handled by the part of the system responsible. By defining several classes of errors, this distinction is significantly simplified.

COMPONENT INTROSPECTION is essential to allow the CONTAINER to discover the features of a COMPONENT. Only then can the COMPONENT be installed correctly and the ANNOTATIONS checked for correctness.

IMPLEMENTATION PLUG-INS can help to use your architecture more efficiently, because they can be used to implement certain parts of your COMPONENTS automatically, increasing reliability and reducing the chance of errors.

8 Component Deployment

This chapter, the last chapter of Part I of the book, 'closes the circle' by discussing three patterns that deal with the deployment of COMPONENTS as part of applications.



After discussing the CONTAINER and COMPONENTS, as well as their relationships and internal structures, we need to investigate how COMPONENTS are deployed in the CONTAINER. The step at which this is done is called COMPONENT INSTALLATION. It is essential to be able to implement several other patterns successfully.

When COMPONENTS are used as building blocks for applications throughout an enterprise or in some sort of COMPONENT marketplace, you need to define a suitable packaging, a COMPONENT PACKAGE. This must contain everything necessary for COMPONENT INSTALLATION in a compatible CONTAINER.

Most cases will not involve only single COMPONENTS. Usually, you must distribute, manage or transfer complete applications that consist of many collaborating COMPONENTS. An ASSEMBLY PACKAGE can make life a lot easier here by 'packaging' complete applications. As a typical application consists of many COMPONENTS, and because each COMPONENT usually specifically generated or configured CONTAINER, you will need several CONTAINERS to run an application. An APPLICATION SERVER provides a shell for those CONTAINERS and the additional required services such as NAMING.

Component Installation

Your COMPONENTS implement the functional concerns. The CONTAINER provides a GLUE CODE LAYER to handle the technical concerns.

❊ ❊ ❊

Your ANNOTATIONS specify the technical requirements for a COMPONENT. The CONTAINER integrates them with the COMPONENT IMPLEMENTATION. However, before a COMPONENT can be made available to clients, the consistency of the ANNOTATIONS must be checked and the GLUE CODE LAYER needs to be generated. The COMPONENT HOMES also need to be registered in the NAMING service before the COMPONENT can actually be used. Other, more specific tasks might also be necessary.

Using ANNOTATIONS to specify aspects of the COMPONENT's behavior has a disadvantage – the compiler cannot verify that the ANNOTATIONS are syntactically and semantically correct and consistent with the COMPONENT IMPLEMENTATION and COMPONENT INTERFACE. To avoid run-time errors, checking is necessary. COMPONENT INTROSPECTION can help the CONTAINER here. However, you do not want to force the developer to use a special tool to create the ANNOTATIONS and verify them, because such a tool usually cannot be integrated easily into a make/build process.

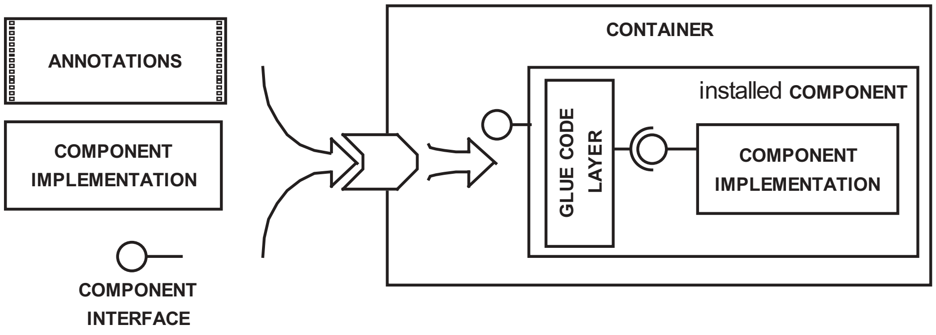
For reasons of performance and reliability, the CONTAINER does not interpret the ANNOTATIONS at run-time. It uses a layer of generated code, the GLUE CODE LAYER, to integrate the COMPONENT with the CONTAINER's management of technical concerns.

Ideally, the CONTAINER would check whether a COMPONENT sticks to the IMPLEMENTATION RESTRICTIONS required by the CONTAINER. Another problem is that the CONTAINER needs to publish the COMPONENT HOMES in its NAMING service so that they can be looked up by clients.

Last but not least, many COMPONENTS rely on a specific environment, to which the CONTAINER has to provide access. Examples include certain MANAGED RESOURCES such as database connections, a specific database schema, or files. This environment has to be set up before the COMPONENT can be made available to clients.

Therefore:

Provide a well-defined installation procedure. This procedure handles all the artifacts that make up a COMPONENT. During this procedure the CONTAINER can do everything necessary to host the COMPONENT successfully.



❊ ❊ ❊

The artifacts that make up a COMPONENT usually include:

- The COMPONENT INTERFACE and all its parameter types
- The COMPONENT IMPLEMENTATION and all the types used in it
- The ANNOTATIONS
- The COMPONENT HOME

All these are typically packaged within a COMPONENT PACKAGE. This explicit installation step allows the CONTAINER to create the GLUE CODE LAYER necessary to adapt your COMPONENT's functional requirements to the technical infrastructure provided by the CONTAINER, taking into account the declarative requirements in the ANNOTATIONS. COMPONENT INSTALLATION is also usually the time when the COMPONENT HOMES are published in the NAMING service, and the CLIENT LIBRARY is created as a side effect of GLUE CODE LAYER generation.

As mentioned in the problem discussion, a COMPONENT usually needs certain things in its environment. These include the MANAGED RESOURCES and perhaps some configuration of the database. Some of these, such as the MANAGED RESOURCES, must be set up in the CONTAINER by the CONTAINER administrator. Usually, this is done before COMPONENT INSTALLATION. Others, such as the creation of a specific schema in the associated database, can be done by the COMPONENT itself after it has been installed. You should provide a set of 'preparation operations' that do all these things. Before the COMPONENT is actually used by clients, these operations must be executed. Make sure, by setting the permissions accordingly, that they cannot be executed by arbitrary clients.

In many cases it is actually the APPLICATION SERVER that runs the COMPONENT INSTALLATION and, as part of this installation, generates or configures a CONTAINER suitable for the COMPONENT. If the CONTAINER is completely generated, the COMPONENT INSTALLATION process is handled by the APPLICATION SERVER, generating a CONTAINER on the fly.

A disadvantage of an explicit COMPONENT INSTALLATION procedure is that testing – and development in general – become more time consuming, because the installation takes up additional time and effort. You should make sure that most of the functionality is tested before your COMPONENT is installed. This also simplifies debugging significantly, because it can be done outside the CONTAINER. You can also set up your development environment such that COMPONENT installation handled automatically, so that it becomes less error-prone and easier to handle.

❊ ❊ ❊

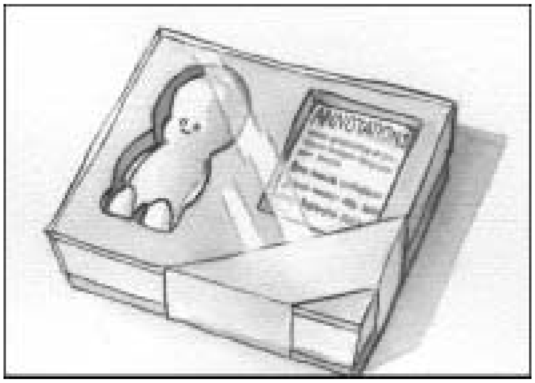
EJB requires an explicit installation step as described in this pattern, called *deployment*. All parts of the COMPONENT are packaged in a *.jar* file (a COMPONENT PACKAGE) and provided to the APPLICATION SERVER for deployment. Upon deployment the CONTAINER creates the GLUE-CODE LAYER to host the COMPONENT. It also registers the Bean with the NAMING service and carries out all other necessary configuration or installation steps. The result is an installed Bean that can be accessed by a client.

As with EJB, CCM also requires an explicit installation step. Because CCM might use languages that do not provide reflective capabilities, code generation is even more important than with Java in EJB. COMPONENT INTROSPECTION can help here.

In COM+ a COMPONENT has to be installed explicitly, which is done using the Component Services Explorer. You can define several attributes – ANNOTATIONS – for the COMPONENT using this tool. The information is stored partly in the COM+ catalog and partly in the Windows Registry. To deploy a COMPONENT to another machine without manually resetting its attributes, you can create an export file that contains all the attributes. This can be re-imported on another machine.

Component Package

You use COMPONENTS as reusable building blocks for enterprise applications. COMPONENT INSTALLATION is used to bring the COMPONENT into the CONTAINER in which it will be executed.



❊ ❊ ❊

A COMPONENT consists of several artifacts such as the COMPONENT INTERFACE, COMPONENT IMPLEMENTATION, ANNOTATIONS, and usually a set of support classes or files. All these artifacts must be available to the CONTAINER in consistent, compatible versions. This is also true when COMPONENTS are distributed in a large enterprise or via a COMPONENT marketplace.

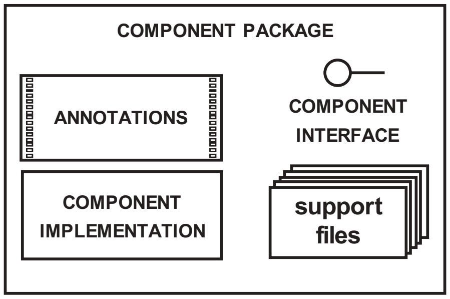
COMPONENTS are meant to be reused as building blocks for large applications. To install a COMPONENT successfully in the CONTAINER, and to make sure it runs smoothly thereafter, all the ingredients of the COMPONENT must be available in the correct versions. For example, if the COMPONENT INTERFACE operations are not implemented in the COMPONENT IMPLEMENTATION, the COMPONENT cannot be executed.

The situation becomes even worse when different COMPONENTS in a system use the same classes, but in different versions. You then need to make sure that each COMPONENT has access to the version it requires.

When a given COMPONENT is used as part of many different applications, distribution of the COMPONENT is something worth considering. Irrespective of whether it is used only in a large company or on a global COMPONENT marketplace, the COMPONENT must be successfully transported to its destination – while making sure that everything stays consistent and compatible. The users expect 'plug'n'play', or more correctly, 'plug'n'configure'n'play' operation, and this expectation must be met.

Therefore:

Define a COMPONENT PACKAGE file format and structure. This archive contains all the artifacts of a COMPONENT. The package is used for distribution and upon COMPONENT INSTALLATION. The COMPONENT is supplied to the target CONTAINER in such an archive.



❊ ❊ ❊

Using this pattern makes it impossible for part of the COMPONENT to be missing or in the wrong version. Distribution is therefore significantly simplified. If several applications run in one CONTAINER, the pattern ensures that all of them use the same, consistent COMPONENT.

Note that the COMPONENT PACKAGE file usually does not contain the GLUE CODE LAYER, because this is CONTAINER-specific and generated during COMPONENT INSTALLATION, while the COMPONENT PACKAGE is CONTAINER-independent.

One of the common archive file formats can typically be used to serve as a COMPONENT PACKAGE. *Zip*-format files are often used in practice. To make sure all the necessarily artifacts are available in such a file and in the correct directories, a special packaging tool is usually included with the CONTAINER. This makes sure that all necessary artifacts are available when the COMPONENT PACKAGE file is created.

If the programming language used does not provide reflective features, the creation of the COMPONENT PACKAGE is also a good time at which explicit COMPONENT INTROSPECTION data can be gathered and packed into the COMPONENT PACKAGE, for later use by tools or the CONTAINER.

If the COMPONENTS are to be distributed on their own, the COMPONENT PACKAGE file should also contain human-readable information about the features of the COMPONENT. Machine-readable metadata is also useful to allow automatic categorization of COMPONENTS in a marketplace.

❊ ❊ ❊

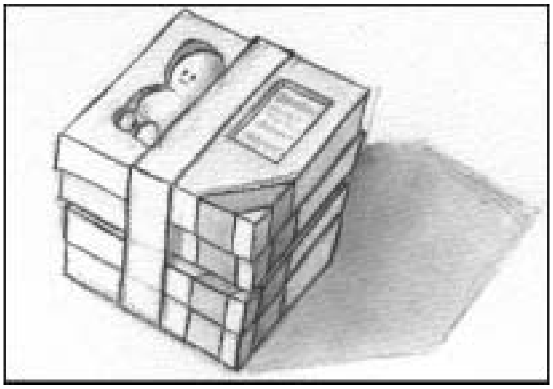
In EJB, a COMPONENT is packaged into a specific *.jar* file. The file contains the Remote Interface, the Bean class, the Home Interface, the PRIMARY KEY class, if required, the Deployment Descriptor, and all other required classes or resources such as icons, et cetera, are also contained. *jar* files are *.zip* files in a specific defined format that contains a *manifest*, a contents list for the file. *jar* files can be created using the standard Java JAR tool or other special tools provided by the APPLICATION SERVER vendor.

CCM defines the general notation of a software *package*. A software package always consists of a set of files and a descriptor in an XML format that defines the meaning of the files. The package is stored in a *.zip* file with a specific suffix. In the case of a packaged COMPONENT, the files are stored in a file with a 'CAR' suffix. For COMPONENTS, the descriptor contains a Component Descriptor, as described in ANNOTATIONS. The COMPONENT Archive file (CAR file) contains basically the same artifacts as in EJB.

In COM+, a configured application can be exported in a file that contains all the COMPONENTS, the type libraries and the necessary client proxies. The information in the file can be imported into another computer's Component Services to form a new application. This differs from the workflow described above, because you first need to install the COMPONENT to create a COMPONENT PACKAGE.

Assembly Package

You use COMPONENTS to separate your business logic into reusable entities. Now you want to reassemble an application from these COMPONENTS.



❊ ❊ ❊

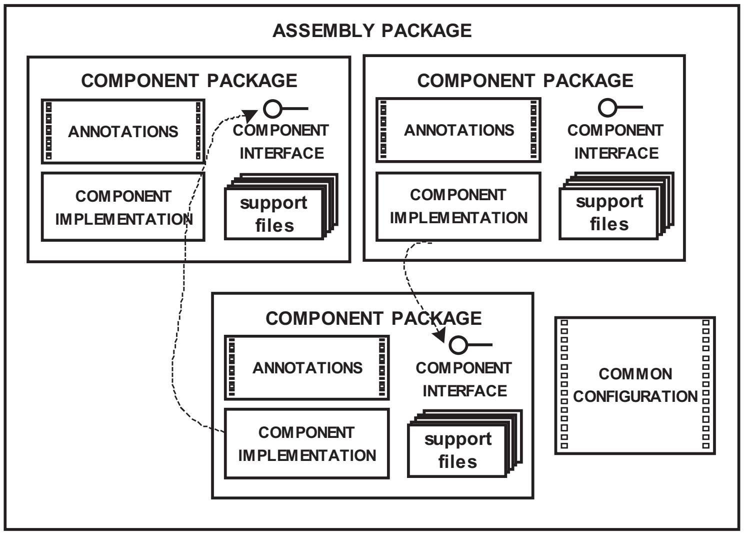
A COMPONENT application on the server consists of a set of collaborating COMPONENTS. It might become difficult to control which COMPONENTS belong to which application, because the 'application' is really invisible – it is just a set of COMPONENTS. This makes it difficult to distribute an application and hard to ensure that the application is assembled from the correct COMPONENTS with the correct versions.

COMPONENTS consist of a set of artifacts. They must all be available in correct and compatible versions. This is why you provide a COMPONENT PACKAGE to group all the necessary artifacts of a COMPONENT. The same is true one level higher, on the level of applications. An application consists of a set of collaborating COMPONENTS, so it is equally important that the COMPONENTS that make up the application are consistent and compatible.

There is usually some common configuration for a set of COMPONENTS in an application. For example, security and role information are usually specified on an application-level basis. Repeating all the configuration information for each COMPONENT, for example in its COMPONENT PACKAGE, is tedious and error-prone.

Therefore:

Provide an ASSEMBLY PACKAGE that contains all the COMPONENT PACKAGES that make up an application. Specify the relationships between the COMPONENTS of the application and include common configuration data.



❊ ❊ ❊

Using this pattern allows you to manage the applications in the APPLICATION SERVER more efficiently. Dependencies between COMPONENTS become explicit, especially if the REQUIRED INTERFACES pattern is also used. It is clear which resources are used by which 'application'. Note that this does not compromise loose coupling. COMPONENTS can still be replaced individually as long as the COMPONENT INTERFACES remain stable. Also, a specific COMPONENT can be part of multiple applications.

❊ ❊ ❊

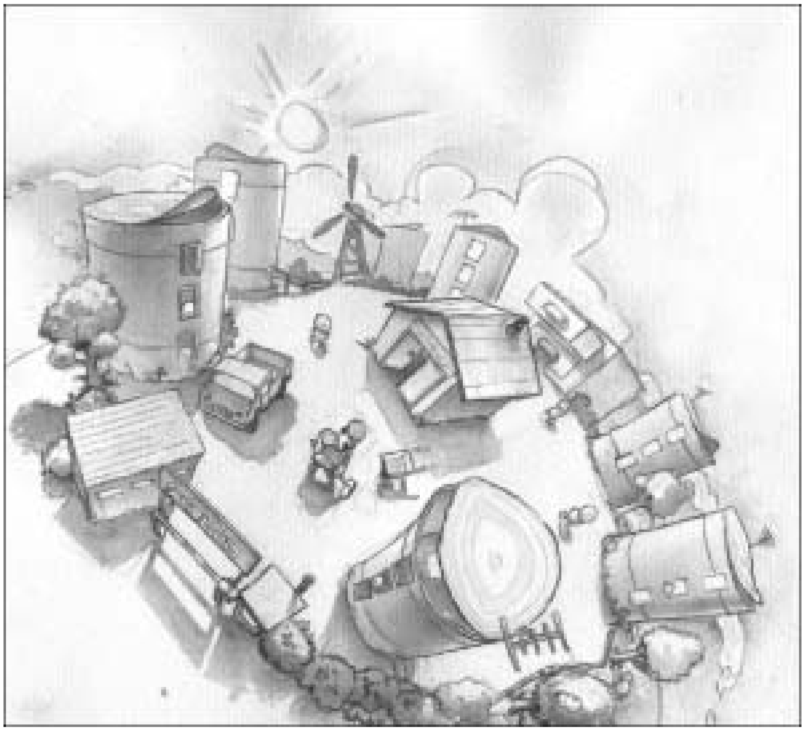
EJB, or more precisely, J2EE provides *.ear* files. These are basically *.jar* files that contain several COMPONENT-*jar*s and other resources required for a complete J2EE application. For example, an *.ear* file might hold web archives – '*.war*'s – packages that contain JSPs, Servlets and their configuration information.

CCM supports the concept of an *assembly*. Several COMPONENT PACKAGES can be grouped together into an assembly archive file (.AAR). In addition to the COMPONENT PACKAGES, the file contains an assembly descriptor that is also an XML document. This describes which COMPONENTS make up the assembly, how they are partitioned, and how they are connected to each other. Note that CCM provides REQUIRED INTERFACES as part of the CIDL definition of a COMPONENT.

COM+ supports the concept of an application running as part of Component Services on a Windows machine. A COM+ application consists of all the COMPONENTS used to form the application, together with the roles of users. User management is integrated with the user management of the computer or the domain by using Windows security. Complete applications can be exported to move them to another machine (see COMPONENT PACKAGE).

Application Server

Different kinds of COMPONENTS (SERVICE, SESSION and ENTITY) are usually used together in a software system – most applications use all of these types. You usually have one dedicated CONTAINER instance per COMPONENT, customized to host the COMPONENT using a GLUE CODE LAYER specific to the COMPONENT.



❊ ❊ ❊

As you have a separate CONTAINER for each COMPONENT type, you usually need to provide a single run-time environment in which all the CONTAINERS run. The different CONTAINERS usually use the same set of MANAGED RESOURCES and other services such as NAMING. You do not want to configure and maintain all these things on a per-CONTAINER basis. Moreover, a complete application

usually consists of additional parts, not just COMPONENTS, which also have to run somewhere.

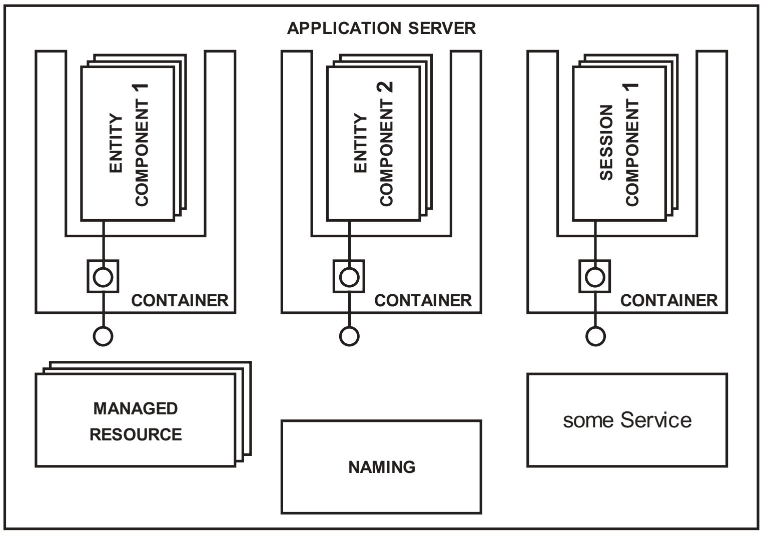
An application consists of a set of collaborating COMPONENTS. For reasons of efficiency, these usually run in the same process if no clustered is used to enhance scalability. However, they run in different CONTAINERS, because a CONTAINER contains a GLUE-CODE LAYER specific to the hosted COMPONENT.

Consider a typical MANAGED RESOURCE, a set of database connections. Such a connection will usually be used by several COMPONENTS in an application. While the CONTAINER allows the COMPONENTS to access the MANAGED RESOURCES, their management, configuration and maintenance can be done in one place for all the CONTAINERS.

Additional services might also exist that are part of an application system but cannot be implemented using COMPONENTS, for example a web server-based presentation layer. These services should also run in the same process as the CONTAINERS for performance reasons.

Therefore:

Provide an APPLICATION SERVER that serves as a shell for the different CONTAINERS, the MANAGED RESOURCES and additional, non-COMPONENT services. All these services should run in one process. An APPLICATION SERVER usually provides a GUI-based configuration and monitoring tool.



❊ ❊ ❊

An APPLICATION SERVER provides a homogeneous execution environment for COMPONENTS and other parts of an application. Application servers can usually be clustered to achieve fail-over and load balancing.

❊ ❊ ❊

EJB CONTAINERS run in a J2EE-compliant APPLICATION SERVER. In addition to hosting CONTAINERS, APPLICATION SERVERS can for example run Servlets for web-based presentation, or be used to manage JMS message queues. An APPLICATION SERVER also runs a JNDI-based NAMING service. APPLICATION SERVERS are available from a variety of vendors. Quite sophisticated implementations are available, for example providing clustering features.

CCM will also use an APPLICATION SERVER for running COMPONENTS. It is not yet clear yet whether commercial servers will contain additional features such as web server functionality, but it is clear that they will contain a CORBA NAMING service, as well as implementations of other CORBA services such as Notification and Security.

In COM+, the Windows 2000 operating system plays the role of the APPLICATION SERVER. Additional services such as transaction coordination or a limited form of NAMING are also provided by specific parts of the operating system, the Distributed Transaction Coordinator (DTC) and the Windows Registry.

Summary

This chapter completes the description of COMPONENT architectures.

An APPLICATION SERVER hosts all the things that are needed for a COMPONENT-based application. This comprises the CONTAINERS, the NAMING service, the MANAGED RESOURCES, and more. The COMPONENTS to be used must be installed in the APPLICATION SERVER, which needs to put the respective CONTAINERS in place and generate the GLUE-CODE LAYER. This is done during COMPONENT INSTALLATION.

To make COMPONENTS, and collections of COMPONENTS that form an application, more easily deployable, COMPONENTS are packaged into a COMPONENT PACKAGE. Such a package contains everything the COMPONENT needs to run on a compatible APPLICATION SERVER. An ASSEMBLY PACKAGE contains the COMPONENT PACKAGES of related COMPONENTS that must be deployed as a group.

Part II
The Patterns Illustrated with EJB

This part of the book illustrates the patterns from Part I with examples using the Enterprise JavaBeans technology. For each chapter in Part I, this part contains a matching chapter with the corresponding EJB examples. It is not intended as an EJB tutorial, but shows how the patterns can be implemented and helps to improve understanding of them.

Readers who already have a basic knowledge of EJB should find this part especially useful, because it relates their knowledge about EJB to the more abstract concepts presented in the patterns.

Whenever we cannot stay with Sun's EJB standard in general and have to choose a specific APPLICATION SERVER, we use Sun's J2EE Reference Implementation [SUNRI] or jBoss [JBOSS]. The latter is especially practical because it is open source, allowing some insight into the CONTAINER implementation.

9 EJB Core Infrastructure Elements

The patterns from Chapter 1, *Core Infrastructure Elements*, namely COMPONENT, CONTAINER, SERVICE COMPONENT, ENTITY COMPONENT and SESSION COMPONENT, are the basis of any component-based infrastructure. This is therefore also true for EJB. In contrast to other component infrastructures, EJB supports all three component types directly.

Component

In EJB you cannot use a COMPONENT *per se*, you must instead use a particular kind of component, one of:

- SERVICE COMPONENT
- ENTITY COMPONENT
- SESSION COMPONENT

The COMPONENT pattern defines the basics of what a COMPONENT is and how we structure our applications using COMPONENTS.

In EJB, a COMPONENT is made up of several distinct artifacts:

- The *Remote Interface* (COMPONENT INTERFACE)
- The *Home Interface* (COMPONENT HOME)
- The *implementation* (COMPONENT IMPLEMENTATION)
- The *Deployment Descriptor* (ANNOTATIONS)

EJB 2.0 additionally allows local versions of the Remote and Home Interfaces for purposes of performance optimization. Each of these artifacts and their realization in EJB is described in its own example.

COMPONENTS are packed together in a *.jar* file, technically identical to a *.zip* file. This file forms the COMPONENT PACKAGE and is used for deployment to an APPLICATION SERVER. Deployment is the EJB term for COMPONENT INSTALLATION. In EJB, deploying a COMPONENT to an APPLICATION SERVER creates the CLIENT LIBRARY 'on the fly' and also creates the CONTAINER's GLUE-CODE LAYER that implements the ANNOTATIONS.

Container

In EJB, the CONTAINERS are a part of the APPLICATION SERVER, which provides a complete J2EE environment. A J2EE-compliant APPLICATION SERVER supports the following technologies in addition to EJB's[1]:

- Servlets – a framework for creating applications that run as part of a web server and handle HTTP requests
- Java Server Pages – a server-side scripting technology based on Servlets
- JMS – a framework for accessing message-oriented middleware platforms
- JNDI – an interface to naming and directory services
- JDBC – an API for accessing relational databases
- RMI/RMI-IIOP – Java's Remote Method Invocation, optionally using CORBA's IIOP protocol for transport
- JavaIDL – Java's interface for CORBA programming
- JTA and JTS – APIs for managing distributed transactions
- JAXP – A set of APIs for working with XML
- JCA – The Java Connector Architecture, used to integrate enterprise information systems (legacy systems) into J2EE applications – see PLUGGABLE RESOURCES
- JavaMail – An API for sending and retrieving e-mail using SMTP, POP3 and IMAP protocols

A CONTAINER might also come as a separate piece of software that provides special features, such as integration with a specific database. For example, Versant provides a CONTAINER that is integrated with its EnJin OO database. Unfortunately, the interface between the

1. Some of these technologies are not specific to J2EE, they are also available in the Java 2 Standard Edition (J2SE). Nevertheless, they are available in J2EE, because J2EE is a superset of J2SE.

APPLICATION SERVER and the CONTAINERS has not been standardized. So CONTAINERS are thus always specific to a particular APPLICATION SERVER.

As described in the CONTAINER pattern, the CONTAINER is a run-time environment for COMPONENTS. It is responsible for implementing the technical concerns of an application, while the COMPONENTS implement the functional concerns. The CONTAINER therefore gives clients the illusion of integrated technical and functional concerns.

In EJB, the CONTAINER actually uses a mixture of frameworks and code generation to implement the technical concerns and to provide the seamless integration with the functional concerns in the COMPONENT. We will see how this is done in more detail throughout the following chapters.

The next section lists the (technical) services provided by the CONTAINER and the surrounding APPLICATION SERVER, as far as they are relevant to the discussion here.

Managing resources

As described in the MANAGED RESOURCE pattern, the CONTAINER manages all resources used by the COMPONENTS. This is necessary to integrate them with the management of technical concerns such as security and transactions, as well as for the purpose of connection pooling.

Persistence

There are two fundamentally different ways to use persistence in EJB: using Entity Beans, or accessing persistent data from a Session Bean. As persistence is a technical concern, the CONTAINER offers some help for implementing persistence.

Making an Entity Bean persistent

EJB supports two ways to make Entity Beans persistent: Bean-Managed Persistence (BMP) and Container-Managed Persistence (CMP).

Using BMP, it is the responsibility of the COMPONENT programmer to provide all the code that handles persistence. This code must be placed in specific LIFECYCLE CALLBACK operations inside the Bean implementation. The CONTAINER calls these operations whenever it is necessary to manage persistence. To access the database, the LIFECYCLE CALLBACK operations use connections that are MANAGED RESOURCES, to benefit from pooling and security/transaction integration.

In the CMP case the Bean developer simply leaves the LIFECYCLE CALLBACK operations empty. The CONTAINER provides specific IMPLEMENTATION PLUG-INS that generate the code to handle persistence.

Accessing persistent data from Session Beans

The CONTAINER does not explicitly provide support here, except for providing MANAGED RESOURCES that should be used by COMPONENTS to access the database. Be sure to use these resources correctly. Pooled resources can be used without the creation overhead that is common to non-pooled resources. The examples of the MANAGED RESOURCE pattern illustrate this.

Security

EJB CONTAINERS provide a means for declarative security in addition to programmatic permission checking. This means that the COMPONENT programmer can specify in the ANNOTATIONS which logical roles are allowed to invoke specific operations.

Upon deployment, these logical roles are then mapped to concrete user groups or users that are configured inside the APPLICATION SERVER. The CONTAINER makes sure that these security policies are enforced at run time. Access to MANAGED RESOURCES can also be controlled using this feature. In addition, authentication is also provided, and some APPLICATION SERVERS also provide (proprietary) encryption for wire communication.

Transactions

The APPLICATION SERVER acts as a transaction coordinator for all transactional MANAGED RESOURCES, and the COMPONENT instances involved. The COMPONENT developer only needs to specify transaction attributes for the operations in the ANNOTATIONS.

At run time, the CONTAINER initiates, commits and rolls back the transactions, as necessary. This feature is called Container-Managed Transactions (CMT). It also usually coordinates a two-phase commit, if more than one XA[2] resource is involved in the transaction. Note, however, that this is not a J2EE requirement, although most APPLICATION SERVERS support it.

It is also possible to use Bean-Managed Transactions. In this case the COMPONENT has to access the transaction object to manage the transaction state by itself. This is not allowed for Entity Beans from EJB 1.1 onwards, however – they have to use CMT.

The transaction object is available to the instance by calling *getUserTransaction()* on the COMPONENT CONTEXT. COMPONENT instances can use the transaction object to *begin()* a new transaction, or to *commit()* or *rollback()* the current transaction. The state of the current transaction can be determined by calling the *getStatus()* method.

Other features

Additional features are necessary to make up an APPLICATION SERVER, but these are not standardized, because they are not important for the portability of the COMPONENTS. Among others, they include:

- Set-up and configuration of MANAGED RESOURCES
- Management of users and user groups
- Tools to facilitate deployment, the creation of the ANNOTATIONS, and set-up of Container-Managed Persistence

2. XA is a standard transaction protocol.

Service Component

In EJB, SERVICE COMPONENTS are called Stateless Session Beans.

They have no 'memory', hence they are stateless – they have no knowledge of state changes made during previous invocations. Each invocation starts with a new, 'clean' instance[3]:

- The CONTAINER uses INSTANCE POOLING to implement VIRTUAL INSTANCES
- Several LIFECYCLE CALLBACK operations are mandatory – for examples, see LIFECYCLE CALLBACK

From a bird's eye logical view, Stateless Session Beans *are* stateless. However, this does not mean that they cannot have attributes that can keep values. The best definition of *stateless* in this context is probably that they do not *remember* a change of state from one method invocation to the next, even if the two operations are invoked by the same client. The reason for this behavior is that the CONTAINER uses INSTANCE POOLING to manage the instances. For each incoming request, the CONTAINER takes an arbitrary instance from the pool and uses it to handle the client request.

Consider the following scenario:

- A client looks up the COMPONENT HOME of a Stateless Session Bean
- It creates a new instance of the Bean
- It invokes an operation
- It then invokes a second operation

The second operation may be handled by a different physical instance in the CONTAINER. This instance does, of course, not know of any changes made to the instance used to handle the first operation. It is also possible that the second operation is handled by the same

3. This is not the whole truth, but is a helpful illustration of a Stateless Session Bean. More detail is given later.

instance, but it is not guaranteed and you must not rely on it. As a consequence, Stateless Session Beans must provide operations that complete a logical 'unit of work' during a single invocation.

Stateless Session Beans can nevertheless carry state. For example, you can keep a reference to the COMPONENT HOME of another Bean used in the implementation of the Stateless Session Bean. This state is never changed, is identical for all instances, and it is not part of the business logic. Other examples for this kind of state include the *InitialContext*, MANAGED RESOURCE factories and other technical data. Attributes of Stateless Session Beans are often used to keep references to resources that should be cached within a Bean instance for improved performance.

Note that there is nothing to keep you from holding client-specific state inside a Session Bean. If you do this, however, you must be sure to mark the Session Bean as *stateful*. Stateful and Stateless Session Beans are technically identical – the CONTAINER cannot easily distinguish them. Therefore you have to tell the CONTAINER in the ANNOTATION how you want it to handle the Bean.

The following must be true when creating a Stateless Session Bean:

- The Stateless Session Bean must implement the *javax.ejb.SessionBean* interface
- The HOME INTERFACE is only allowed to include one *create()* method without parameters
- In the ANNOTATIONS the Session Bean is marked as 'Stateless Session Bean'

When to use Stateless Session Beans and how

Use Stateless Session Beans whenever there is no clear reason to use another type of Bean. They are the least resource-consuming Beans available. As long as you can tolerate their limitations of statelessness, use them.

Typically, they are used to implement short business processes, to provide a wrapper for legacy system's APIs or as facades for more complicated subsystems [VSW02]. The latter also improves performance, because the number of network hops can be reduced.

Message-Driven Beans

With EJB 2.0 a new Bean type has been introduced, the Message-Driven Beans (MDB). What are they and why do we introduce them in this pattern?

In the COMPONENT BUS pattern, we described how the bus should be able to use synchronous or asynchronous transport to access COMPONENTS. This is not the case in EJB. Here, method invocations on 'normal' Beans are always synchronous.

However, EJB provides another means to access Beans asynchronously – they can be used to handle messages that are posted to a JMS queue or topic. JMS queues and topics always transport their data asynchronously. You cannot use a normal Bean to listen to such JMS queues or topics, though, a special Bean type has to be used: the Message-Driven Bean. MDBs are simply JMS message consumers. The Bean has no identity for the client – it is not even visible – and it is stateless, just as Stateless Session Beans.

So a Message-Driven Bean can be seen as a SERVICE COMPONENT that is invoked asynchronously. But a Message-Driven Bean has two interesting differences compared to other Bean types:

- It does not have a COMPONENT INTERFACE[4]
- It does not have a COMPONENT HOME

4. An MDB can receive messages, so one could therefore say that it has at least a simple interface. However, a client can only send messages to a JMS destination, not to the COMPONENT. The COMPONENT is just a receiver of these messages. So the client of the COMPONENT does not get a COMPONENT INTERFACE – it just has to deal with JMS.

A Message-Driven Bean does not have a COMPONENT INTERFACE, because there is simply no need for it. Clients never access MDBs directly, they send a message to a JMS queue or topic and these messages are then handled by an MDB. They could just as well be handled by any other JMS-compliant message listener.

This is also the reason why a COMPONENT HOME is not needed: the client does not need to obtain a Bean instance explicitly, as the CONTAINER creates a pool of instances that are used to listen for messages. Of course, the server needs a means of delivering the messages to the instance. This is why the Bean has to implement the *javax.ejb.Message-DrivenBean* and the *javax.jms.MessageListener*[5] interfaces. These two interfaces define the LIFECYCLE CALLBACKS needed for a MDB.

When to use Message-Driven Beans and how

Message-Driven Beans should be used whenever an asynchronous invocation of a Bean is necessary – that is, when no immediate response is required. They provide a way to integrate JMS with EJB.

As with Stateless Session Beans, a common use for MDBs is the *Facade* pattern. The MDBs are used as facade for a complex business process and may access many other Beans to implement the business logic. Again, the advantage is that all business logic is on the server, which is not only good for performance reasons but also for supporting different kinds of clients.

5. The EJB 2.0 specification says the additional *MessageListener* interface is necessary to define the MDB to use JMS. In future versions of EJB a Message-Driven Bean might not only be able to support JMS, but also other message-oriented middleware (MoM). An example might be the Java API for XML Messaging (JAXM). A Message-Driven Bean needs to be able to implement other interfaces that match the requirements of the other MoMs, for example the interfaces defined in *javax.xml.messaging*.

Entity Component

In EJB, Entity Beans are the implementation of the ENTITY COMPONENT pattern. They are used primarily to represent business entities. Entity Beans have the following features:

- Entity Beans can be accessed by many clients concurrently. The CONTAINER makes sure concurrent access is synchronized.
- Logical Entity Beans instances have a unique identity, specified by a PRIMARY KEY.
- The state of a logical Entity Bean instance is persistent.
- They exist until they are explicitly removed.
- The CONTAINER uses INSTANCE POOLING to implement VIRTUAL INSTANCES.
- Several LIFECYCLE CALLBACK operations are mandatory – see the LIFECYCLE CALLBACK pattern for examples.

What is an Entity Bean?

The usual belief is that Entity Beans are a 'componentized' representation of a row in a relational database table, in part because they are identified by a PRIMARY KEY. However, this is wrong.

In many cases, a table in a relational database (and thus, its rows) is the result of a *normalization* process[6]. This means that the business entity is spread over many tables, which will then be joined again in queries. So, an Entity Bean should represent the whole business entity. The database joins should be done internally in the Beans *ejbLoad()/ejbStore() operations.*

An Entity Bean also is not a typical, small object. If you change a normal, local Java program to be 'distributed' by making each object

6. Normalization is used to prevent duplicate data storage. This basically means that an entity's representation in the database is distributed over several rows in different tables.

an Entity Bean, performance will suffer badly. Each access to the Bean will be remote, slowing down the system. Each call to an Entity Bean operation also involves a significant overhead, because security, concurrency, and transactions have to be handled. So, an Entity Bean is relatively 'heavyweight' compared to a normal Java object[7].

In the case in which the caller and the callee are located in the same JVM, APPLICATION SERVERS typically provide some optimizations. In EJB 2.0, this is explicitly handled by using the Local Interfaces in such a case. This allows for finer-grained use of Entity Beans. However, there is still some overhead involved that stems from the CONTAINER's management of technical concerns.

So what *is* an Entity Bean? An Entity Bean represents a logical, coarse-grained business entity. Internally it may be structured, for example using Dependent Objects [VSW02], normal Java objects that are used to structure a Bean internally. The Entity Bean's database representation can be spread over many tables if a relational database is used.

The technical perspective

The COMPONENT IMPLEMENTATION of an Entity Bean must implement the *javax.ejb.EntityBean* interface. This interface is different from the *javax.ejb.SessionBean* interface, because of the different LIFECYCLE CALLBACK operations required for Entity Beans: persistence has to be managed as part of its lifecycle.

The COMPONENT INTERFACE is not different from other Beans. However, the COMPONENT HOME might contain *find...()* methods that are used to re-find instances based on specific criteria. A *findByPrimaryKey()* operation is obligatory, which finds an instance based on its PRIMARY KEY. The PRIMARY KEY class has to be specified in the ANNOTATIONS, and of course the PRIMARY KEY class must be programmed if no built-in class – such as *String* – is used. *create(...)* operations with different signatures are also possible.

7. Several patterns can help to overcome these problems: *Bulk Setter*, *Value Object*, see [VSW02].

Container-Managed Persistence in EJB 2.0

Version 2.0 of EJB completely redefined Container-Managed Persistence (CMP). Nevertheless, the underlying principles as described in the patterns remain the same. The following paragraphs provide a brief overview of CMP 2.0 but do not delve into details of how to use it.

The first thing that has been changed is the way in which persistent attributes are declared in Entity Beans. In former CMP versions, CMP attributes had to be public members of the Bean implementation class. Additionally, in the Deployment Descriptor these attributes had to be specified for persistence in the database.

In Version 2.0 of CMP, the fields must not be defined in the Bean implementation class at all. Instead, *accessor* methods must be specified for every CMP field. These *getter* and *setter* methods have to be public and abstract, because the CONTAINER implements them later. Their names follow the usual JavaBeans naming conventions, as shown in the code below:

```
public abstract class PersonEJB implements EntityBean {
  public abstract String getName();
  public abstract void setName(String name);
  // ...
}
```

In this example the setter and getter for the property *name* are defined. Note that the attribute itself must not be defined. The idea behind this concept is to give the CONTAINER more freedom in implementing the attributes and their mapping to the database.

Typically, the CONTAINER will generate a non-abstract subclass of the Bean implementation class, implementing the abstract operations in a suitable way. For example, this allows for ‘lazy-loading’ of attributes, which is especially important for container-managed relations (see the next section). Also, the CONTAINER is now able to include some kind of ‘dirty’ flag to reduce store operations if the Bean’s properties have not been changed. Note that because of the abstract accessor methods, the Bean implementation class has to be declared abstract too.

While each persistent attribute has to provide a public setter and getter operation, not all of these accessors have to be published through the COMPONENT INTERFACE. You can define read-only attributes for clients, for example, by only providing the getter in the COMPONENT INTERFACE.

All CMP-attributes have to be either a Java primitive or a Java *Serializable* type. As in older versions of EJB, the CMP fields have to be declared in the Deployment Descriptor. The relevant part of the Deployment Descriptor looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-
jar_2_0.dtd'>

<ejb-jar>
 <display-name>PersonBean</display-name>
 <enterprise-beans>
  <entity>
   ...
   <cmp-field>
    <field-name>name</field-name>
    <description>The name of the person</description>
   </cmp-field>
   ...
```

Also as in older versions, the APPLICATION SERVERS provide proprietary tools for creating this Deployment Descriptor, including the CMP parts.

Container-managed relationships

EJB 2.0 not only changed the style of handling persistent attributes, it also introduced the concept of container-managed relationships (CMRs). In previous versions of CMP the handling of relationships between Bean instances was entirely up to the Bean developer. From Version 2.0 onwards, the CONTAINER helps to manage them by introducing so-called 'CMR fields'. Basically these are CMP fields that contain references to other Beans. When they are accessed, the referenced Entity Bean is automatically retrieved and accessible. Note that this is quite simple for the CONTAINER, as it can override the abstract getter operation in a way that retrieves the referenced instance by using the COMPONENT HOME, the PRIMARY KEY and the *findBy...()* operations.

Relationships may come in the following, well-known flavors: one-to-one, one-to-many and many-to-many. The navigability of the relationships may be unidirectional or bidirectional. Note that Beans can only be referenced via their Local Interface to make sure the relationships have acceptable performance – see the COMPONENT INTERFACE example. If the referenced Entity Bean has only a Remote Interface, it cannot be used as a partner in a container-managed relationship.

To use one-to-many or many-to-many relationships, the CMR fields have to be of type *java.util.Collection* or *java.util.Set*. These collections store references to Local Interfaces of other Beans. For a one-to-one relationship, the Local Interface of the related Bean has to be stored in a CMR field.

The accessor methods for CMR fields must not be exposed through the COMPONENT INTERFACE of the Entity Bean. The Local Interface types of the related Entity Beans must also not be exposed through the Remote Interface. The collections themselves must not be exposed through the Remote Interface. No external client should be allowed the means to access a Local Interface, as this would not work.

As the CONTAINER now knows about the relationships between the Beans, it can offer additional services. For example, it can help if an instance in a relationship should be removed – the *cascade-delete* option in the Deployment Descriptor can be set to instruct the CONTAINER and also delete the referenced Entity Beans.

Select operations

Another new feature introduced in EJB 2.0 are *select* methods. These have the same basic purpose as finder operations, namely to search and return a set of instances. Unlike finder methods, however, select methods are not defined in the Bean's Home Interface, they are just abstract methods defined in the Bean's implementation class, and must not be exposed through the Remote Interface. Their purpose is to provide internal find capabilities for use by the Bean, using the CMP features of the CONTAINER. Also in contrast to finder methods, select methods may return not only *EJBObjects* or *EJBLocalObjects*, but also CMP fields, or collections of them.

The actual queries for finder and select operations are not specified using a database-dependent query language like SQL or OQL – instead, EJB QL has been introduced. The purpose of EJB QL is to allow the specification of APPLICATION SERVER and database-independent queries. This makes porting the persistence layer to a different APPLICATION SERVER or database significantly simpler. EJB QL itself should look familiar to those who know SQL or OQL – queries consist of a *select*, a *from* and a *where* part. For the complete specification of EJB QL in Backus-Naur form (BNF), see [SUNEJB].

The following are a few examples to illustrate what EJB QL looks like:

```
SELECT OBJECT(p) FROM Person p
```

The statement above finds all instances of *Person* Beans.

```
SELECT DISTINCT OBJECT(p) FROM asn AS p WHERE p.name='alex'
```

Find the *Person* named *alex* from the abstract schema called *asn*.

The EJB QL queries for the select and finder operations are defined in the Deployment Descriptor:

```
...
<query>
  <description></description>
  <query-method>
   <method-name>findByFirstName</method-name>
   <method-params>
    <method-param>java.lang.String</method-param>
   </method-params>
  </query-method>
  <ejb-ql>SELECT DISTINCT OBJECT(p) FROM asn AS p WHERE p.name=
?1</ejb-ql>
</query>
```

When to use Entity Beans and how

In theory, Entity Beans should always be used to represent business entities, as described above. However, because of the overhead involved in accessing these kinds of Beans, there are situations in which Entity Beans should not be used, or when you need to help the CONTAINER to use them efficiently.

The following paragraphs give a brief overview of when and when not to use Entity Beans. Using Entity Beans efficiently is possibly the most complex issue in EJB, so this overview cannot provide a full coverage of the topic. With each new version of the EJB specification, and with each new generation of APPLICATION SERVERS, the issues described should become less and less serious.

Access to an Entity Bean is always synchronized. If an Entity Bean is accessed, the CONTAINER has to take an instance from the pool, load the data from the database and forward the request to the instance[8]. This is a significant overhead for a simple getter operation, for example. In cases in which read-only access by a large number of clients is required – for example looking up a stock price in a stock trading application – this overhead is too big. Using a TYPE MANAGER[VSW02], a Stateless Session Bean that accesses the database directly, is more efficient here.

More optimization is possible in the *ejbLoad()/ejbStore()* operations: these are called at the beginning and end of a transaction respectively. If a Bean has a large amount of persistent state – which it usually has, because it is coarse-grained – a lot of data will be loaded or stored for each call. If much of this data is not needed by the called operation, a lot of unnecessary work has to be done by the CONTAINER and the database. Use the *Lazy State* and *Deferred Store* patterns [VSW02] [VSW02] to optimize this. Some APPLICATION SERVERS also do this automatically – consult your server's documentation for details.

8. This is a little simplified and a worst-case scenario. Actually it only has to be done once within a transaction, if the isolation flags are set correctly and the container does not synchronize the state of the Bean more often than necessary.

If you need to read or update *the same group of attributes* on many Entity Bean instances, you should not access each of them in turn. Accessing a large group of Entity Beans requires the above-mentioned overhead for every call to every instance. The resources expended on technical concerns could thus greatly outweigh the business effort. You should use *Data Transfer Objects*[9] and *Bulk Setters/Entity Bulk Modification* patterns [VSW02] [VSW02] instead, which define operations that do all the work at once.

Entity Beans can also be used to represent long processes, as described in *Process As Entity Bean* [VSW02]. When a process lasts longer than one method invocation and needs to be persistent, or has more than one participant, then you should probably use an Entity Bean to model the process.

9. The DATA TRANSFER OBJECT pattern is also known as VALUE OBJECT in some EJB literature. However, because the name VALUE OBJECT has been used for a different pattern in the past (which describes something completely different to a DATA TRANSFER OBJECT), the pattern community has asked for EJB's VALUE OBJECT to be renamed to DATA TRANSFER OBJECT, to avoid confusion.

Session Component

Stateful Session Beans are the implementation of SESSION COMPONENTS in EJB. They also have some features worth mentioning:

- They remember state changes from one method invocation to the next within a specific client session
- They have no identity
- No synchronization is implemented, because they must be used by only one client at a time
- Their state is not persistent
- They live until they are removed by a client or until a timeout occurs
- The CONTAINER uses PASSIVATION to keep their resource usage low
- Several LIFECYCLE CALLBACK operations are mandatory – see the example for LIFECYCLE CALLBACK

In the EJB programming model, Stateful Session Beans are the extension of a client's state on the server. In practice, Stateful Session Beans can cause some problems worth mentioning.

Stateful Session Beans have one important characteristic – as their name implies, they are able to hold (conversational) state. However, their state is not persistent. Because it is not persistent, and because they have no identity, instances cannot easily be re-found by clients other than those that created them. An instance is thus always assigned to a specific client. The state of such a Bean usually carries log-in information, user preferences, or data needed for small workflows[10].

10. A popular example here used to be a 'shopping cart' on an e-commerce web site. This example is no longer valid, however, because such shopping carts are persistent on modern e-commerce web sites, and are therefore candidates for Entity Beans.

Every client may have one corresponding Stateful Session Bean instantiated for its whole lifetime, or even longer. This quickly leads to scalability issues and potentially slow applications. Imagine a Stateful Session Bean that uses 10 Kbytes of memory. Not that much, perhaps, but multiplied by 10,000 more or less concurrent users, you then need 100 Mbyte of memory on your server for a single Bean type. The 'more or less concurrent' results from the fact that a lot of applications don't enforce correct and timely log-outs for users, resulting in many more active sessions than is really necessary. In fact, web applications don't have a chance to enforce a clean user log-out – users may simply close their browser window. So most of the Stateful Session Beans instances will remain in existence until their timeout allows the CONTAINER to remove them.

A Bean timeout is necessary because this is the only chance the CONTAINER has to get rid of unused Stateful Session Bean instances. On the other hand, defining a value for such a timeout is critical. To keep resource usage as low as possible, it seems a good idea to define a very short timeout, so that the CONTAINER can clean up as soon as possible. However, this may result in users experiencing timeouts while using the application. As the bare minimum, client code needs to be resistant to 'lost' Stateful Session Bean instances, and of course timeout values have to be long enough to avoid upsetting users.

It is fairly obvious that this is mostly a 'lose–lose' scenario. Help is on the way, however – PASSIVATION. PASSIVATION allows the CONTAINER to store a Stateful Session Bean instance in a persistent store when available memory is getting low and the instance is not being accessed. This is intended to ensure that the CONTAINER never runs into an out-of-memory situation due to too many active Stateful Session Bean instances. When an invocation on a passivated instance subsequently arrives, the instance is reactivated.

While PASSIVATION is certainly useful and 'comfortable' for developers, it is nevertheless relatively slow compared to in-memory access, and should therefore be avoided when possible. To achieve maximum performance, make sure that your APPLICATION SERVER machine has enough physical memory to keep all instances in memory all the time.

Stateful Session Beans are meant to be used by only one client at a time. The CONTAINER does not enforce this, because no synchronization is used on the instances. This is good, because it improves performance, but it also means that you as a developer have to make sure that concurrent access to an instance really does not happen. In practice this means that usually, when you pass a reference to an instance to another client (for example, using a HANDLE) it should really use 'move' semantics and not 'copy'– the client that passes the reference should then forget about the reference and not use it any more.

To make a COMPONENT a Stateful Session Bean, the COMPONENT IMPLEMENTATION must implement the *javax.ejb.SessionBean* interface. Because the CONTAINER cannot distinguish Stateful and Stateless Session Beans directly, you also have to mark the Session Bean as being stateful in the ANNOTATIONS.

When to use Stateful Session Beans and how

If used wrongly, Stateful Session Beans can be a 'scalability killer', especially if the APPLICATION SERVER needs to use PASSIVATION heavily. Thus, the first thing you have to do when using Stateful Session Beans is to make sure that the server machine has enough memory to keep all instances in memory all the time, or that clients use the system in such a way that PASSIVATION does not happen too often. You can use load tests to ensure this.

Use Stateful Session Beans really only to represent short-lasting, client-specific processes on the server, be sure to *destroy()* the instance when you have finished using it, and use reasonably short timeouts.

Remember also that the state of a Stateful Session Bean can be lost due to a timeout or a server crash. Therefore use Stateful Session Beans with care.

10 EJB Component Implementation Building Blocks

The previous examples described what a COMPONENT is and what type of COMPONENTS exist in EJB. The CONTAINER, the run-time environment for the COMPONENTS, has also been introduced.

This chapter shows in more detail how a COMPONENT is structured, illustrating the relevant parts of a COMPONENT and what is needed to allow interaction between the CONTAINER and the COMPONENT instances it hosts.

Component Interface

The only way to access a Bean is through its Remote or Local Interface. These consist of a set of operations that define the syntax for how the Bean can be used. In EJB there is no formal way to specify the semantics of the operations in the COMPONENT INTERFACE. Thus only the syntactic part of the contract between a Bean and its client is formally specified and can be enforced. The Bean's implementation is responsible for providing the functional implementation for the operations in the Bean's interface.

The base interface of Remote Interfaces

The Remote Interface of all EJBs must extend a common base interface, called *javax.ejb.EJBObject*. This interface declares a couple of important operations, which can be invoked by the client on a reference to any kind of EJB, independent of whether it is an Entity or Session Bean:

- *getEJBHome()* returns a reference to the COMPONENT HOME that created this particular instance of the Bean. If a client works with instances created by different homes, the operation returns the one that actually created the instance. This can happen, for example, as a possible consequence of a load-balancing strategy of the APPLICATION SERVER.
- *getHandle()* returns a HANDLE to the COMPONENT instance. For more details on handles, see the HANDLE pattern or its EJB example.
- *getPrimaryKey()* returns the PRIMARY KEY object for the instance. This works only for Entity Beans, because only Entity Beans have PRIMARY KEYS. If *getPrimaryKey()* is called on a different Bean type, a *RemoteException* is thrown.
- *isIdentical(EJBObject)* determines whether the current instance is identical to another. Note that *equals()* cannot be used to compare two server-side COMPONENTS, because it only compares the two client-side proxies. One client-side proxy can be used for several

instances, and vice-versa. On the other hand, one COMPONENT instance can be represented by several proxies. If proxies are identical, it therefore does not imply anything regarding equality of the COMPONENT instances.

Note also that *isIdentical()* is based on the logical identity of an instance. This means that two physically different Entity Bean instances that represent the same logical entity are considered to be identical. All Stateless Session Beans of the same type are considered equal, as they are indistinguishable by definition.

- *remove()* removes the specific Bean instance. Note that *remove()* has different semantics depending on the kind of Bean. Removing an Entity Bean removes its logical entity from the database – an act that has functional business relevance, that is, is part of the business logic. Removing a Stateful Session Bean is more technical in nature, as it just removes the Stateful Session Bean instance from the CONTAINER's memory. Removing a Stateless Session Bean has no meaning at all, as it still remains in the pool.

Note that these operations cannot be provided as part of the COMPONENT CONTEXT, because they are intended to be used by the *client*, which can be a standard Java application, and no CONTAINER is available on the client's side. Note also that you don't need to implement these operations manually, because they are purely technical in nature – their implementation is part of the GLUE-CODE LAYER. The only exception is *remove()* for Entity Beans, because this operation actually implements business logic.

Business operations in the Remote Interface

The business operations you define in the Remote Interface for your functional code are normal Java interface operations, although with some minor twists.

Firstly, all operations have to throw the *java.rmi.RemoteException*. Whenever a technical exception in the server is raised, the CLIENT-SIDE PROXY raises a *RemoteException* – see SYSTEM ERRORS. As a consequence, clients have to catch the *RemoteException* whenever they communicate with EJBs through their Remote Interface.

This also has another consequence: you cannot easily use a *normal* Java interface type as a Remote Interface, because the *Remote-Exception* is not declared in its operations. Reusing the same interface type for purely local objects as well as remote COMPONENTS is therefore not possible. So, it is not completely transparent to a client that it is talking to a remote instance. This is necessary, of course, because remoteness enables types of errors to occur that could not occur in a purely local environment, and the client programmer must write code to handle these errors. Such errors include network failure, server crashes, or unavailable Bean instances.

Using an interface type of another EJB as a parameter to such an operation always has *pass-by-reference* semantics. This means that the receiving COMPONENT creates a new proxy for the same remote Bean instance. In contrast, primitive Java types can be used as usual – their values are copied with *pass-by-value* semantics.

The most interesting case is for Java objects, whose types are not EJB Remote Interfaces and do not implement *java.rmi.Remote*. Although in a normal, *non-distributed* Java environment these objects would be passed by reference, in the case of EJBs they are passed by value – they are serialized and copied to the receiver of the operation[1]. This has several consequences:

- An object that is unique in a purely local application can now be available on several machines with different object identities! In other words, due to the copying, the objects represent the same data but have different identities. Changes to one are not reflected in the other copies.
- Even if the object has its object identity as an attribute, it is possible for the object to be passed to several clients, and upon return, several objects with different identities but with the same identity attribute can come into existence in one process. You have to be aware of this and make sure it will not result in hard-to-find bugs.

1. This is the normal RMI behavior.

- When an object 'returns' from its journey to a client, it has a different identity – a different object reference. *equals()*[2] has to be overridden correctly in order to compare object identity based on suitable attributes, such as the above-mentioned object identifier attribute.

As a consequence, all non-remote and non-primitive types used in EJB interface operations must be *Serializable.* More precisely, they must be a valid RMI/IIOP value type. To avoid problems, it is a good idea to only use immutable objects – objects that cannot change their externally-visible state once they have been created. Strings are an example of such a type.

Local interfaces

Local Interfaces were introduced with EJB 2.0. Before EJB 2.0, the CONTAINER handled an invocation to a Bean in the same way irrespective of whether it was invoked from a remote or a local client – the CONTAINER always handled it as if it were a remote call. This is very inefficient if a Bean was called by another Bean in the same CONTAINER, as in this case the overhead from the remote invocation is unnecessary. A distinction between local and remote method invocations is therefore necessary from a performance point of view[3].

This is what Local Interfaces are designed for. An EJB 2.0 Bean can offer a Local Interfaces in addition to Remote Interfaces. This Local Interface is only accessible inside the same CONTAINER as the Bean[4]. The interface therefore does not need to support remote calls and invocations through it can be optimized. This has a few implications:

- All arguments and return values are passed by reference. The Bean provider has to allow for this. One danger is that the state

2. Therefore *hashCode()* also has to be overridden.
3. EJB 1.1 APPLICATION SERVERS already optimized method invocations on collocated Bean instances. However, there was usually still some overhead. For Local Interfaces the APPLICATION SERVER can be *sure* that it can only be local – more efficient optimizations are thus possible.
4. Note that inside a cluster multiple JVMs might form a CONTAINER. Therefore it might be necessary to migrate the state of a Stateful Session Bean from JVM to JVM or to have multiple Entity Bean instances representing the same logical entity.

of one Bean might be assigned to the state of another. Remember that the CONTAINER manages the Beans and their state – building up links between Beans makes this management impossible. Arguments and return values must therefore be cloned before they are attached to the state of a Bean.

- The usual *narrow()* operation on the Home Interface is not needed, simple casts are enough. Narrow operations are required because of the IIOP compatibility of the COMPONENT BUS. This is not needed with Local Interfaces, no IIOP is part of the invocation, it is purely local.
- As a consequence of the more lightweight, local access, Beans now allow for fine-grained access without too much impact on performance.

It is possible for a Bean to support both local and Remote Interfaces.

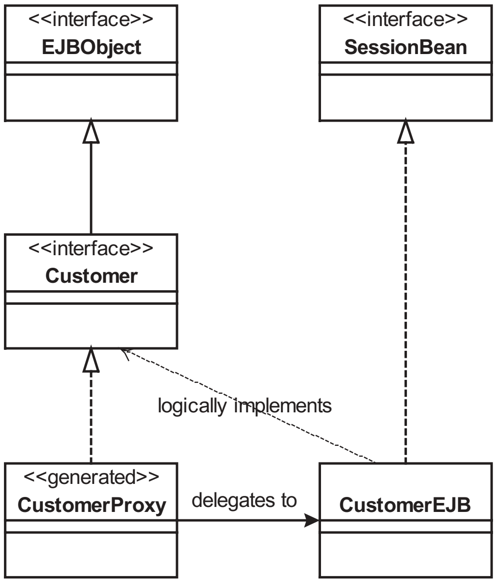
To create a local variant of a Bean, the Bean's interface needs to implement *javax.ejb.EJBLocalObject* instead of *javax.ejb.EJBObject*. A local Home Interface is also necessary: this extends *javax.ejb.EJBLocalHome,* instead of *javax.ejb.EJBLocalHome*. Note also that operations in the Local Interfaces must not throw a *RemoteException*.

Implementing the interfaces

The COMPONENT IMPLEMENTATION has to provide the implementations for the operations defined in its Local and Remote Interface. However, the COMPONENT IMPLEMENTATION does not really implement the interfaces in a Java sense. Note that no client should ever receive a reference to the implementation class, as the client only accesses the COMPONENT PROXY. Only the CONTAINER, specifically the COMPONENT PROXY inside the GLUE-CODE LAYER, accesses the implementation object. As the GLUE-CODE LAYER is generated anyway, there is no need to use a specific interface. Moreover, if the implementation class cannot be accessed by clients – because it does not implement the Local or Remote Interface – its reference cannot be passed to the client accidentally. For details and rationales on this, see VIRTUAL INSTANCE and the illustration on the following page[5].

Another reason for not directly implementing the Local and Remote Interface is that the technical part of the interface – those methods in *javax.ejb.EJBObject* and *javax.ejb.EJBLocalObject* – will not be implemented by the COMPONENT developer. All of these methods cover technical issues. A basic principle for COMPONENTS is SEPARATION OF CONCERNS, and thus these issues must not be the responsibility of the component developer.

We will look at this in more detail for the Remote Interface. The next diagram shows the relationships between the respective classes for the Remote Interface.



The generated COMPONENT PROXY implements[6] the Bean's Remote Interface and delegates to the COMPONENT IMPLEMENTATION, and of course the CLIENT-SIDE PROXY also implements it. The implementation instance is never accessible by the client, only managed by the

5. With some APPLICATION SERVERS it is possible to allow the implementation class to really *implement* the Remote Interface. However, this is not recommended, for the reasons explained above.
6. Depending on the implementation of the COMPONENT BUS, the COMPONENT PROXY need not implement the Remote Interface, because it is accessed generically by the COMPONENT BUS. However, the CLIENT-SIDE PROXY at least has to implement the interface – there is no way around that.

CONTAINER, which is why it must implement the *EntityBean* interface – it's the CONTAINER's handle on the instance.

The correct way to create an implementation for a Bean is therefore as follows:

```
// Remote Interface
public interface Customer extends javax.ejb.EJBObject {
 public String getName() throws java.rmi.RemoteException;
}

// implementation,
public class CustomerEJB implements javax.ejb.EntityBean {
 public String getName() {
  // some useful code here...
 }

 // More operations go here. Not shown for brevity.

}
```

This approach has another benefit – it must never be possible to pass the Bean implementation object to the client directly, because only VIRTUAL INSTANCES exist. Imagine an operation like the following:

```
public interface Customer extends javax.ejb.EJBObject {
 public Customer getMe() throws RemoteException;
}
```

If the Bean implementation were a subtype of the interface, the implementation could simply return *this*, a reference to the Bean implementation object. Using two different inheritance graphs, this is no longer possible. There would also be another problem: as *Customer* extends *EJBObject*, you would have to implement the operations declared in *EJBObject*, which you cannot do, because it is the CONTAINER's job.

The correct implementation of this operation uses the Bean's COMPONENT CONTEXT to access the server-side proxy, which is returned:

```
public class CustomerEJB implements javax.ejb.EntityBean {

 private EntityContext entityContext;

 public void setEntityContext( EntityContext ctx ) {
  entityContext = ctx;
 }

 public Customer getMe() {
  // return this; would be wrong and is not possible
  // because of Java typing rules
  return (Customer)entityContext.getEJBObject();
 }
}
```

This has a disadvantage in the development cycle – your compiler cannot detect if your interface supports operations or signatures other than your implementation class. You will only notice this during deployment, which will slow down development. To get around these problems, use the BUSINESS INTERFACE pattern [VSW02].

Several interfaces for one Component

A Bean always has at most one Remote and one Local Interface. But there is a simple way to have several statically-joined interfaces:

```
public interface Person extends javax.ejb.EJBObject {
 String getName() throws RemoteException;
}

public interface BusinessEntity extends javax.ejb.EJBObject {
 Order[] getOrders() throws RemoteException;
}

public interface Customer extends Person, BusinessEntity {
}
```

Here we define separate interfaces (*Person, BusinessEntity*) and in a second step, a new Remote Interface that extends both of these base interfaces, making a *Customer* a *Person* and a *BusinessEntity*. Of course the same is possible with Local Interfaces as well.

This has several advantages. A COMPONENT user can now either use *Person* or *BusinessEntity* as its COMPONENT INTERFACE, or *Customer* as a combination of the two. You can also use the base interfaces for other combinations, such as making an *Order* also some kind of *BusinessEntity*. This approach also enhances the modularity of the code, because you can define separate interfaces for related operations and then use interface inheritance to 'combine' the COMPONENT INTERFACE.[7]

In addition to the above-mentioned benefits, you also reduce client dependencies. A client can use any of the interfaces from which the COMPONENT INTERFACE was defined:

```
Object ref=... // JNDI Lookup
CustomerHome ch=(CustomerHome)
 PortableRemoteObject.narrow(ref,CustomerHome.class);
Person p = ch.create();
```

Note that you must either know the most specific Home Interface (*CustomerHome* in this example) or must just deploy the Bean with different Home Interfaces. Home Interfaces cannot inherit from one another like Remote Interfaces, because the return types of *create()* are different – they are the respective Remote Interfaces. Overwriting a method with a different return type is not allowed in Java.

Note that this combination of interfaces is static. If it is necessary to adapt the interfaces of a Bean dynamically, see the EXTENSION INTERFACE pattern in [VSW02].

Message-Driven Beans

The exception to the rule that 'every COMPONENT has a COMPONENT INTERFACE' is the Message-Driven Bean, introduced with EJB 2.0. A Message-Driven Bean is an asynchronous message consumer, attached to a JMS queue or topic. This means that a client publishes a

7. On some APPLICATION SERVERS this might not work as described. Some require the Remote Interface to literally implement *EJBObject*. An indirect implementation through *Interface1* or *Interface2* is not sufficient to make the deployment tool work correctly. Other servers require all remote operations to be literally declared in the Remote Interface, not in 'superinterfaces' of the Remote Interface. Again, this is necessary to make the deployment tool work correctly.

message to a queue or a topic and the message-oriented middleware takes care of the message's delivery. Finally the CONTAINER calls a specific LIFECYCLE CALLBACK operation on the Bean implementation, namely *onMessage(),* to deliver the message.

So the only way for a client to invoke an operation on an MDB is to send a message to the queue or topic to which the Bean is attached. There is therefore no need for a COMPONENT INTERFACE, as no-one can invoke business operations on an MDB directly using a classic remote method call, as is possible with other Bean types.

Component Implementation

In EJB COMPONENT IMPLEMENTATION, the Bean class does not *implement* the COMPONENT INTERFACE directly, as mentioned above. The implementation must support all the operations declared in the interface. The consistency of these two artifacts is checked during COMPONENT INSTALLATION.

A Bean class must have the following properties:

- It has to implement the base interface for the respective Bean kind: Entity Beans have to implement *javax.ejb.EntityBean,* Session Beans have to implement *javax.ejb.SessionBean* and Message-Driven Beans have to implement *javax.ejb.MessageDrivenBean*. These interfaces declare the LIFECYCLE CALLBACK operations that the Bean implementation has to implement correctly.
- All business operations declared in the Remote Interface have to be implemented with the same signature, with the exception that the implementation does not have to throw a *RemoteException*. This is necessary because the generated COMPONENT PROXY will delegate the business operations to the implementation. For a Message-Driven Bean this is of course different, because it provides only *onMessage()* instead of individual methods. This method is defined in the *javax.jms.MessageListener* interface.
- All *create()*, *find()*, and *remove()* operations declared in the Home Interface have to be implemented. The names of the implementing methods are prefixed with *ejb*, for example *ejbCreate()* for the *create()* method. For each *create()* operation, an *ejbPost-Create()* operation with the corresponding signature has to be provided. The generated proxy expects these operations to be there, so they have to be implemented correctly.
- The Bean class has to provide attributes to keep the state, if using Bean-Managed Persistence (BMP) or EJB 1.1's Container-Managed Persistence (CMP). In the case of CMP as defined in EJB 2.0, you just have to provide abstract *getter* and *setter*

methods for each attribute. An attribute to store a reference to the COMPONENT CONTEXT object should also be provided.

The following is an example of an Entity Bean. First, the class has to implement the *EntityBean* Interface:

```
// imports
public class AdressBean implements javax.ejb.EntityBean {
```

In this example we are implementing an Entity Bean with Bean-Managed Persistence. So the state of the Bean has to be declared. At the minimum, it consists of the COMPONENT CONTEXT and the PRIMARY KEY:

```
  private EntityContext entityContext;
  private String id;
```

In our case, members for the attributes of the Bean are also provided:

```
  private String city;
  private String zip;
  // more state...
```

Note this would be different for CMP. In the case of CMP as defined in EJB 1.1, you have to make these attributes *public*. In the case of EJB 2.0, for each attribute an abstract getter and setter method have to be declared. These getter and setter methods are implemented as a part of the GLUE CODE LAYER.

The next set of operations consists of those enforced by the implemented base interface – the LIFECYCLE CALLBACK operations. See the example of the LIFECYCLE CALLBACK pattern to learn what exactly has to be done in these operations.

```
  public void ejbActivate() {
   // ...
  }
  public void ejbPassivate() {
   // ...
  }
  public void ejbLoad() {
   // database code omitted here ...
  }
  public void ejbStore() {
   // database code omitted here ...
  }
  public void ejbRemove() throws RemoveException {
   // ...
  }
  public void setEntityContext(EntityContext entityContext) {
   this.entityContext = entityContext;
   // ...
  }
  public void unsetEntityContext() {
   entityContext = null;
   // ...
  }
```

The next set of operations consists of those defined in the Home Interface. This follows the following rules[8]:

- For each *create(params)* operation in the interface, you need to implement a corresponding *ejbCreate(params)* operation and an *ejbPostCreate(params)*.
- For each *findByXXX(params)* operation, an *ejbFindByXXX(params)* is required:

```
public String ejbCreate( String _id ) throws CreateException {
  id = _id;
  // ...
  return id;
}
```

8. Note that this is simply a naming convention. The generated proxy delegates to the implementation object and expects the respective operations to be named as explained.

```
public void ejbPostCreate( String _id ) throws CreateException {
  // ...
}
public String ejbFindByPrimaryKey(String primaryKey) throws
 RemoteException, FinderException {
  // ...
}
```

- Last but not least, the operations declared in the Remote Interface have to be implemented according to the functional requirements:

```
 public String getID() {
  return id;
 }
 public void setCity( String _city ) {
  city = _city;
 }
 public String getCity() {
  return city;
 }

 // more...
}
```

Note that these methods would be implemented by the GLUE CODE LAYER in the case of CMP 2.0.

When implementing Beans, be sure to look at the IMPLEMENTATION RESTRICTIONS, as not everything is allowed within Bean implementation classes.

Implementation Restrictions

The EJB programming model limits the freedom of the Bean developer in several ways. Although this might seem overly constraining to start with, the limitations are necessary to guarantee portability and consistent semantics, as well as correct operation of the CONTAINER. The following is a list of the programming restrictions and the reasons for them.

Limitations

This subsection introduces the limitations in EJB, while *Checking these restrictions* on page 271 explains when and how these limitations are enforced by the CONTAINER or APPLICATION SERVER.

Static fields

A Bean is not allowed to use read/write static fields. This is because different instances of the Bean might be distributed over different JVMs or even different computers, invalidating the semantics of the *static* keyword. Using static for constants nevertheless works. This resembles the semantics usually implemented using *static final* members. Remember that you might act on different static instances, because of your potentially clustered APPLICATION SERVER, with several different JVMs each having its own set of static variables. So something like:

static final start = new Date();

can actually lead to non-homogenous values within your application. Also remember that even in a non-clustered environment the APPLICATION SERVER might use different class loaders, which again might result in inconsistent static fields.

Thread synchronization

Use of thread synchronization primitives is not allowed, because they could interfere with the CONTAINER's synchronization and pooling strategy. This is especially problematic in the case of replication of instances over several JVMs, which is possible in a clustered environment.

AWT/GUI features

APPLICATION SERVERS obviously do not normally support direct GUI interaction. They might be machines without a GUI at all, or might be located in an unmanned server room. EJBs are server-side COMPONENTS, so GUI access would not be useful.

Input/output

Using *java.io* for access to the file system is not allowed – a Bean is not allowed to read or write a file directly. The reason for this is that files are not considered to be a suitable way of storing data in an enterprise system. In a clustered environment you cannot ensure that the accessed files are available on each cluster node, and if they are, you would either have to replicate them or coordinate shared access.

This limitation is not completely enforced by some APPLICATION SERVERS – they provide files as MANAGED RESOURCES, which means that file access is managed and coordinated by the CONTAINER. It then becomes the responsibility of the administrator to make sure that the files are available and in the correct version on all nodes of a cluster.

Sockets

A Bean can use client sockets to connect to socket servers, but it is not allowed to use server sockets themselves. This is mainly because blocking calls on server sockets could conflict with the CONTAINER's threading/pooling strategies and is quite problematic in a server cluster.

Reflection

Java Reflection cannot be used to obtain information about a class that is not otherwise available, such as private fields or operations. This is important to ensure that security and integrity in the CONTAINER are not compromised.

Threading

Threading is a responsibility of the CONTAINER. To make sure its threading policies are not corrupted, a Bean must not start, stop, suspend or resume a thread. Changing a thread's priority or its name is also not allowed. Again, this is because threading is something that the CONTAINER manages, and COMPONENTS should not interfere with it.

Native libraries

Loading a native library is not allowed. The reason is obvious – portability would be compromised, and it would create a security hole. A native library might also crash and therefore compromise the stability of the APPLICATION SERVER.

Security

For security reasons, the Bean must not:

- Change the security policy information for a specific code base.
- Change or set a security manager.
- Change or set the class loader. The Bean must also not define a class. Defining a class means converting a byte array to an instance of *Class*. This is part of the classloading functionality and thus prohibited.
- Access or modify the security configuration objects (*Policy, Security, Provider, Signer* and *Identity*).

Miscellaneous

- Stopping the JVM is not allowed (obviously).
- Standard streams (*System.in*, *System.out*, *System.err*) must not be redirected, because the APPLICATION SERVER might expect them to behave in a specific way.
- The *ServerSocketFactory* and the *StreamHandlerFactory* must not be modified, because the I/O subsystem might be APPLICATION SERVER-specific.

- A Bean must not subclass or substitute features of the Java Serialization protocol. Again, this is because the APPLICATION SERVER might rely on it functioning in a specific way.
- A Bean must not pass *this* as an argument or result, because the implementation object must not be accessed by clients.

Checking these restrictions

Some of these restrictions can easily be checked at run time using Java's security provisions. Other restriction can be checked during COMPONENT INSTALLATION, to some degree, by performing byte code analysis. Nevertheless, it is not guaranteed that the CONTAINER checks some or all of these restrictions.

A COMPONENT that is well behaved has to follow all the rules described – only then can the level of portability and stability promised by EJB be realized.

Lifecycle Callback

To understand the LIFECYCLE CALLBACK operations of the different kinds of Bean, we must make sure that the lifecycle of Bean instances itself is well understood. We will try to achieve both together. As the lifecycles for the different Bean kinds are different, we will look at each in turn. Please also refer to the examples for PASSIVATION and INSTANCE POOLING, because these concepts are closely related to the Bean lifecycle.

Note that it is not possible to state explicitly when some of the LIFECYCLE CALLBACK operations are called. The specification only requires the particular LIFECYCLE CALLBACK to have some specified behavior, but it does not define exactly when the CONTAINER has to call them. The details are left open, to allow CONTAINER implementers some freedom to optimize and tune the system.

An example of such a situation is the invocation of the *ejbStore()* operation, which is explained below in the Entity Bean lifecycle. This is usually called before a transaction that involved the respective instance is committed. However, if the CONTAINER knows that the in-memory state of the instance did not change, it can decide not to call the operation – there would be no visible effect in the database, anyway. As this is not conformant with the standard, such behavior is usually optional in a CONTAINER and can be turned off.

Stateless Session Beans

Lifecycle Operations

A Stateless Session Bean must implement the following operations:

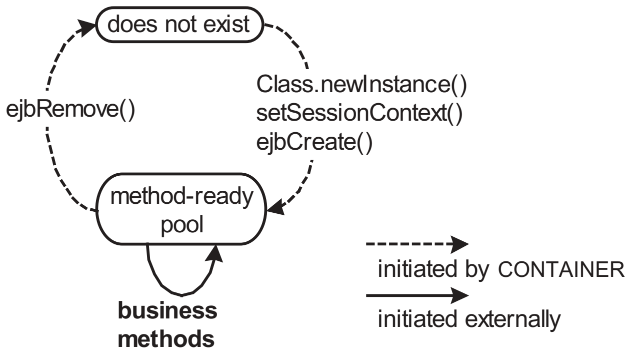
- One parameterless *ejbCreate()* operation.
- The *setSessionContext(SessionContext ctx)* operation.
- An *ejbActivate()* and an *ejbPassivate()* operation, which are in fact never called. These are included, unfortunately, to ensure that

Stateless and Stateful Session Beans have the same implementation interface.

- An *ejbRemove()* operation, which is called when the CONTAINER wants to get rid of the physical instance when shrinking the pool.

Lifecycle

Stateless Session Beans are stateless, so it is irrelevant which instance of a specific Bean is used for a client's request – all are equal. Thus the lifecycle of a Stateless Session Bean is very simple, and INSTANCE POOLING can easily be used by the CONTAINER:



The CONTAINER creates a set of physical Bean instances at start-up, calling *setSessionContext()* and *ejbCreate()*. These operations are implemented by the Bean developer and initialize the instance (see below). Note that the creation of a new instance is not related to a call to *create()* on the Home Interface. Such a call just results in a reference to a VIRTUAL INSTANCE that is given to the client.

After creation, the instance is put into a pool and is ready to receive business method invocations from clients. Whenever a client calls *create()* on the Home Interface, a reference to one of the Beans in the pool is returned to the requesting client. The client can then call a business method on the reference it received. When the CONTAINER wants to shrink the pool, it calls *ejbRemove()* on the Bean instance.

What goes where

ejbCreate() and *ejbRemove()* are called only once during the lifecycle of a physical Bean instance and serve as a kind of constructor/

destructor pair. Resource factories for MANAGED RESOURCES that will be used in several of the Bean's operations can be allocated here and stored in member variables. This is possible because it is client-independent state. The *InitialContext* should also be created only once, because it can take a significant time to create.

The following skeleton Stateless Session Bean implementation can be used as a template:

```
// imports
public class MailingSystemEJB implements javax.ejb.SessionBean {
  private InitialContext initialContext = null;
  // see NAMING example for details
  private SessionContext sessionContext = null;
  // see COMPONENT CONTEXT example for details

  public void ejbCreate() {
    // look up and store resource factories here
  }

  public void setSessionContext( SessionContext ctx ) {
    sessionContext = ctx;
    initialContext = new InitialContext();
  }

  public void ejbRemove() {
    // release resource factories here
  }

  public void ejbActivate() {
   // never called
  }

  public void ejbPassivate() {
   // never called
  }

  // business methods...

}
```

Message-Driven Beans

Lifecycle Operations

A Message-Driven Bean must implement the following operations[9]:

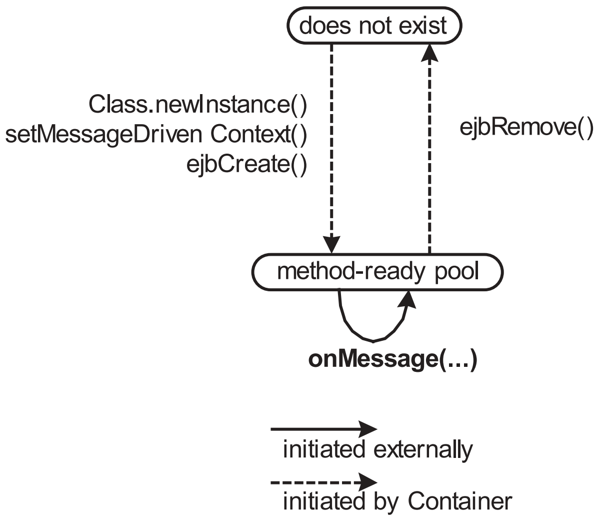
- One parameterless *ejbCreate()* operation.
- One *setMessageDrivenContext(MessageDrivenContext ctx)* operation.
- One *ejbRemove()* operation.
- An *onMessage(Message msg)* operation. This operation is declared in JMS' *MessageListener* interface, which an MDB is required to implement.

Lifecycle

Like the Stateless Session Beans, the lifecycle of a Message-Driven Bean is very simple. As in the case of Stateless Session Beans, it is irrelevant which instance handles a message, as all Message-Driven Beans instances of the same type are equal. INSTANCE POOLING can be implemented easily.

The CONTAINER creates a pool of Message-Driven Beans, usually during start-up. To do this, it first creates the instances, then calls *setMessageDrivenContext()* and *ejbCreate()*. These are the methods the Bean provider can implement to initialize the Message-Driven Bean instance. Thereafter the Bean is considered initialized and put into the method-ready state. It can now handle messages from a JMS queue or topic. Handling a message is initiated by the CONTAINER by invoking *onMessage()* of the *MessageListener* interface[10], passing the message as an argument.

9. 'Implementing' an operation can also mean defining an empty operation. The operation just has to be defined in the Java sense.
10. This could be changed in upcoming revisions of EJB to support other MoM Standards than JMS, such as JAXM.



When the Container needs to remove a Bean instance from the pool, it calls the *ejbRemove()* method.

What goes where

The CONTAINER calls the methods *setMessageDrivenContext()* and *ejbCreate()* once when it creates a new instance of the Message-Driven Bean. In this case an object of the type *MessageDrivenContext* will be given to the Bean implementation. It is the responsibility of the Bean's implementation to store the COMPONENT CONTEXT for later use. In *ejbCreate(),* additional initialization steps can be implemented.

In the method-ready state, a Message-Driven Bean can receive messages from clients. The Bean handles the messages in its *onMessage()* method. Calls to this method will be serialized by the CONTAINER. Nevertheless, many instances of the Bean type can handle messages from one queue or topic in parallel – the CONTAINER manages concurrency.

At the end of the lifecycle of an MDB instance, the *ejbRemove()* method is called by the CONTAINER. The Bean implementation should free resources at this point.

The following skeleton Message-Driven Bean implementation can be used as template:

```
// imports
public class MessageReceiverEJB implements
  javax.ejb.MessageDrivenBean, javax.jms.MessageListener {

  private MessageDrivenContext messageContext = null;

  public void ejbCreate() {
   // look up and store resource factories here
  }

  public void setMessageDrivenContext
   ( MessageDrivenContext ctx ) {
     messageContext = ctx;
  }

  public void ejbRemove() {
   // release resource factories here
  }

  public void onMessage(javax.jms.Message msg) {
   // implement business logic here
  }
}
```

As you can see, the Message-Driven Bean has to implement two interfaces. The reason for this is to make the *MessageDrivenBean* interface independent of the underlying message-oriented middleware's interface requirements. It will be possible in forthcoming EJB versions to use a different interface than MessageListener here to provide MDBs for messaging middleware other than JMS. An example might be JAXM, the Java API for XML Messaging.

Stateful Session Beans

Lifecycle operations

Stateful and Stateless Session Beans implement the same *SessionBean* interface. They have to implement almost the same operations, only the creators are different – Stateless Session Bean creators must not take any parameters. However, some of the operations have different semantics. The following are required:

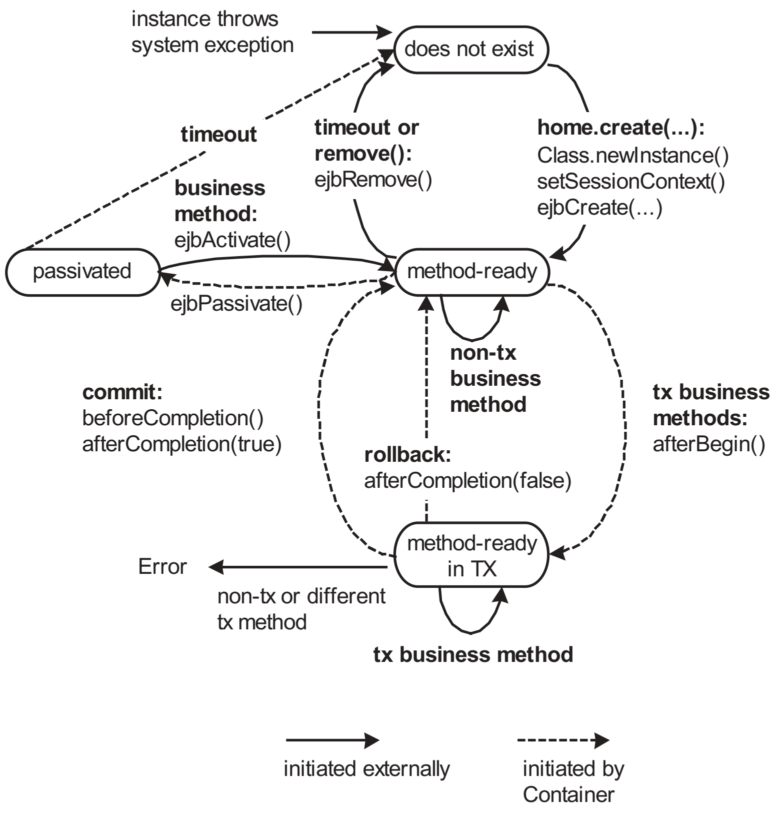
- One *ejbCreate(…)* operation for each *create(…)* operation in the Home Interface. In contrast to Stateless Session Beans, Stateful Session Beans can carry client-specific state between different invocations. Therefore it makes sense to be able to initialize a Stateful Session Bean while creating it. This is why it is possible to define many *create(…)* operations in the Remote and/or Local Interface. The ellipsis is a placeholder for different user-defined parameter combinations. For every *create(…)* operation, an *ejbCreate(…)* operation with the same parameter set has to be implemented in the Session Bean's implementation class.
- An *ejbActivate()* operation, called whenever the Bean has been reactivated from PASSIVATION.
- An *ejbPassivate()* operation, called just before the instance is PASSIVATED.
- An *ejbRemove()* operation, called when the client calls *remove()* on the Remote Interface or when a timeout occurs.

The following LIFECYCLE CALLBACK operations have to be implemented if a Stateful Session Bean that uses container-managed transactions wants to be notified about state changes in its transaction. They are defined by the *SessionSynchronization* interface:

- *afterBegin()* informs the Bean that a new transaction has been started. Subsequent business operations will be invoked in the context of this transaction.
- *beforeCompletion()* is called by the CONTAINER when the transaction associated with the instance is about to complete. The COMPONENT CONTEXT can be queried to find out whether the transaction will be committed, or rolled back using *getRollback-Only()*. The implementation of the *beforeCompletion()* method can also ensure that the transaction will be rolled back, by calling *setRollbackOnly()*.
- *afterCompletion(Boolean)* is called when the transaction commit protocol has finished. The *Boolean* flag specifies whether the commit was successful, or whether a rollback has been executed.

Lifecycle

Stateful Session Bean instances have a unique state, so not all instances are equal. It is therefore important that subsequent client requests on the same Bean really reach the same instance, or at least an instance with the same state. To conserve resources, PASSIVATION is used here.


instance throws system exception
does not exist
timeout
timeout or remove(): ejbRemove()
home.create(...): Class.newInstance() setSessionContext() ejbCreate(...)
business method: ejbActivate()
passivated
method-ready
ejbPassivate()
non-tx business method
commit: beforeCompletion() afterCompletion(true)
tx business methods: afterBegin()
rollback: afterCompletion(false)
method-ready in TX
Error
non-tx or different tx method
tx business method
initiated externally
initiated by Container

When a client invokes one of the home's *create(…)* operations, a new instance of the implementation class is created, and the *setSessionContext()* operation is called by the CONTAINER to provide it with the necessary COMPONENT CONTEXT. The corresponding *ejbCreate(…)* operation is then invoked.

The Bean is now in the method-ready state, ready to handle business method invocations. Two kind of invocations are possible: invocations on non-transactional methods and those on transactional methods. For non-transactional invocations, the instance simply remains in the method-ready state. If the method invocation is transactional, the Bean goes to the 'method ready in transaction' state, allowing transactional business methods to be invoked. After this, the transaction has either to be committed or aborted. In both cases the Bean returns to the method-ready state. It is considered an error if a method that is marked to require no active transaction is called while the Bean in active in a transaction – the Bean will go into an error state.

If the CONTAINER is running out of resources, or otherwise decides that there are too many physical instances, the Bean instance might be PASSIVATED. This means that the instance's state will be written to secondary memory. Before this happens, *ejbPassivate()* is called. This enables the instance to free resources and bring the instance into a serializable state. When the next business method is called on this Bean by a client, the CONTAINER reactivates the required Bean instance. *ejbActivate()* is called so that the Bean can restore the state necessary to handle business operations. The original business method then will be invoked.

When the client no longer needs an instance, it can call *remove()* on the Remote Interface. Alternatively, if the client doesn't invoke a business operation for a defined interval, a timeout occurs and the container removes the Bean instance to save resources. Before the instance is actually destroyed, the CONTAINER calls *ejbRemove()*, allowing the implementation to clean up resources. If a SYSTEM ERROR occurs during the lifetime of the Bean, the Bean is immediately destroyed.

In some situations a Stateful Session Bean's implementation needs to be informed about state changes of the transaction in which it runs. This is necessary, for example, for cleaning up the state of a Bean if a transaction is rolled back. The Bean therefore needs a means by which it can request information about the transaction status. To be notified, the COMPONENT IMPLEMENTATION has to implement the *SessionSynchronization* interface. The CONTAINER then calls *after-Begin()* when a transaction has been started. Just before a transaction completes, it calls *beforeCompletion()*, giving the Bean a chance to either prepare for commit or ensure the transaction is rolled back by using *setRollbackOnly()*. After the transaction has completed, the CONTAINER notifies the instance by calling *afterCompletion(Boolean)* – the Boolean flag is *true* in case of a commit, otherwise *false*.

It is important to note that implementing this interface is only allowed for Stateful Session Beans with Container-Managed Transactions. This makes sense, because Stateful Session Beans are the only kind of Bean that can carry state from one transaction to another. CMT is the only case in which the Bean does not handle the transaction state by itself, which it does in the case of Bean-Managed Transactions (BMT).

What goes where

You acquire resources in the *ejbCreate(…)* operations and release them in *ejbRemove()*. The same is true for the *InitialContext*. This begs the question – what you have to do in *ejbPassivate()* and *ejbActivate()*?

Before PASSIVATION is about to happen, all non-transient fields of the Bean must be one or more of:

- A Java primitive type
- A *Serializable* object
- *Null*

If the Bean has other types as an attribute, such as non-serializable Java classes, they either have to be declared *transient* – which means that they are ignored when the instance is serialized during PASSIVATION – or the developer must use code in *ejbPassivate()* to ensure that the respective attributes are set to *null*. This also means that *ejbActivate()* has to be used after reactivation, to re-set values for these attributes.

This is not necessary for:

- References to another EJB's Remote/Local Interface
- References to another EJB's Home/Local Home Interface
- A *SessionContext* object
- A COMPONENT-LOCAL NAMING context (as opposed to other NAMING contexts)

- A user transaction
- References to MANAGED RESOURCE factories

So the procedure must only be used for references to individual resources such as concrete database connections and other non-serializable attributes that do not fall into one of the categories above.

So, the template could look like the following:

```
public class ShoppingCartEJB implements SessionBean {
  private InitialContext initialContext = null;
  private SessionContext sessionContext = null;

  public void ejbCreate() {
    // look up and store resources here
  }

  public void setSessionContext( SessionContext ctx ) {
    sessionContext = ctx;
    initialContext = new InitialContext();
  }

  public void ejbRemove() {
    // release resources here
  }

  public void ejbActivate() {
    // look up and store resources here
    // reinitialize transient fields here
  }

  public void ejbPassivate() {
    // release resources here
  }

  // business methods...

}
```

Entity Beans

Lifecycle Operations

Some of the operations have the same names as the Session Bean's LIFECYCLE CALLBACK operations, but they have a different semantic meaning. The operations are:

- *setEntityContext()* and *unsetEntityContext()*, which are used to pass the *EntityContext* to the instance and to remove it, respectively. The *EntityContext* object provided by the *setEntityContext()* operation can be stored locally for later use. These methods must also be used to set up or release any data needed by a physical instance during its complete lifecycle, such as MANAGED RESOURCE factories.
- *ejbLoad()* is called by the CONTAINER when the instance is in the ready state, associated with a logical entity. The operation must load its state from the database. It can query the COMPONENT CONTEXT to find out about its current identity by calling *EntityContext.getPrimaryKey()*.
- *ejbStore()* is called only when the instance is in the ready state. In this operation, the state must be stored back into the database.
- *ejbActivate()* is called when a pooled instance needs to be associated with a logical entity. The identity is available through the COMPONENT CONTEXT. *ejbActivate()* might be called as a reaction to a client's call to a finder operation on the Home Interface.
- *ejbPassivate()* is used whenever an instance should be disassociated from a logical entity, usually to put it back to the pool.
- *ejbRemove()* is called when the logical entity should be removed, rather when the instance is no longer needed. It is important to note the difference here between Entity Beans and Session Beans. The operation must delete the data in the database to ensure that the logical entity no longer exists after *ejbRemove()* has been called. *ejbRemove()* will be called by the CONTAINER when the client calls *remove()* on the Remote or Local Interface.
- *ejbCreate(…)* and *ejbPostCreate(…)* operations are called as the CONTAINER's reaction to a client's call to the corresponding *create(…)* operation on the Home Interface. The implementation must then insert the record into the database. Note that no additional *ejbActivate()* operation is invoked. As with Stateful Session Beans, an Entity Bean can provide several *create(…)* operations in its COMPONENT HOME.

However, the parameters of every create operation must contain enough data to construct the PRIMARY KEY[11].

- The *ejbFindByXXX(…)* operations are called in reaction to a client calling *findByXXX(…)* on the Home Interface. *XXX* can be any descriptive name and the parameter set can also be chosen by the Bean developer. *ejbFindByXXX()* returns a *Collection* (or *Enumeration*) of PRIMARY KEYS. The state and thus the identity of the Bean instance will not be changed by the finder methods.

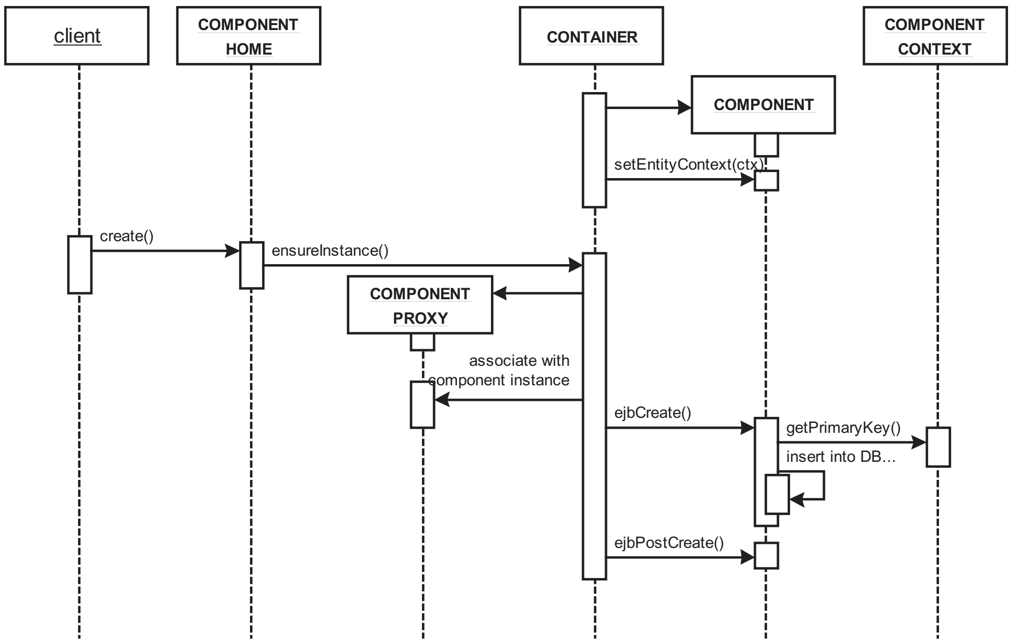
  Note that the descriptions above assume that the database access was coded explicitly and Bean-Managed Persistence (BMP) was used. In the case in which the APPLICATION SERVER is responsible for persistence – Container-Managed Persistence – *ejbFindByXXX()* methods need not be present in the implementation, and *ejbLoad()/ejbStore()* and *ejbCreate()/ejbRemove()* need not contain any database code, as this is handled by the CONTAINER. They are still called at the appropriate times in the lifecycle, so they must be present, albeit possibly empty.

- The *ejbSelectXXX()* operations have been introduced with EJB 2.0. These are intended for Bean-internal use in the case of CMP. Their definition therefore differs from the definition of the finders – the select methods are defined as abstract methods in the Bean implementation, not in one of the interfaces, so they can only be used internally to the Bean.

  Another difference between the finders and the select methods is that select methods can be invoked on instances in the ready state, while finders will only be invoked on instances in the pooled state. A select method is also allowed to return CMP field types, not only *EJBObjects* (or PRIMARY KEYS, respectively) or *Collections* of them. As *ejbSelectXXX()* operations can only be called from inside the Bean, these usually return *EJBLocalObjects* for performance reasons – see COMPONENT INTERFACE.

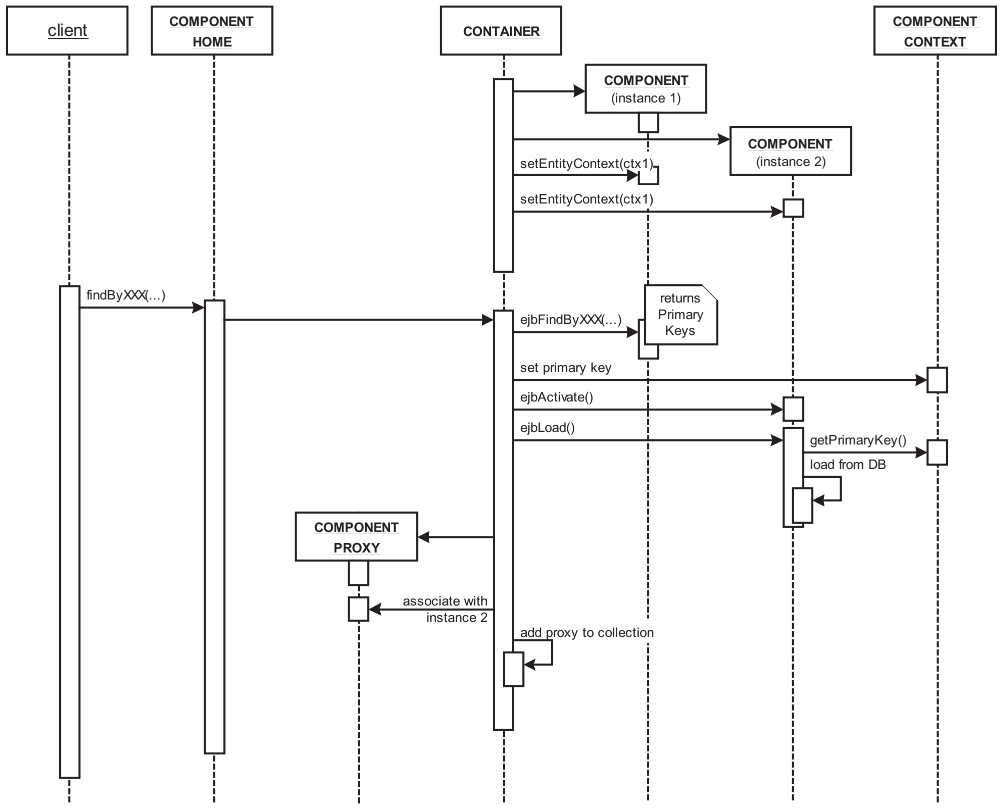
11. If the PRIMARY KEY is created automatically, either by the database or by some application logic, without being dependent on the attributes of the instance, then ‘enough data’ is satisfied by passing no arguments.

The following set of sequence diagrams illustrates the descriptions above. The first shows the creation of an Entity Bean instance:



When the server starts, it creates one or more COMPONENT instances and supplies the COMPONENT CONTEXT. The client eventually calls *create()* on the Home Interface. The server creates a COMPONENT PROXY and associates it with the instance. It then calls *ejbCreate()* on the instance, which retrieves the PRIMARY KEY from the COMPONENT CONTEXT and inserts the respective data into the database. The CONTAINER then calls *ejbPostCreate()*.

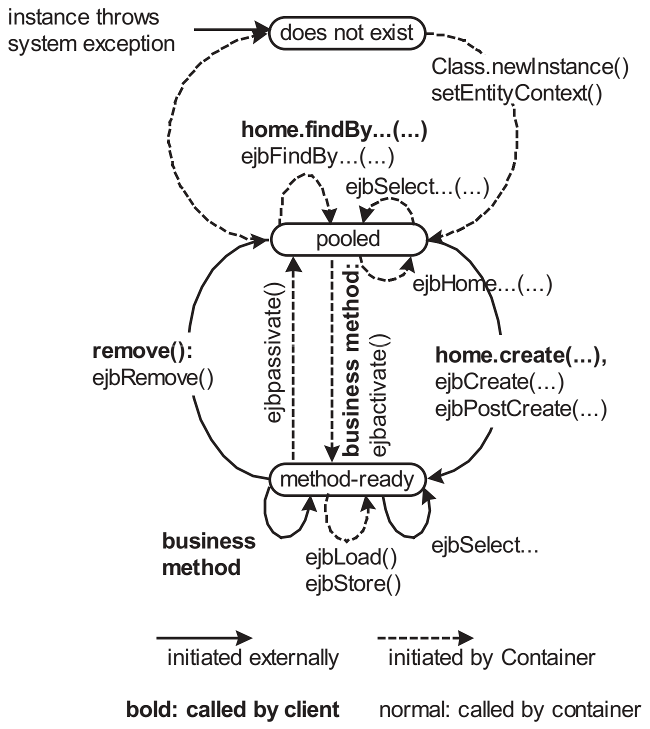
The next diagram illustrates the process of finding and activating an Entity Bean:



As before, when the server starts up, it creates a pool of instances of each Bean. When a client calls a finder operation on the Home Interface, the Home Interface implementation delegates to a pooled instance, which executes the query. Note that only proxies for the individual instances are created – the creation of an actual implementation is delayed until a business method is called.

Lifecycle

An Entity Bean's lifecycle is completely different from that of Session Beans. Although some operations have the same names, they have different semantics. This is because Entity Beans are persistent and stateful. The CONTAINER uses INSTANCE POOLING to manage instances.



When the CONTAINER starts up, it creates a pool of physical instances for each Entity Bean. At the start of their lifecycle, these instances do not yet represent a logical entity – see VIRTUAL INSTANCE for an explanation of the terms. When they are put into the pool, they are passed an *EntityContext* that the Bean can store locally. From that point on, the server can proceed in different ways:

- It can use a pooled instance to create a new logical entity. This happens when the client calls one of the *create(…)* operations on the Home Interface. The CONTAINER invokes *ejbCreate(…)* and *ejbPostCreate(…)* as a consequence. In the *ejbPostCreate(…)* operations, the Bean's identity and remote object are available through the COMPONENT CONTEXT – this is not the case in *ejbCreate(…)* operations. So the reason for the existence of *ejbPostCreate()* is to provide a place for initialization that takes this information into account. The instance now represents a logical entity and can be used for method invocations – it is method-ready.

- Alternatively, the Bean instance can be used to execute queries implemented in a finder operation. This happens when the CONTAINER calls the respective *ejbFindByXXX(…)* operation as a reaction to a client's call to a *findByXXX(…)* operation on the Home Interface. This operation executes and returns the PRIMARY KEY(S) of the logical entities matching the query. Executing the operation does not change the logical identity or the state of the instance – they stay pooled. It is therefore not a good idea to change or access the state of the instance that executes a finder, because the change will probably be lost – no *ejbStore()* will be called. Only when a client really accesses one of the entities returned by *findByXXX(…)* invocations will the CONTAINER use *ejbActivate()* to make one of the pooled instances ready to handle invocations.

  Before calling *ejbActivate()*, however, the CONTAINER will also change the PRIMARY KEY in the COMPONENT CONTEXT. This is important, because the CONTAINER will now invoke *ejbLoad()* on the instance, the implementation of which will load the COMPONENT's state, completing the 'impersonation' of the logical entity. In *ejbLoad()*, the Bean accesses the COMPONENT CONTEXT to find the PRIMARY KEY of the Bean it should impersonate, and then queries the database accordingly and sets its internal state. Note that this means that it is generally not a good idea to store direct references to the content of the COMPONENT CONTEXT, as it might become outdated.

  The instance is now method-ready for the respective logical entity. Directly after this, the original business operation called by the client on the Local or Remote Interface will be invoked on the respective instance.

While a Bean is method-ready, it can receive several operation invocations for the same logical entity. The *ejbStore()* operation is called at least at the end of each transaction, to synchronize the Bean's state with the persistent store, but *ejbStore()* might also be called at any time before this, whenever the CONTAINER thinks it might be appropriate to synchronize the Bean's state with the database. The extreme case would be to call *ejbStore()* after each invocation of an operation.

Whenever the CONTAINER wants the instance to represent a different logical entity, it first calls *ejbPassivate()* to put the instance back into the pool, then changes the PRIMARY KEY in the COMPONENT CONTEXT, and finally calls *ejbActivate()* and *ejbLoad()* to make it method-ready again.

When the CONTAINER no longer needs an instance to impersonate a logical entity, *ejbPassivate()* is called to return it to the pool of free instances. If a logical entity needs to be deleted, the client calls *remove()* on the Local or Remote Interface, which results in the CONTAINER calling *ejbRemove()*. This removes the entity's data from the database, so references from the client are now invalid.

If the instance throws a SYSTEM ERROR during a method call, it will be discarded. It therefore goes to the 'does not exist' state.

selectXXX() operations, introduced in EJB 2.0, are a special case. These can be called in both the method-ready and the pooled state.

What goes where

Resources that are required throughout the lifecycle of a physical instance, such as MANAGED RESOURCE factories or NAMING contexts, should be acquired in *setEntityContext()* and released in *unsetEntity-Context()*. Resources that depend on a specific logical instance should be acquired in *ejbActivate()* and released in *ejbPassivate()*. This difference makes sense, because *setEntityContext()* and *unsetEntityContext()* are called at the very beginning and the end of the lifecycle of the physical entity. Note that *ejbCreate()* is only called at the beginning of the lifecycle if a new *logical* entity is created. During this lifecycle, the physical entity can represent several different logical entities. Each time the logical entity changes, *ejbActivate()* and *ejbPassivate()* are called.

Thus, the template could look like the following:

```
public class Address implements EntityBean {
  private InitialContext initialContext = null;
  private EntityContext entityContext = null;

  public void setEntityContext( EntityContext ctx ) {
    entityContext = ctx;
    initialContext = new InitialContext();
    // acquire resource factories here
  }

  public void unsetEntityContext() {
    entityContext = null;
    initialContext = null;
    // release resource factories here
  }

  public PKClass ejbCreate(...) {
    // insert record in DB here
    // return PK (BMP) or null (CMP)
  }

  public void ejbPostCreate(...) {
    // use primary key and remote reference for
    // addition initialization here
  }

  public void ejbLoad() {
    Object somePK = context.getPrimaryKey();
    // query DB for data and update fields (BMP)
  }

  public void ejbStore() {
    // store state in DB here (BMP)
  }

  public void ejbRemove() {
    // delete stuff in database (BMP)
  }

  public void ejbActivate() {
    // get instance-specific resources here
    // get resources that could not be retained
    // during passivation
  }
  public void ejbPassivate() {
     // release instance-specific resources here
     // release resources that can not be retained during
     // passivation
   }

   // business methods...

 }
```

Annotations

The ANNOTATIONS pattern describes a way in which technical concerns can be specified as part of the COMPONENT without actually implementing them manually. This is necessary because the CONTAINER is responsible for implementing such technical concerns, rather than the COMPONENT IMPLEMENTATION.

In EJB, ANNOTATIONS are called *Deployment Descriptors*. A Deployment Descriptor must be created for each deployed Bean. The Deployment Descriptor is an XML file with a specific Document Type Definition (DTD)[12]. DTDs define which XML elements are legal in a specific XML file. Deployment Descriptors are either created manually, or by using either a graphical tool provided by the APPLICATION SERVER or special features of integrated development environments (IDEs).

Manual creation of the Deployment Descriptors is useful for larger projects in which bulk deployments or integration with automated build processes are necessary. This might be especially true if you need to support many different APPLICATION SERVERS. APPLICATION SERVERS are allowed to add their specific parts to the Deployment Descriptor, and these are different for each APPLICATION SERVER. If you want to create the Deployment Descriptor for only one Bean, use of a GUI tool might be simpler. We illustrate aspects of this tool as part of the COMPONENT INTROSPECTION example in Chapter 15.

A Deployment Descriptor usually consists of two files: the standard descriptor, as specified by the EJB standard, and a CONTAINER-specific descriptor. The structure of the CONTAINER-specific descriptor is defined by the CONTAINER vendor. Both can be found in the EJB *.jar* file – see COMPONENT PACKAGE.

12. Note that in EJB 1.0 the Deployment Descriptor was a serialized object. EJB 1.1 changed this to an XML document.

Contents of the standard Deployment Descriptor

The following section illustrates some aspects of the Deployment Descriptor. We do not aim to give a complete description, only to highlight some interesting examples.

Just as any XML document, the Deployment Descriptor starts with a document type declaration:

```
<?xml version="1.0" encoding="Cp1252"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-
jar_1_1.dtd'>
```

Structural description

The structure of the Beans is defined in the first part of the Deployment Descriptor. This includes the classes and interfaces that make up the Bean and the references to other EJBs or MANAGED RESOURCES.

The root element of a Deployment Descriptor is *ejb-jar*. As an EJB *.jar* file can contain several Beans, the next two tags, *description* and *display-name*, are used to specify information on the group of Beans in the *.jar* file. Each contained Bean is then listed in the *enterprise-Beans* tag, either within an *entity* or a *session* tag:

```
<ejb-jar>
 <description>...</description>
 <display-name>Address-Bean Package</display-name>
 <enterprise-beans>
  <entity>
```

Within this tag, which is present for each Bean in the *.jar* file, there is again a *description*, *display-name* and *ejb-name* tag:

```
 <description>Represents an Address for a Party</description>
 <display-name>Address</display-name>
 <ejb-name>Address</ejb-name>
```

Things now get a bit more interesting. The Deployment Descriptor next lists the names of the following classes: the Local Interface and Local Home Interface (if present), the Home Interface and the Remote

Interface (if present), and the implementation class. These names are important for the CONTAINER, to allow it to instantiate the Beans or to create code that instantiates them respectively. In this example, only a Home Interface, a Remote Interface and the implementation are provided:

```
    <home>de.mathema.example.AddressHome</home>
    <remote>de.mathema.example.Address</remote>
    <ejb-class>de.mathema.example.AddressBean</ejb-class>
```

The next couple of tags specify how persistence is handled in the current Bean. In the example here, we use Bean-Managed Persistence. The PRIMARY KEY class is especially important, because the generated wrapper class must use this class as a formal return value:

```
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
```

The next section is also interesting, as it is a form of REQUIRED INTERFACE. It lists the set of Beans that are referenced from within this Bean:

```
    <ejb-ref>
      <ejb-ref-name>MCCFConfigService</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <home>de.mathema.mccf.ejb.MCCFConfigHome</home>
      <remote>de.mathema.mccf.ejb.MCCFConfig</remote>
    </ejb-ref>
```

Lastly, references to MANAGED RESOURCES are also specified:

```
    <resource-ref>
      <res-ref-name>config_ADDRESS</res-ref-name>
      <res-type>java.net.URL</res-type>
      <res-auth>Application</res-auth>
    </resource-ref>
```

This section also contains the CONFIGURATION PARAMETERS. These are described in more detail in the example of each respective pattern.

```
  </entity>
</enterprise-beans>
```

Behavioral description

The behavioral part of the Deployment Descriptor specifies security and transaction attributes for each operation of the COMPONENT. Note that the Home Interface is also part of the COMPONENT, so its operations are also annotated here.

First, the security roles are specified. Later, at deployment time, accounts of the user management system are assigned to these roles – this provides a level of indirection, so that the Bean does not need to reference user accounts directly, which would limit portability across different user management systems. Here, two roles are defined:

```
<assembly-descriptor>
 <security-role>
  <role-name>admin</role-name>
  <role-name>regular-user</role-name>
 </security-role>
```

The method permission section lists, for each of the above-mentioned roles, the operations users of the role are allowed to invoke. For each method, the *ejb-name*, the interface and the name, as well as its parameters, are specified.

The following section, for example, specifies that users of the *admin* role are allowed to invoke the *remove(Object)* operation of the Home Interface of the *Address* EJB, which is a technical operation provided by the Home Interface:

```
<method-permission>
  <role-name>admin</role-name>
  <method>
   <ejb-name>Adress</ejb-name>
   <method-intf>Home</method-intf>
   <method-name>remove</method-name>
   <method-params>
    <method-param>java.lang.Object</method-param>
   </method-params>
  </method>
 </method-permission>
```

The following method permission defines *regular-user*'s access to the *setCity(String)* operation of the Remote Interface:

```
  <method-permission>
    <role-name>regular-user</role-name>
    <method>
     <ejb-name>Adress</ejb-name>
     <method-intf>Remote</method-intf>
     <method-name>setCity</method-name>
     <method-params>
      <method-param>java.lang.String</method-param>
     </method-params>
    </method>
   </method-permission>
```

The next part of the Deployment Descriptor defines the transaction attributes for each operation. It uses the same format to define the method as for security attributes, so the schema should be understandable:

```
  <container-transaction>
    <method>
     <ejb-name>Adress</ejb-name>
     <method-intf>Home</method-intf>
     <method-name>remove</method-name>
     <method-params>
      <method-param>java.lang.Object</method-param>
     </method-params>
    </method>
    <trans-attribute>Supports</trans-attribute>
   </container-transaction>
  </assembly-descriptor>
 </ejb-jar>
```

Typical contents of the vendor-specific part

As the name implies, this part of the descriptor is vendor-specific, so we cannot give a general example here. However, there are specific contents you will typically find in vendor-specific Deployment Descriptors.

Security role mapping

In the standardized Deployment Descriptor, security information is specified using logical role names only. Upon deployment, these roles have to be mapped to concrete users, or more generally, principals. The vendor-specific part of the Deployment Descriptor is the place to do this, because the concrete implementation of security concepts is not part of the EJB standard.

The following part of the Deployment Descriptor for the J2EE Reference Implementation maps the *admin* role to a principal named *administrator* and all members of the group named *itDeparment*:

```
<rolemapping>
 <role name="admin">
  <principals>
   <principal>
    <name>administrator</name>
   </principal>
  </principals>
  <groups>
   <group name="itDepartment" />
  </groups>
 </role>
</rolemapping>
```

JNDI registration and mapping

The next section registers the Bean in the JNDI context and maps the local resource references to global ones. First, the current Bean, the *Address* Bean, is registered in the JNDI context under the name *mathema/example/Address*:

```
<enterprise-Beans>
 <ejb>
  <ejb-name>Address</ejb-name>
  <jndi-name>mathema/example/Address</jndi-name>
```

Then the logical names of the resource references are mapped to concrete entries in the global JNDI context. In this case, as above, we use a reference to a MANAGED RESOURCE, namely a database connection factory. The logical name *config_ADDRESS* is mapped to a database connection factory in the global JNDI context:

```
<resource-ref>
 <res-ref-name>config_ADDRESS</res-ref-name>
  <jndi-name>jdbc/AdressDB/p:/data/address.def</jndi-name>
 </resource-ref>
<resource-ref>
```

See COMPONENT-LOCAL NAMING CONTEXT for more detail.

Persistence configuration

Persistence configuration is not standardized in EJB 1.1 and 2.0. The part that is important to the programmer is standardized, of course, such as what to implement in operations like *ejbStore()* for Bean-Managed Persistence. The information the CONTAINER might need to implement persistence, especially in the case of CMP, is left to the CONTAINER provider, however.

We don't want to go into details here, but information that is typically required is the name of the data source, the structure of the used tables, et cetera. Besides the obvious mapping of an Entity Bean type to a specific table in the database, other mappings in which the data is distributed across different tables are often used.

11 EJB Container Implementation Basics

This chapter is an overview of CONTAINER implementation. We highlight only the most important issues from a COMPONENT developer's point of view.

Implementing a CONTAINER is non-trivial, and includes many issues that are outside the scope of this book. However, such deeper technical details should be invisible to a COMPONENT developer. This is also the reason why some of the examples are rather short.

In places we will use the implementation of JBoss or the Sun Reference Implementation to illustrate a concrete implementation of a pattern. While the chapter describes details of the internal workings of the CONTAINER, it is important to understand the patterns and their examples, because patterns such as VIRTUAL INSTANCE lay the foundation for the complete EJB run-time environment.

Virtual Instance

EJB implements this pattern to keep resource consumption for Bean instances low.

The key to implementing the pattern, as described in that pattern's section, is to use a COMPONENT PROXY together with a set of LIFECYCLE CALLBACK operations. Requests arrive initially at the COMPONENT PROXY. This first handles technical concerns, then selects a suitable instance to handle the request, prepares it and finally forwards the request to the instance. The preparation of the instance is done by calling the appropriate LIFECYCLE CALLBACK operations on the instance.

The example for LIFECYCLE CALLBACK provides details of the responsibilities required to make VIRTUAL INSTANCES possible that are placed on the COMPONENT developer. EJB uses INSTANCE POOLING and PASSIVATION as two concrete ways to keep resource usage low. These two patterns and their examples will be studied in more detail. These are also the concrete examples of how VIRTUAL INSTANCE is actually implemented in EJB servers.

Instance Pooling

INSTANCE POOLING basically means that the CONTAINER creates a pool of instances of a specific COMPONENT, then reuses these instances over time to handle subsequent method invocations on different logical instances.

The main advantage is that repeated object creation and garbage collection are avoided, because instances are reused. Garbage collection, especially, can slow down applications significantly. A further benefit is that, due to reuse, resources for the instances need to be acquired and released less often, significantly improving the performance of the application.

To implement INSTANCE POOLING, the Bean must provide the necessary LIFECYCLE CALLBACK operations. These enable the CONTAINER to notify the Bean of important events in its lifecycle, especially those related to INSTANCE POOLING. In these operations, the Bean has to react in a way that prepares it for the new role it is about to perform. For details of the necessary operations, see the examples for LIFECYCLE CALLBACK.

INSTANCE POOLING is used in EJB for two kinds of Bean: Stateless Session Beans and Entity Beans. Pooling is implemented quite differently for these two Bean types, however, because one of them is stateless, which makes INSTANCE POOLING trivial. The other kind, Entity Beans, has persistent data and a logical identity.

Pooling for Stateless Session Beans

Stateless Session Beans have no client-visible state. Thus the CONTAINER can use any instance in a specific Bean's pool to serve any request. There is no notion of a 'session' or any other dependency on previous invocations on the Bean. When a request arrives for such a Bean, the CONTAINER can take any available Bean from the pool and dispatch the request to it.

In the JBoss implementation [JBOSS] the CONTAINER uses a *StatelessSessionInstancePool* that provides an operation *get()*. It simply takes the next Bean instance from the pool and returns it. If none are available, a new instance is created, thus the pool grows dynamically:

```
public class StatelessSessionInstancePool
  extends AbstractInstancePool {
  ...
}

public abstract class AbstractInstancePool
    implements InstancePool {
  public synchronized EnterpriseContext get()
    throws Exception {
    if ( !pool.empty()){
      return (EnterpriseContext)pool.pop();
    } else {
      try {
        return create(container.getBeanClass().newInstance());
      } catch (InstantiationException e) {
        throw new ServerException
         ("Could not instantiate Bean", e);
      } catch (IllegalAccessException e) {
        throw new ServerException
         ("Could not instantiate Bean", e);
      }
    }
  }
}
```

If the Bean is no longer required, the *free()* operation of the *StatelessSessionInstancePool* is called to push the Bean back onto the stack of free instances:

```
public synchronized void free(EnterpriseContext ctx) {
 ctx.clear();
 if (pool.size() < maxSize) {
  pool.push(ctx);
 } else {
  discard(ctx);
 }
}
```

Who invokes the *get()* and *free()* operations? JBoss uses a linked list of interceptors to implement the functionality of the different CONTAINERS for the various kinds of Bean. Each interceptor handles a specific aspect of the overall functionality – for example, there are interceptors for pooling, transactions, security, and more. It is therefore one of the interceptors in the CONTAINER that invokes the *get()* and *free()* operations – in this case it is the *StatelessSessionInstanceInterceptor.* Its *invoke()* operation basically looks like this:

```
public class StatelessSessionInstanceInterceptor
  extends AbstractInterceptor {
   public Object invoke(MethodInvocation mi) throws Exception {
```

The first step in the implementation is to access the associated *InstancePool* to obtain an instance on which to invoke the operation. This instance, an *EnterpriseContext,* is stored in the *MethodInvocation* object for use in a downstream *Interceptor*:

```
  mi.setEnterpriseContext(container.getInstancePool().get());
```

The request is then handed over to the next downstream interceptor:

```
  try {
    return getNext().invoke(mi);
  } catch (RuntimeException e) {
    mi.setEnterpriseContext(null);
    throw e;
  } // catch more errors...
```

When the execution of the method invocation is done, the instance is returned to the pool by calling *free()* and passing the *EnterpriseContext* object:

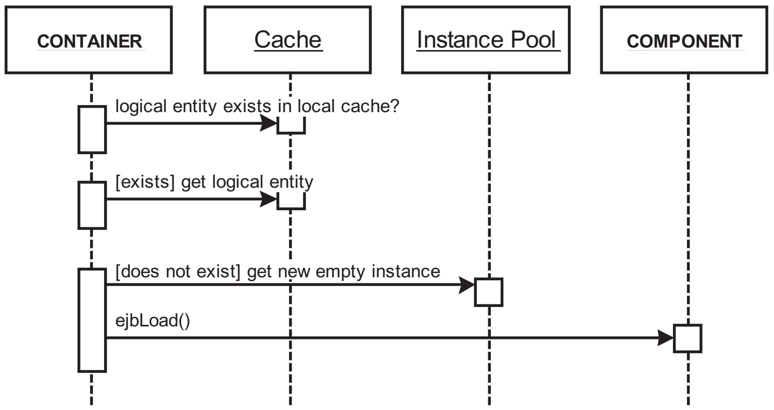
```
  } finally {
    if (mi.getEnterpriseContext() != null)
    container.getInstancePool().free(mi.getEnterpriseContext());
  }
 }
}
```

ejbCreate() and *ejbRemove()* are only called directly after the creation and just before the deletion of the physical instance, so the call to these operations is not visible in the above code for handling a specific request.

Pooling for Entity Beans

INSTANCE POOLING is a bit more complex for Entity Beans. When a request arrives for a specific logical entity, the CONTAINER must make sure that a physical instance that impersonates the required logical entity is available. It therefore takes a Bean from the pool, calls *ejbActivate()*, then calls *ejbLoad()* to make the Bean impersonate the required logical entity. Before it does that, it may check a cache to see whether a Bean instance that impersonates the required logical entity is already available.

Once a suitable instance is found, the request can then be forwarded. The following sequence diagram shows this:


CONTAINER
Cache
Instance Pool
COMPONENT
logical entity exists in local cache?
[exists] get logical entity
[does not exist] get new empty instance
ejbLoad()

For a concrete example of an implementation, let's look at JBoss again. As mentioned above, JBoss uses interceptors to implement the functionality of a CONTAINER. Here the chain of interceptors starts with an interceptor that manages Entity Bean INSTANCE POOLING:

```
public class EntityInstanceInterceptor
  extends AbstractInterceptor {
   ...
   public Object invoke(MethodInvocation mi) throws Exception {
```

First, the identifier of the logical identity is taken from the request. The instance cache for the CONTAINER is then retrieved. Note that the instance cache is necessary, because a logical entity must be represented in the CONTAINER only once to preserve consistency:

```
CacheKey key = (CacheKey)mi.getId();
AbstractInstanceCache cache =
  (AbstractInstanceCache)container.getInstanceCache();
```

In the next step the lock object for this instance is retrieved. This is necessary, as before, to preserve consistency in the face of several concurrent accesses to the same logical entity. The lock is then acquired and the instance checked to ensure that it is not currently in an active transaction (the checking code is not shown):

```
Sync mutex = (Sync)cache.getLock(key);
mutex.acquire();
```

The *acquire()* operation blocks until the instance becomes available, when it is then taken from the cache. This happens in the following section of code:

```
ctx = cache.get(key);
mi.setEnterpriseContext(ctx);
return getNext().invoke(mi);
```

Note that if no instance in the cache represents the requested logical entity, *ejbActivate()* is called in the *get()* operation. In other words, to make an instance represent a different logical entity, the instance goes through the pooled state and so *ejbPassivate()* and *ejbActivate()* are called on it.

Further downstream, an *EntitySynchronizationInterceptor* is responsible for invoking the *ejbLoad()* operation on the respective instance. Its *invoke()* operation looks roughly like the following:

```
public Object invoke(MethodInvocation mi) throws Exception {
 EntityEnterpriseContext ctx =
   (EntityEnterpriseContext)mi.getEnterpriseContext();
```

First, it checks if the instance has the correct state loaded. If this is not the case, the associated persistence manager is told to load the state for the instance:

```
  if (!ctx.isValid()) {
    ((EntityContainer)getContainer()).
     getPersistenceManager().loadEntity(ctx);
  }
```

Downstream interceptors are then invoked, because the Bean state is now correctly loaded:

```
  return getNext().invoke(mi);
```

Now let's look at the persistence manager, which has been called by the interceptor in the last step. It in turn invokes the *ejbLoad()* operation on the instance:

```
public class BMPPersistenceManager
  implements EntityPersistenceManager {
  Method ejbLoad;

  public void init() throws Exception {
    ejbLoad = EntityBean.class.getMethod("ejbLoad", new Class[0]);
    ...
  }

  public void loadEntity(EntityEnterpriseContext ctx)
    throws RemoteException {
    try {
      ejbLoad.invoke(ctx.getInstance(), new Object[0]);
    } catch (IllegalAccessException e) {
       // handle errors
    }
  }
}
```

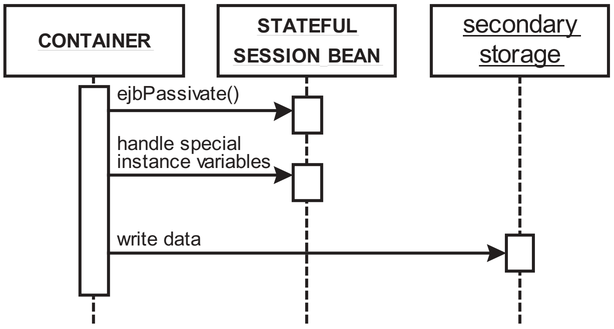
This completes the preparation of the instance, and a downstream interceptor will now do the actual method invocation on the physical instance.

Passivation

The PASSIVATION pattern has two main aspects. First, instances in memory have to be passivated after they have been unused for long enough – the actual act of PASSIVATION. Second, passivated instances that are accessed by a method call need to be activated again.

Passivating a Bean instance

PASSIVATION can easily be achieved by a background thread that checks the set of active instances to find out if any have been unused for a specific time. To make this possible, the time of the last invocation on the instance is stored with each instance. Any instance the thread finds will then be passivated by writing its state to secondary storage. The next sequence diagram shows this process:



Before this actually happens, the instance must be able to prepare itself for the forthcoming PASSIVATION, for example by releasing references to MANAGED RESOURCES. The *ejbPassivate()* LIFECYCLE CALLBACK operation is used for this purpose.

JBoss, for example, uses a *StatefulSessionFilePersistenceManager* to handle PASSIVATION: its *passivateSession()* operation passivates an instance by writing the Bean's state to a 'flat' file:

```
public void passivateSession(
 StatefulSessionEnterpriseContext ctx) throws RemoteException {
  try {
   ejbPassivate.invoke(ctx.getInstance(), new Object[0]);
   ObjectOutputStream out =
     new SessionObjectOutputStream(new
     FileOutputStream(getFile(ctx.getId())));
   out.writeObject(ctx.getInstance());
   out.close();
  } // handle errors
}
```

This operation is called by the *StatefulSessionInstanceCache's passivate()* operation:

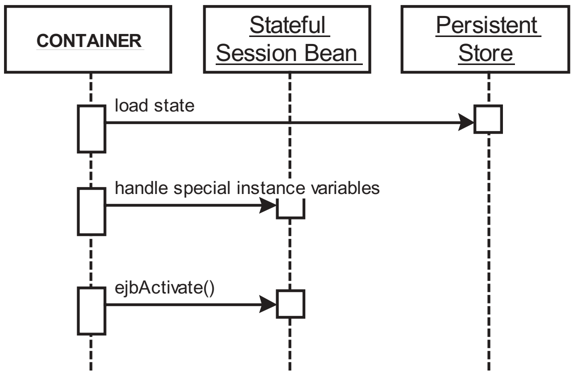
```
protected void passivate(EnterpriseContext ctx)
  throws RemoteException {
    m_container.getPersistenceManager().
      passivateSession((StatefulSessionEnterpriseContext)ctx);
    m_passivated.put(ctx.getId(),
      new Long(System.currentTimeMillis()));
}
```

The operation is in turn called by the background thread mentioned before.

Reactivation of passivated instances

Reactivation of passive instances is simple. The CONTAINER keeps a list of the identifiers of passivated instances. Whenever a request arrives for such a passivated instance, it loads the instance from disk. References to other Beans, to the *SessionContext*, the *UserTransaction* and the *NamingContext* must also be restored. These might point to different but similar instances.

Subsequently, the CONTAINER calls *ejbAcivate()* to allow the instance to re-obtain references to *Managed Resources* and to do additional preparation work. The next sequence diagram shows this:



Let's look at the JBoss implementation again. In the *StatefulSessionInstanceInterceptor's invoke()* operation, the instance with the specified identifier is taken from the cache and the invocation forwarded to the next interceptor in the chain:

```
public Object invoke(MethodInvocation mi) throws Exception {
 AbstractInstanceCache cache =
   (AbstractInstanceCache)container.getInstanceCache();
 EnterpriseContext ctx =
   container.getInstanceCache().get(mi.getId());
 mi.setEnterpriseContext(ctx);
 return getNext().invoke(mi);
}
```

The cache's *get()* operation first looks into the instance cache. If it finds a suitable instance, this is returned:

```
public EnterpriseContext get(Object id) throws RemoteException,
 NoSuchObjectException {
  EnterpriseContext ctx = (EnterpriseContext)getCache().get(id);
  if ( ctx != null ) return ctx;
```

Otherwise, it creates a new, empty Bean, sets its identifier to the required value and then calls *activate()*:

```
 if (ctx == null) {
  ctx = acquireContext();
  setKey(id, ctx);
  activate(ctx);
  return ctx;
}
```

Activate, as the next fragment shows, is implemented to call *activate-Session()* on the associated persistence manager:

```
protected void activate(EnterpriseContext ctx)
  throws RemoteException {
    m_container.getPersistenceManager().
      activateSession((StatefulSessionEnterpriseContext)ctx);
    m_passivated.remove(ctx.getId());
}
```

If this is a *StatefulSessionFilePersistenceManager*, then the state is loaded from a file and the new instance's *ejbActivate()* operation is called:

```
public void activateSession(StatefulSessionEnterpriseContext ctx)
throws RemoteException {
 try {
  ObjectInputStream in;
  in = new SessionObjectInputStream(ctx,
    new FileInputStream(getFile(ctx.getId())));
  ctx.setInstance(in.readObject());
  in.close();
  ejbActivate.invoke(ctx.getInstance(), new Object[0]);
 } // handle errors
}
```

Component Proxy

The COMPONENT PROXY is part of the GLUE-CODE LAYER and is generated by the CONTAINER during COMPONENT INSTALLATION. Its purpose is to hide the physical Bean instance from direct access by clients. This is necessary to support VIRTUAL INSTANCES and to implement INSTANCE POOLING and PASSIVATION. In EJB, COMPONENT PROXIES are used as described in that pattern's description.

The following example is taken from Sun's J2EE Reference Implementation. It shows a fragment of source code that handles incoming method invocations from the COMPONENT BUS and forwards them to the COMPONENT IMPLEMENTATION.

The method handled by the COMPONENT PROXY is an operation *getCity()* of an *Address* Beans business interface:

```
public java.lang.String getCity()
  throws java.rmi.RemoteException{
```

First, an *Invocation* object is created. This is used to collect all information relevant to the invocation, and is subsequently passed to the CONTAINER:

```
  com.sun.ejb.Invocation i = new com.sun.ejb.Invocation();
```

The current EJB object (the COMPONENT PROXY) is then set in the *Invocation* object:

```
  i.ejbObject = this;
```

In the next step, a *java.lang.reflect.Method* object will be retrieved that describes the method that is about to be executed. The method object is then stored in the *Invocation* object, because the CONTAINER will need it later:

```
try {
 i.method =
  de.mathema.mccf.example.Adress.class.getMethod("getCity",
  new java.lang.Class[] {});
} catch(NoSuchMethodException e) {
 // handle
}
```

A variable that stores the return value is then created. The type of this variable must be the same as the return type of the respective method:

```
java.lang.String retVal = null;
try {
```

In the next step, the CONTAINER is informed by the COMPONENT PROXY about the operation to be invoked. The passed *Invocation* object contains the PROXY and the *Method* object of the operation:

```
 this.getContainer().preInvoke(i);
```

During the *getContainer().preInvoke(i)* call the CONTAINER can check security, manage transactions, and finally set the *ejb* attribute of the *Invocation* object to contain, in this example, an instance of *de.mathema.mccf.example.AddressBean*, the COMPONENT IMPLEMENTATION. This is also the point at which the IMPLEMENTATION is prepared by calling appropriate LIFECYCLE CALLBACK operations, in this case *ejbLoad()*.

We have now reached the point at which the actual business operation, the method we implemented when we wrote the Bean, is called. The return value is remembered for later use. If an exception is thrown, it is also stored:

```
  de.mathema.mccf.example.AdressBean ejb =
    (de.mathema.mccf.example.AdressBean) i.ejb;
  retVal = ejb.getCity();
} catch(Throwable c) {
  i.exception = c;
```

It is first necessary, however, to clean up the COMPONENT IMPLEMENTATION object. This is done by the CONTAINER in *postInvoke()*:

```
} finally {
 this.getContainer().postInvoke(i);
}
```

The exceptions of the IMPLEMENTATION object are then handled, and, if no exception has been propagated, the original return value is returned to the caller:

```
  if(i.exception instanceof java.lang.RuntimeException) {
   // ommitted for brevity
  }
  return retVal;
 }
```

To allow remote access to the COMPONENT, the COMPONENT proxy is then attached to the RMI/IIOP subsystem.

There are other, more sophisticated implementations of COMPONENT PROXY and COMPONENT BUS, but this should be enough to illustrate the principles. For an example of a different approach, see the open source implementation of JBoss [JBOSS].

Glue-Code Layer

The GLUE CODE LAYER provides the adaptation from the standardized interfaces – *SessionBean*, *EntityBean* – and specific business interface of a Bean to the vendor-specific structure of the CONTAINER. It is responsible for providing the semantics defined in the specification and ensuring that the ANNOTATIONS are realized.

The GLUE-CODE LAYER consists of generated code that is produced during COMPONENT INSTALLATION and generic code that is part of the CONTAINER. There are alternative ways in which the generated parts of the GLUE-CODE LAYER and the static parts of the CONTAINER can share the responsibilities. The amount of generated code therefore varies from vendor to vendor.

One rather extreme alternative is to generate the whole CONTAINER based on the COMPONENT INTERFACE and the ANNOTATIONS. This generated CONTAINER can use several predefined strategies to which it delegates some of its work. The advantage is that such a solution can be very efficient, because the code can be very tightly coupled with the Bean and the ANNOTATIONS.

The problem with it, however, is that a change in the ANNOTATIONS usually requires re-generation of the GLUE-CODE LAYER, which takes time and limits run-time deployability.

On the other hand, it is also possible to generate only the COMPONENT PROXY, while using a generic framework to provide all the other features. In particular, the ANNOTATIONS can be translated into an object structure that can be accessed relatively efficiently at run time. The framework can then use these structures to provide the correct technical functionality. This approach is less efficient, but it can be managed more easily because the code is not generated. It can also be tested more easily and is more flexible, because the ANNOTATIONS of a Bean can be changed without regeneration. This is the approach taken by Sun’s J2EE Reference Implementation.

Which implementation is best depends on your requirements. If performance and code size are your most important concerns, more code generation is usually better. If you want to create a test-bed for Beans, use an example of the second approach, for example Sun's Reference Implementation.

We won't go into more detail here, because whatever say would be CONTAINER-specific. We have shown some code in the COMPONENT PROXY example. For more information, look at the source code and documentation of open source APPLICATION SERVERS such as *JBoss* [JBOSS] or *openEJB* [OPENEJB].

12 A Bean and its Environment

An EJB Bean should be a self-contained part of a system, as should any COMPONENT – it should be as independent from other Beans as possible. This independence makes reusing Beans possible, which is one of the driving factors of component-based development.

Nevertheless, a Bean has to deal with its environment. It has to communicate with the CONTAINER and sometimes with other Beans. This chapter describes how the interfaces to the outside world look from a Bean's perspective.

Component Context

The COMPONENT CONTEXT is one of two ways in which the COMPONENT can access its environment, used to access and control the CONTAINER and its services. The other is the COMPONENT-LOCAL NAMING CONTEXT, which is a separate pattern.

The COMPONENT CONTEXT allows the Bean implementation to access all external information that changes with the Bean's logical identity – see VIRTUAL INSTANCE – as well as information about the current method's INVOCATION CONTEXT.

To be able to use the COMPONENT CONTEXT, the Bean needs to be able to access it. LIFECYCLE CALLBACK operations called *setEntityContext()* and *setSessionContext()* exist for this purpose. Whenever these operations are called by the CONTAINER, it passes a new COMPONENT CONTEXT object to the Bean. The Bean should store this reference locally if it needs to use the context later:

```
// imports
class SomeBeanImpl implements javax.ejb.SessionBean {
  private SessionContext context = null;
  public void setSessionContext( SessionContext ctx ) {
    context = ctx;
  }
}
```

Make sure that you never store information obtained from the COMPONENT CONTEXT directly in your Bean class. This is because the CONTAINER is allowed to change the contents of the context object without notice, making your stored information out of date. However, you may (and should) store the context object itself as an attribute.

The operations in the COMPONENT CONTEXT can be organized into three groups:

- Operations for transaction management
- Security-related operations

- Operations necessary for COMPONENT INTROSPECTION

All these operations are described in more detail below.

The COMPONENT CONTEXTS are different for Session Beans and Entity Beans. They are called *SessionContext* and *EntityContext* respectively. However, they have a common base interface called *EJBContext*. We will describe all operations together, pointing out where an operation is only accessible in one of the sub-interfaces.

Transaction-related operations

Although the APPLICATION SERVER acts as a transaction coordinator and thus manages transactions for the Beans, there are still situations in which the application programmer needs to be concerned with the transactional status.

setRollbackOnly() is an operation that can be used to mark the current transaction for rollback if you are using Container-Managed Transactions (CMT). This ensures that the current transaction is rolled back, no matter what other Beans would like to see happening. This is useful, for example, if you notice in the Bean implementation that a business constraint fails, for example when trying to debit funds from an empty bank account. In this example you should mark the transaction for rollback, then throw an *AccountEmptyException* to notify the clients:

```
// imports
class AccountImpl implements EntityBean {
  private EntityContext context = null;
  // will be set in setEntityContext()

  public void transaction ( double amount )
    throws AccountEmptyException {
    if ( getCurrentBalance() - amount < 0 ) {
      context.setRollbackOnly();
      throw ( new AccountEmptyException (
        "Account would go negative after transaction" ) );
    }
    // do transaction
  }
}
```

getRollbackOnly() allows you to query the flag mentioned above in the case of Container-Managed Transactions (CMT). If you are about to execute a very computation-intensive operation, you might want to check in advance whether or not the transaction is marked for rollback – because in case of a transaction marked for rollback, your operation would probably be useless.

For example, look at the following code, in which a complex calculation of a discount rate for an order is only executed if the transaction is not yet marked for rollback:

```
// imports
class OrderImpl implements EntityBean {
  private EntityContext context = null;
  // will be set in setEntityContext()

  public double calculateDiscount() {
    if ( !context.getRollbackOnly() ) {
      double discount = 0;
      // calculate discount: includes statistics of
      // the 10 most recent orders for the current customer
      /// --- very expensive to calculate return discount;
    }
    return 0;
  }
}
```

In some circumstances you may want access the transaction object itself, for example if you use Bean-Managed Transactions, or if you want to pass some transaction information to a back-end system called from within your EJB. To do this, you can call *getUserTransaction()* on the context. This operation returns the current *UserTransaction* instance, allowing you to access the transaction status, or even start a new *UserTransaction* for the current thread. Access to the user transaction is only allowed in Session Beans using Bean-Managed Transaction. This is checked in the EJB 2.0 specification.

The following example shows an *OrderManager* implementation that starts its own transaction in which the order handling process is executed:

```
// imports
class OrderManagerImpl implements SessionBean {
  private SessionContext context = null;
  // will be set in setSessionContext()

  public void handleOrder( Order order ) {
    UserTransaction tx = context.getUserTransaction();
    try {
      tx.begin();
      // create order
      // decrease amounts in the warehouse
      // update customer statistics
      if ( problemsHaveBeenSignalled ) {
        tx.rollback();
      } else {
        tx.commit();
      }
    } catch ( Exception ex ) {
        // Here, several types of exceptions must be handled:
      // RollbackExceptions, HeuristicRollbackExceptions
      // or SystemException. In each of these cases
      // the transaction was already rolled back. So a
      // rollback of the transaction is only necessary
      // if an application exception is caught.
      // The detailed explanation is beyond the scope
      // of this book.
    }
  }
}
```

Security-related operations

Security in EJB can be handled declaratively, but only at a method-level granularity – you can specify which user roles are allowed to invoke which operations. You cannot work at a more detailed level using ANNOTATIONS. Instead, you must implement more advanced security concepts programmatically.

A typical situation in which you might want to handle security programmatically is data-dependent security. Imagine that you want to ensure that only the person who created a specific data entity is allowed to modify it later. This has two consequences:

- When you create the entity, you need to store the person (or *principal* in Java security terms) who created it. You therefore need to access the security data of this person in the store operation.

- When a getter/setter operation is called later to modify the instance, you have to make sure that the calling principal is the one who created the entity. Access to the currently-calling principal is again required. A security exception can be thrown in the case in which the caller is not authorized.

You therefore need to access the role or the principal of the caller. To aid in the implementation of these requirements, the COMPONENT CONTEXT offers two operations:

- *getCallerPrincipal()* returns the caller principal object of the current call. This information is actually stored in the INVOCATION CONTEXT, but accessed using the COMPONENT CONTEXT.
- *isCallerInRole(String role)* returns a *Boolean* defining whether the current caller is a member of the specified logical *role*. Note that this uses the mapping from logical roles to groups or users set up during COMPONENT INSTALLATION.

The following code shows an example in which a *ClinicalObservationData* Entity Bean checks that an approval process can only be carried out by the person that created the instance. The creator's *principal* has been stored as part of the instance in its *create()* operation:

```
// imports
public class ClinicalObservationDataImpl implements EntityBean {
  private EntityContext context = null;
  // will be set in setEntityContext()

  private Principal getCreatorPrincipal() {
    // returns the Principal object of the caller who called
    // the create() operation. This principal is stored
    // as part of the bean's state.
  }

  public void approveObservation() throws SecurityException {
    Principal currentCaller = context.getCallerPrincipal();
    Principal creator = getCreatorPrincipal();
    if ( !currentCaller.equals( creator ) ) {
      throw ( new SecurityException
        ( "Approval only allowed by creator!" ) );
    } else {
      // approve instance and update state accordingly
    }
  }
}
```

Note that this cannot be achieved using EJB's declarative security, because the decision on whether to allow the approval depends on the *instance* data of the Bean and not just on the operation itself. The declarative facilities could be used, for example, to ensure that only principals who are part of a specific the group such as *doctors* are allowed to call *approveObservation()*.

Operations supporting COMPONENT INTROSPECTION

Sometimes the Bean instance must obtain information about itself or about its 'logical self'. The COMPONENT CONTEXT provides several operations to support this.

If you want to pass a reference to the current instance to a client or another Bean, you must not use *this*, because *this* denotes the physical Bean implementation object. The physical Bean implementation, as VIRTUAL INSTANCE explains, does not implement the Remote Interface and cannot be accessed remotely. Passing *this* out of the CONTAINER therefore does not make any sense and is an error.

Instead, you need a reference to the COMPONENT PROXY, which really implements the Bean's REMOTE INTERFACE. You can access this object by calling *getEJBObject()* on the context. The object returned by *getEJB-Object()* is one you can return to the client or to other Beans.

The following example returns the higher-valued of two *Order* Entity Beans, using *getEJBObject()* in case the one on which the operation has been called is the higher-valued of the two:

```
// imports
class OrderImpl implements EntityBean {
  private EntityContext context = null;
  // will be set in setEntityContext()

  public Order getHigherValued( Order other ) {
    if ( other.getTotalPrice() > getTotalPrice() ) {
      return other;
    } else {
      return context.getEJBObject();
    }
  }

  // More operations go here. Not shown for brevity.
}
```

It is also possible to access the home of the COMPONENT, by using *getEJBHome()*. There are several reasons why you might want to access the Bean's home – one is that the Home Interface provides access to the associated *EJBMetaData* object, which is useful for COMPONENT INTROSPECTION.

getPrimaryKey() is a very important operation that is only applicable to Entity Beans and is thus only found in the *EntityContext* interface. It allows the physical Bean instance to find out about its current logical identity, denoted by the PRIMARY KEY. In some of the LIFECYCLE CALLBACK operations, such as *ejbLoad()*, the implementation must use this operation to find out which data should actually be loaded from the database. This is because if the logical identity of a physical instance changes just the PRIMARY KEY in the COMPONENT CONTEXT is changed:

```
// imports
class OrderImpl implements EntityBean {
  private EntityContext context = null;
  // will be set in setEntityContext()

  public void ejbLoad() {
    OrderPK pk = (OrderPK)context.getPrimaryKey();
    String query =
      "SELECT * FROM orderTable WHERE id ='+pk.toString()+"'";
    // execute Query
    // set attributes from query result
  }
}
```

Naming

NAMING offers a means of locating references to MANAGED RESOURCES or COMPONENTS by using a structured name. This decouples the client's implementation from the actual location of the COMPONENT. Because a name is used, no knowledge about the physical location of the COMPONENT is coded into the client, and therefore the location of the COMPONENT can be changed without requiring changes in the client's source code.

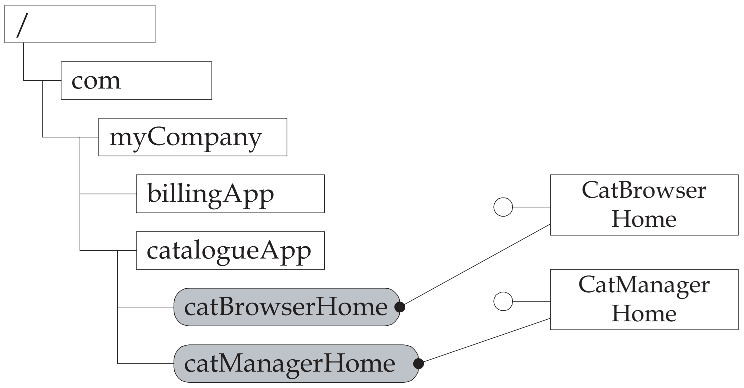
In J2EE, and thus also in EJB, NAMING is accessed through the JNDI. JNDI stands for 'Java Naming and Directory Interface', and is a standardized interface to any kind of naming or directory services. Implementations include adapters for the RMI registry, for the CORBA Naming Service, X.500, and for LDAP services. The main benefit is that the programmer only needs to learn one API and can access all these service through it.

In EJB, NAMING is used primarily to access MANAGED RESOURCES and COMPONENT HOMES.

Contexts and bindings

Generally, NAMING maps a name, usually a structured name, to a reference. Such a mapping is usually called a *binding*. *Contexts* serve to group related bindings. A context can thus contain several bindings, or other contexts. This results in a hierarchical context structure not unlike the directory structure of a file system[1].

1. There are in fact JNDI implementations for accessing the file system.



To access a specific binding in a JNDI service, you need a reference to its parent context. Usually, you start with the root context and navigate to the binding by using path expressions that denote a series of contexts.

For example, refer to the illustration above. If the current context is *catalogueApp*, you can access the binding *catBrowserHome* directly using the name *catBrowserHome*. If the current context is the root context denoted by the slash, then you access the binding using *com/myCompany/catalogueApp/catBrowserHome*. This is the usual way to access a binding, because the root context is usually the starting point. If your current location is another context, you either have to give the absolute path, starting with a leading slash, or navigate amongst contexts using the two dot notation (..) to go one context 'up' the context structure and relative paths to go 'down' it.

Setting up access to the JNDI implementation

The root context is the normal starting point for locating contexts in the NAMING service. This root context is called *InitialContext* in JNDI. The first step in accessing NAMING is therefore to create an instance of *InitialContext*. When creating the *InitialContext* instance, a set of properties has to be specified and passed to the constructor to configure the object appropriately:

```
Properties props = new Properties();
// add properties: see below
InitialContext ic = new InitialContext( props );
```

For a normal application, not for a Bean, the following two properties are the minimum that must be specified:

- *Context.INITIAL_CONTEXT_FACTORY*. Specifies the class name that must be used by the JNDI implementation to construct the *InitialContext* object, for example *com.sun.jndi.ldap.LdapCtxFactory*. This basically denotes the driver for the accessed JNDI implementation technology.
- *Context.PROVIDER_URL*. The URL that the above factory should use to access the respective server, for example:

  ldap://localhost:389/o=jnditest

The following fragment of code shows how such an *InitialContext* is created and how the properties are specified:

```
Properties props = new Properties();
props.put( "Context.INITIAL_CONTEXT_FACTORY",
  "com.sun.jndi.ldap.LdapCtxFactory" );
props.put( "Context.PROVIDER_URL",
  "ldap://localhost:389/o=jnditest" );
InitialContext ic = new InitialContext( props );
```

Initialization parameters can also be specified as JVM command-line parameters by using the *-D* option. There are several other properties, most of which depend on *Context.INITIAL_CONTEXT_FACTORY*. They serve as initialization parameters for the context factory and are beyond the scope of this description.

If you want to access the NAMING of the APPLICATION SERVER in which the COMPONENT runs within a Bean implementation, you don't need to specify any of these properties. This is because the respective JVM has the properties set as environment parameters, and the *InitialContext* instance reads them automatically. Thus, inside a COMPONENT, you can just write:

```
// imports
public class SomeComponentImpl extends SessionBean {
  public void someOperation() {
    InitialContext ic = new InitialContext();
    // use ic
  }
}
```

This makes the code more portable, because no special properties need to be set for the respective JNDI server. The code can thus run on any platform without further modification.

Note that on some APPLICATION SERVERS constructing an *InitialContext* object can take a considerable amount of time. You should therefore create an *InitialContext* only once and store it in an attribute of the instance if you need it again later. Ideally, you should do this at the beginning of the lifecycle of the Bean in the respective LIFECYCLE CALLBACK operation.

Looking up bindings

To look up bindings, just use the *lookup()* operation. The parameter for *lookup()* is a path expression, as explained above. The operation returns an *Object* that must be downcasted to the appropriate type, as demonstrated in the next code fragment:

```
InitialContext ic = new InitialContext( );
String name= (String) ic.lookup("com/myCompany/name");
```

Because of the implications of the underlying COMPONENT BUS, namely its IIOP-based implementation in EJB 1.1 and 2.0, looking up COMPONENT HOMES is a little more complicated. This is because the casting must be done more explicitly, using the *PortableRemoteObject.narrow()* operation:

```
Object o =
  ic.lookup( "com/myCompany/catalogueApp/catBrowserHome" );
CatBrowserHome hm = (CartBrowserHome)PortableRemoteObject.
  narrow( o, CatBrowserHome.class );
```

Component-Local Naming Context

Although NAMING in EJB is always accessed using JNDI, two different kinds of NAMING services are implemented in EJB: the APPLICATION SERVER's global JNDI context and a Bean's local JNDI context. The Bean's local JNDI context is an implementation of the COMPONENT-LOCAL NAMING CONTEXT pattern.

Each Bean can only access its own local context. The entries in the COMPONENT-LOCAL NAMING CONTEXT are links to names in the global context. They do not directly point to a reference, but rather to another entry in the global NAMING system.

As explained in the section on this pattern in Part I, the advantage of using a local context is to improve portability and reduce explicit dependencies on specific global names. In particular, in EJB, the local context can be used as follows:

- Beans can use it to access CONFIGURATION PARAMETERS. The parameters defined in the Deployment Descriptor or during COMPONENT INSTALLATION are placed in the root of the local context. This does make sense, because the configuration is local to one specific Bean and this way name clashes cannot happen.
- MANAGED RESOURCE references should be located in the local naming context. For example *java:comp/env/jdbc* or *java:comp/env/jms* should be used as contexts for JDBC and JMS resources respectively. However, there is nothing that enforces this convention.

  When a Bean is deployed, the mapping is set up between the hard-coded reference to the MANAGED RESOURCE in the local context to the resource's real name in the global context.
- Referenced homes of other Beans are located in *java:comp/env/ejb*. EJB uses the same indirection for references to COMPONENT HOMES as it uses for MANAGED RESOURCES.

It is possible to use other sub-contexts to store local references to other, non-standard resources.

The Bean's local context can be accessed by a look-up operation in the same *InitialContext* as the global context, but using *java:comp/env* as the prefix for every entry in the local naming context. To look up an object with the name *someObject* that is stored in the root of the local context, for example, you have to use:

```
InitialContext ctx = //...
SomeObject = ctx.lookup( "java:comp/env/someObject" );
```

Managed Resource

One of the main reason why component-based systems can offer good performance and scalability is that the CONTAINER provides MANAGED RESOURCES – the CONTAINER is given control over access to all external resources. This allows it to implement clever strategies to optimize access to these resources, such as connection pooling, potentially speeding up the application.

However, there is more to MANAGED RESOURCES than this: to be able to integrate resources into the CONTAINER's handling of technical concerns, such as Container-Managed Transactions or security, for example, the CONTAINER requires control over the resources.

In EJB, MANAGED RESOURCES have three different implications, each of which is described separately below:

- Resources are pooled for efficient resource acquisition
- Resource factories are used to access different resources in a well-defined and consistent manner
- Resources are integrated with the CONTAINER's transaction and security policies

Resource pooling

EJB provides pooling of resources. The pool is accessed by looking up a resource factory object using the logical name of the resource and the COMPONENT-LOCAL NAMING CONTEXT. Pools have to be configured in the APPLICATION SERVER to make them available to COMPONENTS.

While access to resources from within a Bean is standardized to provide portability, the details of connection pool set-up and configuration are different for each APPLICATION SERVER. The APPLICATION SERVER usually provides a GUI tool that assists the administrator in setting up the pool. In the simplest case, the values are entered into a server configuration file manually.

The following fragment of code shows the entries that are necessary to set up a JDBC connection pool in BEA's WebLogic Server 6.0. It uses an XML based configuration file[2]:

```
<JDBCConnectionPool Name="somePool"
  DriverName="sun.jdbc.odbc.JdbcOdbcDriver"
  URL="jdbc:odbc:Test" Properties="user=test;password=test"
  InitialCapacity="5" MaxCapacity="10" CapacityIncrement="2" />
```

This creates a connection pool called *somePool*, which uses the default JDBC-ODBC bridge driver. The database URL is *jdbc:odbc:Test* and the user is defined to be *test*, as is the password. The pool starts with an initial capacity of five connections, which are incremented in steps of two up to a maximum of ten.

In a second step, a *Data Source* for this connection pool is created:

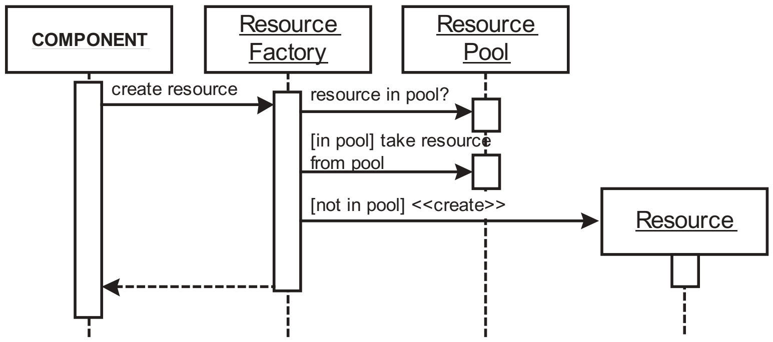
```
<JDBCDataSource JNDIName="jdbc/testpool" Name="test"
  PoolName="somePool" />
```

Here, a new data source is defined, named *test*. The JNDI name of this datasource is *jdbc/testpool*, which means that the COMPONENT can use *java:comp/env/jdbc/testpool* to look up the respective Resource Factory. The datasource is associated with the previously-defined connection pool *somePool*.

Resource factories

A resource factory is a standardized way in which a COMPONENT can acquire access to a resource. The pooling functionality is 'hidden' inside the resource factory. Obtaining a resource from a resource factory is usually a very fast operation, as the resource does not need to be created, just taken from the pool.

2. BEA also provides a web-based configuration tool. The XML configuration file is easier to discuss here.



The sequence diagram above shows how a COMPONENT asks a resource factory to create a resource for it to use. Instead of simply creating such a resource, the factory first looks into a pool of resources, and if one is available, it returns the pooled resource. If not, a new resource is created, put into the pool and subsequently returned.

The following subsections show how these concepts are applied to the different kinds of resources that typically are used inside COMPONENTS, mainly JDBC connections and connections to JMS message queues or topics.

DataSources — resource factories for JDBC connections

The following code fragment looks up a *DataSource* object from the COMPONENT-LOCAL NAMING CONTEXT in the *setEntityContext()* method and uses it in the rest of the implementation to handle Bean-Managed Persistence.

Note that *setEntityContext* is called just after the creation of a new physical instance, while *ejbCreate()* is only called if the new physical instance represents a newly-created logical instance. *ejbLoad()* is called if an existing instance needs to be loaded. So you cannot put this code in these methods. Neither can you put this kind of code in the constructor, because no access to NAMING or the COMPONENT-LOCAL NAMING CONTEXT is allowed there.

```
// imports
public class SomeEntityBeanImpl implements EntityBean {
  private InitialContext initialContext = null;
  private EntityContext entityContext = null;
  private DataSource datasource = null;

  public void setEntityContext( EntityContext ctx ) {
    initialContext = new InitialContext();
    entityContext = ctx;
    datasource = (DataSource) initialContext.lookup(
      "java:comp/env/jdbc/testpool" );
  }

  public void unsetEntityContext() {
    entityContext = null;
    initialContext = null;
    datasource = null;
  }

  public PKClass ejbCreate(...) {
    Connection con = datasource.getConnection();
    // use connection to insert record
    con.close();
  }

  public void ejbLoad() {
    Object somePK = context.getPrimaryKey();
    Connection con = datasource.getConnection();
    // use connection to load data
    con.close();
  }

  public void ejbStore() {
    Connection con = datasource.getConnection();
    // use connection to store data
    con.close();
  }

  // business methods...

}
```

When a COMPONENT has finished using the resource, here the *Connection*, it should return it to the pool as quickly as possible by calling *close()*. *close()* does not really close the connection, but merely returns it to the pool.

ConnectionFactories — resource factories for JMS topics or queues

The concept of resource factories is not limited to JDBC connections – it is available for every type of MANAGED RESOURCE. The Connector API generalizes this concept even further and makes it available for any kind of PLUGGABLE RESOURCE.

Another rather typical example is JMS, the Java Messaging Service. In JMS, you send or receive messages either from a *QueueConnection* (one to one communication) or a *TopicConnection* (publish-subscribe, or one to many communication). JMS provides resource factories for both of these types, called *QueueConnectionFactory* and *TopicConnectionFactory* respectively. The factories are used to access the actual connections, just as in JDBC.

The following example opens a connection to a JMS queue that allows messages to be sent to the queue. Note that while you can send messages to queues or topics from within any kind of Bean, it is not possible with normal Session or Entity Beans to accept incoming messages – this requires the use of EJB 2.0's Message-Driven Beans.

```
// imports
public class SomeStatelessBeanImpl implements SessionBean {
  private InitialContext initialContext = null;
  private SessionContext sessionContext = null;
  private QueueConnection connection= null;
  private QueueSession session= null;

  public void ejbCreate() {
    initialContext = new InitialContext();
    QueueConnectionFactory qconFactory = (QueueConnectionFactory)
      initialContext.lookup(
        "java:comp/env/jms/SomeQueueConnectionFactory” );
    connection= qconFactory.createQueueConnection();
    session = connection.createQueueSession(false,
      Session.AUTO_ACKNOWLEDGE);
  }

  public void setSessionContext( SessionContext ctx ) {
    sessionContext = ctx;
  }

  public void ejbRemove() {
    connection.close();
  }
```

```
    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void someBusinessMethod() {
      // send JMS message here...
    }
  }
```

Other resources

In addition to the resources discussed above, JavaMail sessions and URL Factories can also be defined as resources. With Java Mail sessions (*javax.mail.Session*) it is possible to send and receive e-mails. In this case the server and user account are defined in the Deployment Descriptor. The URL Factory (*java.net.URL*) can be used to load data from an arbitrary source that can be specified using a URL, such as a web page.

To integrate other resources not mentioned here, PLUGGABLE RESOURCES should be used.

Integration with technical concerns

While resource pooling and its standardized access is a common concept that is also used in non component-based systems, an APPLICATION SERVER does more – it integrates the resources with its management of technical concerns. This is only possible because component-based systems use the concept of *Separation of Concerns* – see page 14.

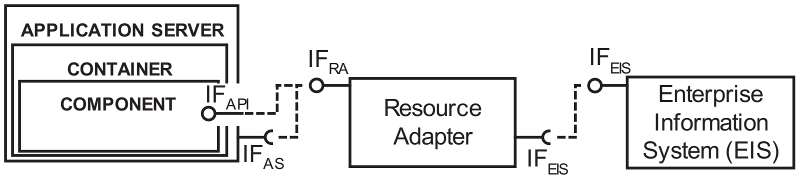
For example, you can specify in the APPLICATION SERVER configuration which users or groups are allowed to access certain resources. If a resource is accessed by any other user or group, the CONTAINER throws a *SecurityException* to the caller.

Another useful feature is the integration with the CONTAINER's transaction management. The CONTAINER must ensure that it begins, commits or rolls back transactions at the right times, according to what is specified in the ANNOTATIONS. While it is not a requirement for EJB systems, most APPLICATION SERVERS can also support a two-phase commit (2PC) protocol if several XA-compliant transactional resources, such as databases, are involved in a single transaction. The APPLICATION SERVER therefore allows transparent handling of transactions covering multiple transactional resources, such as several databases.

Pluggable Resources

PLUGGABLE RESOURCES are not specified at the EJB level, but are instead available for the whole J2EE platform. The corresponding API is called the *J2EE Connector API*. Its goal is to provide integration of any legacy system, such as an enterprise information system (EIS), with the J2EE platform. This includes the provision of a common, generic programming interface, portability of EIS adapters over different J2EE implementations, and integration with J2EE's security, transaction and connection management. All these make the connections to the EIS a MANAGED RESOURCE. Ideally, a vendor of an EIS back-end application has only to provide a suitable *Resource Adapter* implementation to allow the system to be accessed from any J2EE-compliant APPLICATION SERVER.

The basic architecture builds on *ResourceAdapters* between the interface of the EIS (IF_{EIS}) and a generic *ResourceAdapter* Interface (IF_{RA}). The ResourceAdapter interface provides an API (IF_{API}) to the COMPONENTS and a set of interfaces for the APPLICATION SERVER (IF_{AB}).



To make the integration work, the *ResourceAdapter* has several *contracts* with the APPLICATION SERVER: a connection management contract, a transaction contract and a security contract. These contracts are realized as a set of interfaces (IF_{AB}).

The connection management contract basically offers the APPLICATION SERVER a means to connect to the EIS, while also providing an API to implement connection pooling. For the transaction management contract, three different types of support are possible: no transactions, local transactions only, or distributed transactions. In the case of local transactions, a transaction can only include

MANAGED RESOURCES from the EIS, while a distributed transaction might also include other MANAGED RESOURCES such as JDBC connections, JMS queues or connections to other EISs. For a distributed transaction, the ResourceAdapter has to provide some means to support two-phase commit (2PC).

The security contract has two major responsibilities: authentication and encryption. In authentication, either the usual password authentication or a Kerberos-based authentication can be used. The information needed for authentication, such as the password, can be provided either by the COMPONENT or the APPLICATION SERVER. In the latter case the identity of the user can be configured, which means that the connection is built up using the same account on the legacy EIS system.

It is also possible to use so-called *principal mapping*. In this case, each user in the J2EE system might be mapped to a different user in the legacy EIS system. The credentials of the initial caller can also be mapped to different credentials suitable for the EIS. For example, a user is authenticated using a password on the J2EE side and this password is mapped to a certificate for access to the EIS system.

For the COMPONENT programmer, these contracts are transparent, but they provide seamless integration with the APPLICATION SERVER's MANAGED RESOURCES management. The developer therefore does not need to worry about how security of access to the legacy system is managed, or how transactions can cross the boundary to the legacy system – these are all managed by the PLUGGABLE RESOURCE.

The API that is available to the COMPONENT programmer to access these resources has two parts: the specific client API and the optional *Common Client Interface*. The specific client API provides specific connection types to connect to the EIS system. Such a connection is only required to provide a *close()* operation – all other operations can be specific to the particular EIS system.

The *Common Client Interface (CCI)* provides generic interfaces for interaction with an EIS in the following areas:

- *Connections*. The CCI provides interfaces to request a connection, the *ConnectionFactory*, based on a specification, the *ConnectionSpec*, and to represent *Connections*.
- *Interactions*. A client can retrieve an *Interaction* from a *Connection*. The *Interaction* offers an *execute()* operation, passing in an *InteractionSpec*, an input *Record* and an output *Record,* which will be used to store the result. Each EIS provides subclasses of *InteractionSpec*, usually containing information about the function to be executed, for example the name of the function, the execution mode (synchronous or asynchronous) and a timeout value.
- *Data transfer*. Data is transferred in *Records*. EISs either have to define specific subclasses, using the JavaBeans properties conventions, or use a set of predefined *Records*. *MappedRecords* provide a name-value pair representation and *IndexedRecords* provide an array-like structure. Note that these types might themselves contain a *Record*. Complex structures can be built up in this way. *ResultSets* are also offered, which are primarily used for backward-compatibility to JDBC connections.
- *Metadata*. Interfaces to acquire metadata about the connection and the resource adapter are also provided.

The following example shows how the CCI can be used to access a customer management EIS. In the first step, we look up a *ConnectionFactory* and create a *Connection*:

```
InitialContext ctx = new InitialContext();
ConnectionFactory cf = (ConnectionFactory) ctx.lookup(
   "java:comp/env/customerEIS/connectionFactory" );
Connection con = cf.getConnection();
```

Instead of using the *getConnection()* operation without parameters, we could have passed a *ConnectionSpec* object that contained information such as a user name or password, or any other information required to establish the connection.

In the next step, we use the *Connection* to create an *Interaction* object. We use an EIS-specific *InteractionSpec* object to define the request we want to have executed by the EIS:

```
Interaction ix = con.createInteraction();
OurInteractionSpec spec = new OurInteractionSpec();
// EIS-specific
spec.setFunctionName( "getCustomers" );
// EIS-specific
spec.setInteractionVerb( InteractionSpec.SYNC_SEND_RECEIVE );
// Send/retrieve data synchronously.
```

The next step acquires a *RecordFactory* and uses it to create a *Mapped-Record* that specifies the name pattern of the customers we want to find:

```
RecordFactory rf = cf.getRecordFactory();
MappedRecord input = rf.createMappedRecord( "CustomerSearch" );
input.put( "name-pattern", "smith" );
```

Then we execute the interaction:

```
IndexedRecord output = (IndexedRecord) ix.execute( spec, input );
```

In the last step, we iterate through the result *Record* and handle each of its elements. Each element is itself a *Record* of type *Customer*:

```
Iterator it = output.iterator();
while ( it.hasNext() ) {
 Customer cust = (Customer) it.next();
 // handle Customer object
}
```

The CCI provides a generic interface to different EIS systems. This is not the main benefit of the Connector API, however – the main benefit lies in the fact that the resources of the EIS system are integrated into the CONTAINER's security, transaction and connection management. The resource adapters for EISs need to implement a set of interfaces that allow the APPLICATION SERVER to provide such an integration – if the interfaces are implemented correctly, that is. The details are beyond the scope of this book, however.

Configuration Parameters

CONFIGURATION PARAMETERS bring variability to a COMPONENT implementation, by providing a mechanism for passing configuration information into a COMPONENT at deployment time.

In EJB, the CONFIGURATION PARAMETERS are part of the Deployment Descriptor. Their types can be any of the primitive types, using their 'objectified' wrappers such as *Boolean*, *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, or of type *String*. To define the value and type of a property, an entry in the Deployment Descriptor is required:

```
<env-entry>
  <env-entry-name>Currency</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>EUR</env-entry-value>
</env-entry>
```

This example defines the environment entry *Currency* of type *String* to have the value *EUR*, an abbreviation for Euro. To read this entry in the CONTAINER code, something like the following is needed:

```
Context ic=new InitialContext();
String currencySymbol=(String) ic.lookup(
  "java:comp/env/Currency");
```

The point to note here is that the CONFIGURATION PARAMETERS are part of the COMPONENT-LOCAL NAMING CONTEXT, so the access is very similar to the resource factories stored there.

The CONTAINER is responsible for reading these entries from the Deployment Descriptor and making them accessible to the COMPONENTS via the COMPONENT-LOCAL NAMING CONTEXT. Note that each installed Bean has its own COMPONENT-LOCAL NAMING CONTEXT, and therefore the parameters are also specific to one Bean. Consistent configuration of several Beans can therefore become cumbersome – see the CONFIGURATION SERVICE pattern in [VSW02] for information on how to avoid it.

Required Interfaces

REQUIRED INTERFACES list other COMPONENTS that are necessary for a specific COMPONENT to do its job. This is necessary to make it possible for the CONTAINER or the COMPONENT INSTALLATION tool to check whether an application is consistent and installed completely.

In EJB, the REQUIRED INTERFACES are not specified as part of the COMPONENT INTERFACE, but as part of the ANNOTATIONS, the Deployment Descriptor. It is possible to use either a Local or a Remote Interface as a REQUIRED INTERFACE. A reference to such an interface contains the following information for each referenced Bean:

- The name in the COMPONENT-LOCAL NAMING CONTEXT that the COMPONENT IMPLEMENTATION uses to look up the Bean.
- The type of the Bean (*Entity* or *Session*)
- The class name of its COMPONENT INTERFACE
- The class name of its COMPONENT HOME
- A reference to the name of the referenced Bean in the global NAMING (to be defined by the application assembler, upon COMPONENT INSTALLATION)

The COMPONENT implementer declares the first four items in the *ejb-ref* tag. If the REQUIRED INTERFACE was a Local Interface, the *ejb-local-ref* would be used. The respective part of the Deployment Descriptor looks like the following:

```
<enterprise-beans>
 <session>
  ...
  <ejb-name>ASessionBean</ejb-name>
  <ejb-class>com.mycompany.ASessionBean</ejb-class>
  ...
  <ejb-ref>
   <ejb-ref-name>ejb/SomeEntityBean</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>com.mycompany.someEntity.SomeEntityHome</home>
   <remote>com.mycompany.someEntity.SomeEntity</remote>
  </ejb-ref>
  ...
 </session>
 ...
</enterprise-Beans>
```

When the Bean is to be deployed to an APPLICATION SERVER, the deployer defines which other Bean should play the role of the Bean referenced in the code and adds a *ejb-link* element to the descriptor:

```
<ejb-ref>
 <ejb-ref-name>ejb/SomeEntityBean</ejb-ref-name>
 <ejb-ref-type>Entity</ejb-ref-type>
 <home>com.mycompany.someEntity.SomeEntityHome</home>
 <remote>com.mycompany.someEntity.SomeEntity</remote>
 <ejb-link>SomeEntityBeanImplementation</ejb-link>
</ejb-ref>
```

The *SomeEntityBeanImplementation* in the *ejb-link* element is the *ejb-name* of the corresponding Bean's implementation. This approach has two goals:

- First, as described in the REQUIRED INTERFACES pattern, it shows on which other COMPONENT INTERFACES a specific implementation depends.
- Second, it provides an indirection to support portability. Because only the logical name is used in the referencing Bean's implementation, the administrator is free to map it to any Bean on his particular server. Changing the JNDI name of the referenced Bean does not require a change in the implementation of the referencing Bean. This is an example of the benefits of COMPONENT-LOCAL NAMING CONTEXT.

Note that EJB 2.0 container managed relationships are *not* an example of this pattern. Container managed relationships provide run-time relationship management between two Bean instances or collections of instances. The focus therefore is on managing the relations between *instances*. The REQUIRED INTERFACE pattern, alternatively, focuses on the dependencies between different *Beans*, not instances. Its purpose is to enable the CONTAINER to check whether an application is completely deployed and therefore able to run without missing resources.

13 Identifying and Managing Bean Instances

At this point we have already covered a lot of ground – we have COMPONENTS (hence Beans) and know something about their internal structure, we know what a CONTAINER is and what it does, and we know how a Bean interacts with its environment.

There are still some vital points missing, however, the most obvious being that we still don't know how to create or find a Bean instance. This chapter gives details.

Component Home

As described in the COMPONENT HOME pattern, we need something that helps us to create and manage instances of COMPONENTS. This COMPONENT HOME, or *Home Interface* as it is called in EJB, has to be defined for each Bean[1]. This is the place where new Bean instances can be created or old instances can be found.

The Home Interface, as the name implies, is just a declared interface, so no separate implementation class is required. Those parts of the interface that have to be implemented manually – see below – are implemented in the Bean's implementation class, so that we have one COMPONENT IMPLEMENTATION artifact for all parts of the Bean.

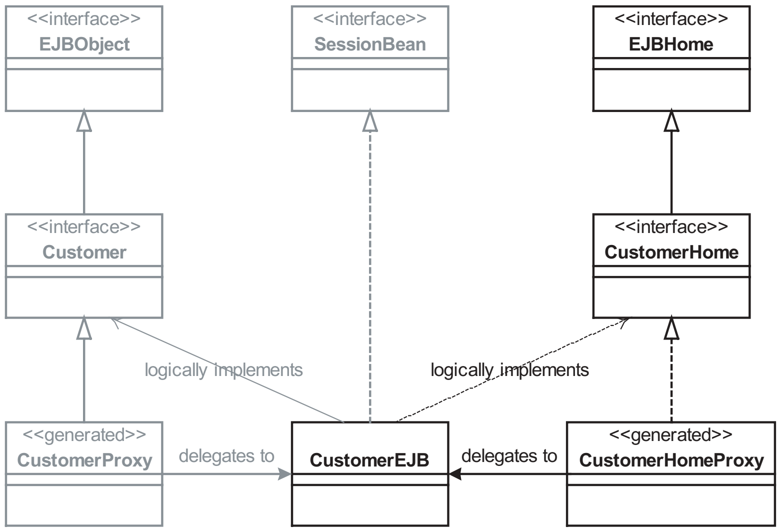
The Home Interface is declared like any other Remote Interface. However, it has to extend the *EJBHome* interface provided as part of the EJB packages:

```
//imports
public interface CustomerHome extends javax.ejb.EJBHome {
}
```

A concrete implementation must be provided somewhere as well, of course. As in the case of the Local and Remote Interface, the Bean implementation does not implement the Home Interface in a Java sense, using the *implements* keyword.

Continuing the Customer example from the COMPONENT INTERFACE section, the relationship of the classes of a *Customer* Bean after deployment is illustrated by the following class diagram:

1. In EJB terms, the interface of the Bean consists of its Home Interface, its Remote Interface and its Local Interface and Local Home Interface. Thus, the Home Interface is formally part of the interface of the Bean. Don't confuse this with our definition of COMPONENT INTERFACE, which only denotes the Local and Remote Interface of an EJB.



As the diagram shows, the Home Interface is 'implemented' in the same way as the Remote or Local Interface[2]. During COMPONENT INSTALLATION, a proxy is generated that implements the Bean's Home Interface. Those parts of the implementation that cannot be generated must be implemented by the developer in the Bean's implementation class, to which the generated proxy delegates the request.

Home operations for different kinds of Bean

Depending on the kink of Bean, different operations can or have to be provided as part of the Home Interface. These are described below.

Stateless Session Beans

For Stateless Session Beans, only one operation is allowed and required for the Home Interface, a *create()* operation without any parameters:

2. A generated proxy provides the 'real' implementation of the interface and the business operations are delegated to the COMPONENT IMPLEMENTATION. For more details, see the COMPONENT INTERFACE pattern example.

```
public interface CalculationServiceHome extends javax.exj.EJBHome {
  CalculationService create()
    throws RemoteException, CreateException;
}
```

This is because a Stateless Session Bean cannot keep client-dependent state – all instances are absolutely equal. Thus it makes no sense to allow a client to pass in information when the Bean is created, because in the next invocation of the Bean the state may be lost – see VIRTUAL INSTANCE and INSTANCE POOLING. Bean configuration that is the same for all instances of a specific Stateless Session Bean can be implemented by accessing CONFIGURATION PARAMETERS.

This *create()* operation must be implemented in the Bean implementation class. According to the naming convention, it must be called *ejbCreate()* and have *void* as the return parameter:

```
public class CalculationServiceImpl implements SessionBean {
  public void ejbCreate() {
    // do any initialization work here
    // in case of problems, throw EJBException
  }
}
```

If you need to throw a technical exception in such a predefined operation, you have to throw the *EJBException*, which is a subclass of *RuntimeException* and therefore does not need to be declared. See the SYSTEM ERRORS pattern for more details.

Stateful Session Beans

Stateful Session Beans can keep client-specific state. It is therefore useful to pass in parameters during instantiation. Consequently, there can be several *create()* operation(s) in the Home Interface, each with a different signature:

```
//imports
public interface ShoppingSessionHome extends EJBHome {
  ShoppingSession create()
    throws RemoteException, CreateException;
  ShoppingSession create( String CustomerID )
    throws RemoteException, CreateException;
}
```

As before, each of these operations has to be implemented. The supplied parameters can be stored as attributes inside the Bean instance:

```
// imports
public class ShoppingSessionImpl implements
  javax.ejb.SessionBean {
    private String customerID = null;

    public void ejbCreate() {
      // initialization work
    }

    public void ejbCreate( String customerID ) {
        ejbCreate();
        this.customerID = customerID;
    }
}
```

Note that even though the Home Interface defines *ShoppingSession* as its return type, the operations still have a *void* return type. It is the job of the CONTAINER and the generated GLUE-CODE LAYER to return the reference of the created Bean to the caller.

Entity Beans

An Entity Bean uses a similar style for *create(…)* operations as Stateful Session Beans. However, there are a couple of differences, mainly because Entity Beans are persistent:

- As with Stateful Session Beans, there can be several *create(…)* operations with different signatures. But for Entity Beans the information passed into the *create(…)* operations must contain enough information to construct the PRIMARY KEY for the Bean[3].
- For each *create(…)* operation in the Home Interface there must be an *ejbCreate(…)* operation in the implementation and an additional *ejbPostCreate(…)* operation, matching the signature of the corresponding *create(…)* operation. The corresponding *ejbPost-Create(…)* operation can be used to do additional initialization work. The difference to *ejbCreate()* is that the instance knows its

3. If the PRIMARY KEY is generated automatically and independently of the instance's attributes, the parameter list can also be empty.

own logical identity and may therefore access the complete COMPONENT CONTEXT, including the PRIMARY KEY, which is not the case in *ejbCreate(…)*.

In addition to the *create(…)* operations that are used to create new (logical) instances of the Bean, the Home Interface must provide operations to find existing instances of the Bean:

- To allow Beans to be re-found, there must at least be one operation called *findByPrimaryKey(…)* that takes the PRIMARY KEY class as a parameter. This should return the Bean instance associated with the supplied PRIMARY KEY.
- There can be additional finder operations, taking other parameters than the PRIMARY KEY. The implementation can use any suitable means to find the corresponding logical instance in a database.

Let's look again at the *Product* Bean. For simplicity, we use *String* as the PRIMARY KEY type:

```
public interface ProductHome extends EJBHome {
  public Product create( String productID )
    throws RemoteException, CreateException;
  public Product create( String productID, String description )
    throws RemoteException, CreateException;
  public Product findByPrimaryKey( String productID )
    throws FinderException, RemoteException;
  public Collection findByDescription( StringdescrWithWildcards )
    throws FinderException, RemoteException;
}
```

The implementation of these operations depends on whether you use Container-Managed Persistence (CMP) or Bean-Managed Persistence (BMP), and of course on the underlying database technology. In case of BMP with a relational database system, you would use a *SELECT* statements to search for the instances and retrieve the attributes, coded manually in the *ejbFind...()* and *ejbLoad()* operations respectively.

The signatures are always the same – although the Home Interface returns the Remote Interface type, the implementations (*ejbCreate(…)* and *ejbFind…(…)* must return the PRIMARY KEY type. The CONTAINER or its GLUE-CODE LAYER takes care of the mapping. Remember that according to the lifecycle of Entity Beans, the finder operations are called before the instance is associated with a logical identity. Only after the finder returns the PRIMARY KEY(s) of the found instances does the CONTAINER retrieve a physical instance from its INSTANCE POOL and associate it with the PRIMARY KEY.

In the case of CMP, you do not even need to implement the operations. They will have empty bodies, because they are implemented by a suitable IMPLEMENTATION PLUG-IN during COMPONENT INSTALLATION. Note that the methods are still called. In EJB 1.1 the IMPLEMENTATION PLUG-IN provides a mechanism, usually GUI-based, to specify any information necessary to generate the finders. This information is specific to the persistence tool and the underlying database technology. In the case of an RDBMS, you have to provide a suitable *WHERE* clause for the *SELECT* statement to be generated.

With EJB2.0, CMP has changed substantially. The finder operations are no longer defined through proprietary descriptions – finders and the new select methods are defined by using EJB QL. For examples based on EJB QL, see the ENTITY COMPONENT pattern, or if you want to read the complete specification, refer to the EJB specification [SUNEJB].

With BMP, more work is required. The following is an example[4] for the *create()* operation using a relational database:

```
class ProductImpl implements EntityBean {
  private String productID;
  private String productDescription;
  public String ejbCreate( String productID, String
    productDescription ) {
    Connection connection = null;
    Statement statement = null;
```

4. It really is just an example, as in a real-world situation it is a good idea to package all database access code in a separate class. This is a so-called DATA ACCESS OBJECT (DAO) – see [VSW02].

```
  try {
       // get connection
       InitialContext ic = new InitialContext();
       DataSource ds =
        (DataSource)ic.lookup("java:comp/env/jdbc/productDB");
       connection = ds.getConnection();
       // insert record into DB using JDBC
       statement = connection.createStatement();
       String strStatement = "INSERT INTO \"product\" VALUES
         ('" + productID + "' , '" +
         productDescription + "')";
       statement.executeUpdate( strStatement );
     } catch(javax.naming.NamingException ex ) {
       ex.printStackTrace();
     } catch(java.sql.SQLException ex1 ) {
       ex1.printStackTrace();
     throw new CreateException(
       "Creation went wrong: SQL Exception"+ex1.getMessage());
     } finally {
       try {
         connection.close();
         statement.close();
       } catch( java.sql.SQLException ex2 ) {
         ex2.printStackTrace();
       throw new CreateException(
         "Creation went wrong: SQL Exception"+ex2.getMessage());
       }
     }
     // initialize Bean
     this.productID = productID;
     this.productDescripion = productDescription;
     // initialize unspecified attirbutes to null ...
     // return primary key
     return productID;
   }
 }
```

Note that the *create(…)* operations really have to insert a suitable record into the database, as the CONTAINER assumes that after creation the instance's state is persistent. It is thus not enough just to initialize the current instance.

The finder operations are not much more complicated than the creators – you do the database lookup, and return the collection of PRIMARY KEYS:

```
// imports
public class ProductImpl implements SessionBean {
  public Collection ejbFindByDescription( String
    descrWithWildcards ) {
    // get data source ...
     try {
      // get connection
      InitialContext ic = new InitialContext();
      DataSource ds =
        (DataSource)ic.lookup("java:comp/env/jdbc/productDB");
      connection = ds.getConnection();
      // insert record into DB using JDBC
      statement = connection.createStatement();
      String strStatement =
        "select pkField from table where description like "
          +descrWithWildcards;
      ResultSet rs=statement.executeQuery( strStatement );
      List result = new ArrayList();
      // iterate over result and add PRIMARY KEYS to result list
      while (rs.next()) {
        result..add(rs.getString(1));
      }
      return result;
     } catch(javax.naming.NamingException ex ) {
      ex.printStackTrace();
     } catch(java.sql.SQLException ex1 ) {
       ex1.printStackTrace();
       throw new EJBException(
         "Search went wrong: SQL Exception"+ex1.getMessage());
      } finally {
        try {
       connection.close();
       statement.close();
      } catch( java.sql.SQLException ex2 ) {
       ex2.printStackTrace();
        throw new EJBException(
          "Search went wrong: SQL Exception"+ex2.getMessage());
      }
     }
    return null;
  }
}
```

A subtle difference between single instance finders and multiple instance finders is that the latter return an empty list if no suitable instances are found, while the single instance finder would throw an *ObjectNotFoundException*.

Local Homes

EJB 2.0 introduced a significant performance optimization technique for accessing Beans. In previous versions, there was only one way to access a Bean – by using its Home and Remote Interfaces – no matter whether the caller was really a remote client or another COMPONENT in the same virtual machine. Because of the more complex implementation, this introduced a significant and unnecessary overhead for local calls to Beans. In EJB 2.0, however, Beans can have additional interfaces that allow efficient local-only access. These interfaces are implemented without using time-consuming remote access facilities.

To allow efficient local access to a Bean, it must provide a Local Home and a Local Interface – see the COMPONENT INTERFACE pattern for details. To create a Local Home, *EJBLocalHome* is used instead of *EJBHome* as the base interface for the Home Interface, and operations must not throw *RemoteExceptions*. Everything else remains unchanged.

Home Business Methods

EJB 2.0 also introduced Home Business Methods. Home Business Methods are methods declared in an Entity Bean's Home interface. Their name must not start with 'create', 'find' or 'remove', and they must throw a *RemoteException*. The Bean provider implements the Home Business Methods in the Bean's implementation class. Just as for operations in the Remote Interface, Home Business Methods also implement business logic. The difference, however, is that Home Business Methods are intended to be used independently of a specific Bean instance[5].

Just as for other operations in the Home Interface, Home Business Methods are called on instances in the pooled state, when the instance does not yet have an identity. Whenever you need to implement functionality that affects all, or a subset of, instances of a

5. Home Business Methods can be seen as 'static methods' for Entity Beans.

specific Entity Bean, you can implement that functionality inside Home Business Methods. While in pre-2.0 EJB you would use a Stateless Session Bean to implement such operations, Home Business Methods are better, because it is more evident that the operations belong to a specific Entity Bean, rather than being an arbitrary collection grouped in a Stateless Session Bean. It also means that all business logic is implemented in one place. This makes the architecture more understandable, reduces the number of Beans and makes the complete system less complex.

Using the Home Interface

Using the Home Interface works in the same way as using any other Remote Interface[6]. Once you have obtained a reference to it, usually using NAMING, you can invoke the specified operations:

```
// get reference using NAMING or another means
ProductHome home =
  Product p = home.create( "car" );

// do something with p
Collection c = home.findByDescription( "*tool*" );
Iterator it = c.iterator();
while ( it.hasNext() ) {
  Product q = (Product)it.next();
  // do something with q
}
```

6. There is one subtle difference. You don't have to use the *narrow* operation, because IIOP is of course not involved when local calls are done. A simple cast is enough here.

Primary Key

The PRIMARY KEY is used to identify a logical business entity uniquely. It is used to look up Entity Bean instances using the COMPONENT HOME[7]. Note that the PRIMARY KEY alone does not completely identify the instance. If you know that the Primary Key is *889842*, for example, you still have to know which COMPONENT HOME you should use. Knowing the type of the Bean is not necessarily enough, because multiple COMPONENT HOMES might provide the same type. For example, person identifiers might be returned by an *EmployeeHome* as well as a *CustomerHome*. Alternatively, the Bean might be installed on several APPLICATION SERVERS, in which case each of the installations would have its own NAMING. Depending on the way in which the system is installed, it is not necessarily the case that an instance created in one of the APPLICATION SERVERS is also available in another.

A PRIMARY KEY in EJB is basically a *Serializable* object. To be more precise, it must be a legal RMI/IIOP value type. Sensible implementations of *equals()* and *hashCode()* must also be provided that conform to Java standards. These requirements are met by *String* or *Integer*, for example. In many applications it is sufficient to use *Strings*, which can easily be stored in a database and searched efficiently.

7. Note that PRIMARY KEYS are not limited to use with Entity Beans. They can also be used if a different approach to managing entities is used, such as in the TYPE MANAGER pattern [VSW02]. The only use of PRIMARY KEYS in standard EJB is in the context of Entity Beans, so we stick to this in the example.

In case a simple *String* is not enough, a typical custom PRIMARY KEY definition class looks like the following:

```
public class PatientPK implements Serializable {
  public String familyName;
  public Date dateOfBirth;

  public PatientPK(String familyName, Date dateOfBirth) {
    this.familyName = familyName ;
    this.dateOfBirth = dateOfBirth;
  }

  public PatientPK() {
  }

  public boolean equals(Object obj) {
   if (obj==null) return false;
   if (obj.getClass()!=PatientPK.class) {
    return false;
   }
   PatientPK pk=(PatientPK)obj;
   if (!pk.familyName.equals(familyName)) return false;
   GregorianCalendar thisCal=new GregorianCalendar();
   thisCal.setTime(dateOfBirth);
   GregorianCalendar otherCal=new GregorianCalendar();
   otherCal.setTime(pk.dateOfBirth);
   if (thisCal.get(Calendar.DATE)!=
     otherCal.get(Calendar.DATE)) return false;
   if (thisCal.get(Calendar.MONTH)!=
     otherCal.get(Calendar.MONTH)) return false;
   if (thisCal.get(Calendar.YEAR)!=
     otherCal.get(Calendar.YEAR)) return false;

   return true;
  }

  public int hashCode() {
   return dateOfBirth.getTime() ^ familyName.hashCode();
  }
}
```

The class uses the *name* and *date of birth* to identify the entity, in this case a patient. It provides two constructors: the one with parameters will be used by client code to construct new instances for look-up, while the other is necessary to make deserialization possible.

It is crucial for the CONTAINER to be able to determine whether two PRIMARY KEY instances of the same type are equivalent, because they then denote the same entity. The *equals()* method must therefore be implemented, and because two *equal* objects must have the same hash code, the *hashCode()* must also be implemented in a corresponding fashion.

A PRIMARY KEY is used to look up the corresponding entity's remote reference by using the respective Home's *findByPrimaryKey(…)* operation. Typically, a client uses the PRIMARY KEY class in the following manner:

```
private Patient getPatient( String name, Date date ) {
 PatientHome home = // ...
 PatientPK pk = new PatientPK( name, date );
 Patient p = null;
 try {
  p = home.findByPrimaryKey( pk );
 } catch ( ObjectNotFound onf ) {
  return null;
 }
 return p;
}
```

If you are using Bean-Managed Persistence, the implementation of the Entity Bean will use the PRIMARY KEY internally at several points in the code:

- In finder operations, it uses database-specific techniques to find the entities that match the finder constraint. The result of these finder operations – if something is found – is either one PRIMARY KEY instance or a collection of PRIMARY KEY instances. It is the CONTAINER's task to take these PRIMARY KEY instances and translate them into actual remote references, which are then returned to the client[8].

  This also means that *ejbFindByPrimaryKey()* takes the PRIMARY KEY as a parameter and returns it unchanged. This sounds like an empty implementation, but in fact you are required to look into the database to determine whether an entity for the provided PRIMARY KEY really exists, and throw an *ObjectNotFoundException* if this is not the case.

8. This indirect approach is necessary to allow the CONTAINER to implement VIRTUAL INSTANCES efficiently.

- *In ejbCreate()* operations, the Bean takes some information about the new entity, and returns the created entity's PRIMARY KEY as a result.
- In the *ejbLoad()* operation, the Bean looks up the PRIMARY KEY in its own COMPONENT CONTEXT and uses it to retrieve the necessary data from the database. This is necessary because, when *ejbLoad()* is called, the Bean does not yet have a logical identity, and is informed about it through its context.

Note that there is no 'magic mapping' from the PRIMARY KEY object to the primary key columns in the table. In theory, they can be completely unrelated, for example the PRIMARY KEY value in the database can be calculated based on information in the PRIMARY KEY object. The methods mentioned above can be implemented to carry out any necessary conversion. An example could be a PRIMARY KEY class with several *String* attributes that are stored in the database in one field. However, in most cases the PRIMARY KEY class and the primary key in the database are identical, typically *Strings* or *Integers*.

For Container-Managed Persistence, things are a little different – the Bean developer does not work directly with the PRIMARY KEY. Depending on the sophistication of the persistence manager used, this might still give you the same possibilities as Bean-Managed Persistence for using a different primary key at the database level as the Bean level.

Note that a PRIMARY KEY is still valid after:

- A new client session has been started
- The server has been restarted or updated
- The database has been migrated
- The COMPONENT has been relocated

This is because it only references some data in the database (not equals, see above), and nothing more. This is different to HANDLES, as described in the next section.

There is glitch in the EJB model: you can query every Bean for its PRIMARY KEY, because *getPrimaryKey()* is declared in the interface *EJBObject*, which is the base interface for every COMPONENT INTERFACE, Entity or Session. However, a PRIMARY KEY is not defined for Session Beans, because they have no identity, this operation throws a *RemoteException*.

The type of the PRIMARY KEY class must be known at COMPONENT INSTALLATION, because several operations in the GLUE-CODE LAYER have the type in their code or in operation signatures. This is why the ANNOTATIONS contain the PRIMARY KEY class name for each Entity Bean.

Handle

When we discussed the PRIMARY KEY in the previous section, we mentioned that the PRIMARY KEY does not contain any information about the COMPONENT's type or HOME INTERFACE – it only identifies a logical instance. You have to use the correct COMPONENT HOME to resolve the PRIMARY KEY to the corresponding COMPONENT instance.

This is different for a HANDLE. A HANDLE is basically a 'serializable reference' for a specific Bean instance. You can call *getHandle()* on any Bean instance, and the return value will be an object that is *serializable* and implements the *Handle* interface. As a HANDLE is serializable, you can store it somewhere, such as a disk file or a database, or you can pass it around, for example via a network, floppy disk, e-mail, or message oriented middleware.

You can call *getEJBObject()* on such a HANDLE, which will return a reference to the original Bean on which *getHandle()* was called. Note that although the behavior may look identical for the three kinds of Bean[9], the term 'original Bean' has a slightly different meaning in each case:

- For Stateless Session Beans, it returns a reference to some instance of the correct type: as they are stateless, all Beans of a certain type are equal.
- For Stateful Session Beans, the *getHandle()* method returns a reference to the Bean that is logically the same. This works only if the Bean has not been removed by the server because it timed out. The same logical instance does not mean it is physically the same Bean, though – the Bean might have been passivated and reactivated in the meantime, or it might have migrated to another cluster node because of the APPLICATION SERVER's load-balancing strategy.

9. Note that there are no HANDLES for Message-Driven Beans, because clients never 'touch' them directly – they just pass messages to a queue or topic that is eventually handled by an MDB.

- For Entity Beans, *getHandle()* returns a reference to an instance that represents the same logical entity as before, with the same PRIMARY KEY.

A HANDLE provides certain guarantees. First of all, it is required to work across different JVMs. It is also required to survive APPLICATION SERVER restarts, at least for Entity Beans and Home Interfaces. Of course, if you delete the resource to which the HANDLE points, for example by deleting the corresponding database record or uninstalling a Bean, the HANDLE becomes invalid.

For Entity Beans, a HANDLE is basically a shorthand for accessing the correct Home Interface and doing a look-up based on the previously-stored PRIMARY KEY, or on an internal identifier for non-Entity Beans. This is how the pseudocode might look for an Entity Bean:

```
public class SomeHandleImpl implements Handle, Serializable {
 // attributes to define the server: ip address,
 // and other params for InitialContext
 // attributes to store name of Home Interface
 // attributes to store primary key object here

 public EJBObject getEJBObject() {
  InitialContext ic = ... // get InitialContext for server
  // on which Component was located before
  Object oHome = ic.lookup(); // look-up with name of
  // home interface
  SomeBeanHome home = // downcast; no problem, because
  // this code is generated
  return home.findByPrimaryKey(...); // use pk as stored in handle
 }
}
```

Note that the HANDLE implementation is generated during COMPONENT INSTALLATION and the implementation is usually part of the CLIENT LIBRARY. As the HANDLE can be sent across the network, the code above must in fact also contain information such as the identity of the APPLICATION SERVER to be used.

If you look at the above implementation, you will see that a HANDLE is not as robust as a PRIMARY KEY. The following circumstances might[10] invalidate HANDLES:

- Moving the COMPONENT from one machine to another
- Using another vendor's APPLICATION SERVER – the generated HANDLE might not be able to contact the new type of server
- Changing the network topology such as firewalls or change of the server's IP address
- Upgrading to newer version of the APPLICATION SERVER
- For Stateful Session Beans, a timeout or a server crash

Taking these limitations into account, HANDLES should never be used for long-term storage, only as a means of temporary storage or 'bridging' transport.

10. Although these points *might* invalidate a HANDLE, this is not necessarily the case. The specification leaves a lot to the implementations, and different APPLICATION SERVERS handle this differently. In fact, the specification only defines the guarantees mentioned above, but of course every APPLICATION SERVER is free to provide additional guarantees.

14 Remote Access to Beans

This book focuses on COMPONENTS that can be distributed over different machines in a network. To achieve this goal, many things have to be done by the component infrastructure, covering simple distributed calls, as described in the COMPONENT BUS pattern, as well as complex security and transaction issues.

For these problems, additional data must be transferred – the INVOCATION CONTEXT. To access the distributed system, the client has to be presented with the CLIENT-SIDE PROXY and the CLIENT LIBRARY. Finally, to keep resource consumption low, CLIENT REFERENCE COOPERATION is needed. The implementation of these patterns in EJB is explained in this chapter.

For these issues EJB relies heavily on available object distribution technologies that we do not cover in detail, in particular RMI (Remote Method Invocation) over JRMP and IIOP. If you need information on these topics, look at the literature section, which lists some good books covering this topic in depth.

Component Bus

In EJB, the communication between clients and the CONTAINER, and thus the COMPONENTS, is based on synchronous remote method invocations for Entity and Session Beans. For these synchronous invocations, EJB uses Java's RMI as its underlying object distribution technology. RMI itself can use a variety of low-level network protocols such as JRMP (Java Remote Method Protocol), IIOP (Internet Inter-ORB Protocol) or even vendor-specific protocols. IIOP is a special case, because it must be supported by an EJB 2.0-compliant system.

Since the release of EJB 2.0 a new Bean type is available, the Message-Driven Bean. Message-Driven Beans are invoked asynchronously and use JMS as a distribution technology.

A synchronous EJB remote operation invocation must not only call methods on objects remotely, it must also transport an INVOCATION CONTEXT. The INVOCATION CONTEXT contains security and transaction information, necessary to allow the CONTAINER to implement technical concerns. Depending on the lower-level transport technology used, this INVOCATION CONTEXT may be embedded in the message natively, such as in IIOP, or transported manually with each remote invocation as an additional parameter, as in JRMP implementations.

RMI has two interesting features. One is that serialized Java objects can be passed 'by value', the other is that an RMI system can be configured to download stubs and other required classes from an HTTP server. This can simplify client deployment, because such files no longer need to be distributed and maintained on the clients, and consequently don't need to be part of the CLIENT LIBRARY.

Using JRMP as the transport protocol

The Java Remote Method Protocol (JRMP) is RMI's native transport implementation. It is a very simple protocol, designed specifically for Java's RMI. Unfortunately, JRMP cannot propagate INVOCATION CONTEXTS implicitly. At first, this makes it seem unsuitable for

COMPONENT-based systems, but this is not really a problem, as there is a simple work-around.

A client never invokes a remote operation directly. Instead stubs are used, which are created by the EJB server during COMPONENT INSTALLATION. These stubs can of course pass the context information explicitly, by adding additional parameters to the operations, or by passing the whole operation information including the INVOCATION CONTEXT in one serialized object. This is invisible to the client – only the stubs need to be concerned about it, but they are generated by the APPLICATION SERVER during COMPONENT INSTALLATION and distributed to the client as part of the CLIENT LIBRARY.

For an example, see the JBoss source code [JBOSS]. Basically, this provides one operation, *invoke()*, for clients to call, which takes a set of parameters:

```
public Object invoke(Object id, Method m, Object[] args,
  Transaction tx, Principal identity, Object credential )
    throws Exception;
```

The parameters describe the target object on which the operation in being invoked (*id*), the data describing the operation itself (*m, args*), as well as transactional (*tx*) and security data (*identity, credential*).

The operation shown here is used by JBoss only for collocated Bean instances. For real remote invocations, a second operation *invoke(MarshalledObject)* exists that is semantically identical. The *MarshalledObject* class contains the arguments from the above method as attributes and is used to simplify serialization.

Using IIOP as the transport protocol

To be compatible with CORBA infrastructures, IIOP has been adopted as the default transport protocol for EJB. To be more precise, every EJB 2.0 implementation must support RMI over IIOP. However, additional CONTAINER-specific protocols might be used. In contrast to JRMP, IIOP can transport transaction and security contexts, but it had to be extended to be able to pass objects by value. The use of IIOP as the transport protocol has not been possible without a slight modification of the programming model. Using RMI with JRMP, a look-up in a JNDI context was done with a regular Java cast:

```
MyInterface object =
  (MyInterface)context.lookup( "/myCompany/myName" );
```

Using IIOP this is no longer possible. A special class, *Portable-RemoteObject*, now has to be used to do the casting, which is a little more complex:

```
Object foundObject = context.lookup( "/myCompany/myName" );
MyInterface object =
  (MyInterface)PortableRemoteObject.narrow( foundObject,
    MyInterface.class );
```

Note that although EJB is formally compliant with CORBA if IIOP is used, there are still significant problems in practice, especially for clients written in languages other than Java. For example, a CORBA client in C++ that calls a Bean implemented in EJB would have to be able to handle the objects that are passed by value, so compatible C++ implementations are necessary. This is also true for the system exceptions used by EJB – see SYSTEM ERRORS.

Other protocols

Because the client uses the stubs in the CLIENT LIBRARY, it is possible to use any other communication protocol beneath RMI as long as interoperability with other APPLICATION SERVERS is not an issue. For example, the BEA Weblogic APPLICATION SERVER uses BEA's proprietary *T3* protocol. Using specialized protocols can have advantages, because the protocol can be optimized for performance, size, QoS concerns, or security. For example, BEA uses this technique to introduce wire-level security into their systems. Another possibility would be to use SSL as basis for the JRMP or IIOP protocol.

As a consequence, this protocol independence opens the EJB world to completely different transport protocols, such as XML/HTTP-based protocols such as SOAP or XMLP (XML Protocol), for example. Although these are a lot less efficient than binary protocols such as IIOP/JRMP, they provide the benefit of portability. Some problems are as yet unsolved for SOAP, however, especially concerning the security and transaction information in the COMPONENT CONTEXT, so the standards for these areas are still in development.

Local invocations

Invocations via networks are time-consuming. In addition to the obvious network latency, the COMPONENT BUS has to do a lot of things to make remote invocations possible. This includes marshalling and unmarshalling parameters and results, as well as handling the INVOCATION CONTEXT. The information passed from a caller to a COMPONENT therefore needs to go through many object instances, which is inefficient. If remote invocation is necessary, though, there is no alternative, of course. There are situations, however, in which this is not required: imagine a COMPONENT that will only be called by other COMPONENTS in the same APPLICATION SERVER, and never by the remote client (see the *Facade* pattern). The facilities provided by the COMPONENT BUS are not needed here, and the performance impact it generates is therefore unnecessary.

To ease this situation, a new kind of COMPONENT access has been introduced in EJB 2.0. The Bean and its Home are given additional *Local Interfaces*, providing separate interfaces for clients located in the same virtual machine (JVM). The CONTAINER can then handle these COMPONENTS differently and use an optimized COMPONENT BUS that is only required to work locally, which can result in a big performance boost.

For more details, see the descriptions of the COMPONENT INTERFACE and COMPONENT HOME patterns.

Asynchronous invocations

The question of whether an invocation should be synchronous or asynchronous should be decided by the COMPONENT BUS, after suitable configuration, as described in the pattern's entry in Part I. This should not be visible to the COMPONENT developer – in an ideal world, the developer should be able to use ANNOTATIONS to configure the COMPONENT BUS to work synchronously or asynchronously, and should not be forced to use implementation that depend on the communication system.

However, this is not possible in EJB – a synchronous/asynchronous decision process is not implemented, and asynchronous access requires a completely different programming model (MDBs and JMS). The only way to implement an asynchronous call is to use Message-Driven Beans, but these are accessed in a completely different way and cannot be invoked synchronously, only sent messages. See the EJB example of COMPONENT INTERFACE.

Implementation options

Different ways of handling requests can be employed on the server side. Sun's J2EE reference implementation [SUNRI] uses RMI/IIOP. The COMPONENT PROXIES are directly exported and operations are invoked through regular IIOP streams. The COMPONENT PROXIES are generated. The following code fragment shows the implementation for an *Address* Entity Bean from the CLIENT LIBRARY's point of view:

```
// imports
public class _Address_Stub extends Stub implements Address {

 // more operations from Address interface

 public String getCity() throws RemoteException {
  try {
   org.omg.CORBA_2_3.portable.InputStream in = null;
   try {
    OutputStream out = _request("_get_city", true);
    in = (org.omg.CORBA_2_3.portable.InputStream)_invoke(out);
    return (String) in.read_value(String.class);
   } catch (ApplicationException ex) {
    in = (org.omg.CORBA_2_3.portable.InputStream)
      ex.getInputStream();
    String id = in.read_string();
    throw new UnexpectedException(id);
   } catch (RemarshalException ex) {
    return getCity();
   } finally {
    _releaseReply(in);
   }
  } catch (SystemException ex) {
    throw Util.mapSystemException(ex);
  }
 }
}
```

Note that this generated stub implements the Remote Interface of the respective Bean, in this case *Address*. The *invoke()* operation highlighted above is interesting – this is where the output request is actually transmitted to the server object. On the server side, a respective server object is available. Its *_invoke()* operation is called, which decodes the request and forwards it to the target object, the COMPONENT PROXY. This is standard CORBA remoting, and the terms *Tie* and *Servant* should be familiar to CORBA developers:

```
// imports
public class _AddressBean_EJBObjectImpl_Tie extends Servant
implements Tie {

 private AddressBean_EJBObjectImpl target = null;

 public OutputStream _invoke(String method, InputStream _in,
  ResponseHandler reply) throws SystemException {
  try {
   org.omg.CORBA_2_3.portable.InputStream in =
     (org.omg.CORBA_2_3.portable.InputStream) _in;
   switch (method.hashCode()) {
    case 2066684435:
     if (method.equals("_get_city")) {
      String result = target.getCity();
      org.omg.CORBA_2_3.portable.OutputStream out =
        (org.omg.CORBA_2_3.portable.OutputStream)
          reply.createReply();
      out.write_value(result,String.class);
      return out;
     }
    // more cases
   }
   throw new BAD_OPERATION();
  } catch (SystemException ex) {
    throw ex;
  } catch (Throwable ex) {
    throw new UnknownException(ex);
  }
 }
}
```

JBoss, on the other hand, uses a different technique. It does not make all the Beans directly available remotely, as in the previous example, but instead employs a specific remote communication object, the

ContainerInvoker, as the entry point of remote calls into the CONTAINER. The *ContainerInvoker* provides two operations:

```
public MarshalledObject invoke(MarshalledObject mi) throws
  Exception;

public Object invoke(Object id, Method m, Object[] args,
  Transaction tx, Principal identity, Object credential )
    throws Exception;
```

Both do the same thing. The first is used for remote invocations, the second for invocations from a collocated Bean – a Bean that is in the same APPLICATION SERVER. A *MarshalledObject* object, as supplied by the CLIENT-SIDE PROXY, contains a byte stream, which represents a serialized *RemoteMethodInvocation* object. The contents of such an object are the same as the arguments in the other operation's signature: the identifier of the target, the method to be invoked and its parameters, security credentials such as the identity of the caller and the transaction context for the call, and transaction information.

Comparing the two implementation options reveals the following differences: the more generic implementation of JBoss can transport invocations for all kinds of Beans, independently of their interface. No code generation is necessary here, and it can easily be used with any other remote transport protocol such as sockets, CORBA – even message-oriented middleware is possible.

On the other hand, the code-generation implementation, as exemplified by Sun's J2EE reference implementation, has the advantage of probably better performance and interoperability with CORBA. Interoperability with other APPLICATION SERVERS using the same standard CORBA communication strategy is also simplified, compared to the generic JBoss implementation.

Invocation Context

The INVOCATION CONTEXT is used to transport additional information from the caller to the receiver with each method call. In the EJB architecture, each method invocation must include information about the current transaction and about the identity of the caller. EJB's COMPONENT BUS can use different implementation protocols for its RMI subsystem, as described in the COMPONENT BUS pattern example.

If the standard protocol of EJB 2.0 (IIOP) is used as the transport protocol, security and transaction information can be transported directly as part of the IIOP message, because IIOP supports these concepts natively. If JRMP is used, this is not as easy, because JRMP just supports normal remote method invocations, although workarounds are possible, as described in the COMPONENT BUS example. A third option is to use a proprietary protocol.

Most APPLICATION SERVERS try to be flexible in their COMPONENT BUS implementation. The simplest approach to this problem is that followed by JBoss. Looking again at the interface of the *ContainerInvoker*:

```
public MarshalledObject invoke(MarshalledObject mi)
  throws Exception;

public Object invoke(Object id, Method m, Object[] args,
  Transaction tx, Principal identity, Object credential )
    throws Exception;
```

These two operations explicitly transport all the necessary data. They are semantically identical – the first variant is just a serialized version of the second for use in remote invocations, whereas the second is used for collocated Beans – Beans in the same Java virtual machine. The underlying protocol therefore executes a normal remote method invocation, without any contexts – it just invokes the *invoke()* operation. The context information (*tx, identity, credential*) is supplied as explicit operation parameters. The CLIENT-SIDE PROXY and the COMPO-

NENT PROXY make sure that this additional information is present and available.

Using this approach allows pluggable protocols. Because the CLIENT-SIDE PROXY part is generated by the APPLICATION SERVER during COMPONENT INSTALLATION, no problem arises from this approach. The downside of course is that the communication is no longer interoperable. However, this is usually not a primary concern, because most systems only use APPLICATION SERVERS from a single vendor. To achieve interoperability among heterogeneous APPLICATION SERVERS, they either all need to use IIOP, or bridges must be used to do the translation. In J2EE 1.3 and EJB 2.0, IIOP is the standard and support is required for it from all J2EE APPLICATION SERVERS.

Client-Side Proxy

In EJB, the CLIENT-SIDE PROXY is basically the implementation of the HOME INTERFACE and REMOTE INTERFACE on the client side, on which clients invoke operations locally that are then forwarded to the APPLICATION SERVER. Depending on the way in which the COMPONENT BUS is implemented, this proxy may be a complete RMI stub on its own, or a very 'lightweight' object that uses a more generic RMI stub for the actual remote communication. The proxy is used to set up the remoting information, the INVOCATION CONTEXT, package it all and forward it to the COMPONENT BUS or to the stub, respectively. The CLIENT-SIDE PROXY is included in the CLIENT LIBRARY – the CLIENT-SIDE PROXY is actually the only piece of the CLIENT LIBRARY that is directly used by the client.

Stubs are a common way of implementing CLIENT-SIDE PROXIES. They are primarily used to access the COMPONENT BUS. Stubs are also used in systems such as CORBA or plain RMI. The proxies defined here differ somewhat, because they also have to transfer the INVOCATION CONTEXT. How this is done is described in the example of the INVOCATION CONTEXT pattern.

Another benefit of using a CLIENT-SIDE PROXY is that it can help to implement the VIRTUAL INSTANCE pattern. Remember that not all logical instances for which an invocation arrives really have an assigned physical instance. The CLIENT-SIDE PROXY will include in the message the identifier of the logical instance for which the request is intended, allowing the CONTAINER to make sure that the instance is actually available – see the JBoss section of the COMPONENT BUS example.

In some APPLICATION SERVERS, the CLIENT-SIDE PROXY is also used to assist with load-balancing and fail-over. If several identical server instances are available, the CLIENT-SIDE PROXY can use round robin or another load-balancing scheme to forward subsequent requests to each instance in turn. If it cannot communicate with a specific instance, it could fail-over to another instance.

The following section of source code shows a concrete implementation of a proxy. We use an *Address* Bean implemented with the J2EE reference implementation as an example. The only responsibility the stub has is to forward calls to the server through the COMPONENT BUS. The generated stubs extend *javax.rmi.CORBA.Stub*. This class and its superclasses define the necessary methods to implement the communication with the implementation of the object on the server, using the COMPONENT BUS in a CORBA-compliant manner. The implementation of a *getCity()* operation in the stub is as follows:

```
public class _Adress_Stub extends Stub implements Adress {
  // ...
  public String getCity() throws RemoteException {
    try {
      org.omg.CORBA_2_3.portable.InputStream in = null;
      try {
        OutputStream out = _request("_get_city", true);
       in = (org.omg.CORBA_2_3.portable.InputStream)_invoke(out);
        return (String) in.read_value(String.class);
      } catch (ApplicationException ex) {
        in = (org.omg.CORBA_2_3.portable.InputStream)
          ex.getInputStream();
        String id = in.read_string();
        throw new UnexpectedException(id);
      } catch (RemarshalException ex) {
        return getCity();
      } finally {
        _releaseReply(in);
      }
    } catch (SystemException ex) {
      throw Util.mapSystemException(ex);
    }
  }
  // ...
}
```

The code is fairly straightforward. First, a CORBA output stream is constructed using the *_request()* operation defined in a superclass. The actual operation is then invoked using the *_invoke()* operation. This returns an *InputStream* that is used to read the return value and return it to the caller. The rest is exception handling.

Client Library

The CLIENT LIBRARY has the primary task of enabling clients to communicate with the COMPONENTS in the CONTAINER. Because it is up to the CONTAINER to choose the concrete COMPONENT BUS protocol, clients need a way to attach to it. In other words, code to support access to the specific COMPONENT BUS is needed. As each J2EE implementation might have a different COMPONENT BUS, these classes usually differ between server vendors[1].

Typical library contents also include the Remote and Home Interfaces, the generated stubs, and all necessary helper classes such as operation parameters and exceptions. However, the CLIENT LIBRARY does not contain information that determines to which APPLICATION SERVER the *InitialContext* points – the client has to specify this explicitly during program start-up or through environment entries.

Sun's J2EE Reference Implementation's client library contains the following files:

- *Remote Interface*, *Home Interface*. Both interfaces are necessary for the client, because client programmers need them to downcast their references.
- *Stub*, *Home Stub*. The stubs are the client-side implementations of the interfaces, returned whenever a client requests a reference to a remote object. They take care of remote communication by providing connectivity to the COMPONENT BUS. This is the implementation of CLIENT-SIDE PROXY.
- *Exception classes*. The exceptions used in the signatures of the Interface operations are contained in the library. They are required so that exceptions can be thrown on the client side.
- *Parameter classes*. All the classes used as parameters in Bean operations are also included, with the exception of classes that are

1. Note that every J2EE implementation must support IIOP as a protocol. However, this does not imply that this protocol is really used by clients, as other protocols might be supported as well.

part of the standard JDK or otherwise available through the *classpath* settings of the client.

Note that some of the classes in the library are not only necessary at run-time to enable communication, but also at compile time, to make sure the client program can be typed strongly. These types are, however, only the interfaces, for example the Remote and Home interface and the exceptions and parameter classes. The concrete implementations – the stubs and home stubs – can be loaded dynamically at run-time.

A further interesting possibility arises from the dynamic class loading capability of RMI: if the client does not have access to the required classes in the CLIENT LIBRARY, they can be downloaded from the APPLICATION SERVER dynamically. To make this work, an HTTP server that serves the class files must be provided and the client needs to specify it as the *codebase*. Security settings must also be adapted.

Client Reference Cooperation

CLIENT REFERENCE COOPERATION is used to help the CONTAINER manage the lifecycle of its COMPONENTS. In particular, it is used to inform the CONTAINER when a client no longer needs an instance.

The VIRTUAL INSTANCE pattern is again important here. Because the client never accesses a physical instance directly, the client only needs to tell the CONTAINER when it no longer needs a logical instance. As before, the situation is different for the three Bean types:

- For Stateless Session Beans, this pattern is not necessary. A logical instance lives only for one method invocation. After the invocation, it is logically dead. The physical instances are pooled, that is, a fixed number of physical instances exist. The CONTAINER can dynamically adjust the number of physical instances depending on the load, at least within certain, administrator-defined limits.
- For Stateful Session Beans, the logical instance will be active until either a client calls *remove()* on the Remote or Local Interface or a timeout occurs. In both cases, the instance is removed from memory, or, if currently passivated, its state is removed from persistent storage. The Bean is then inaccessible to clients.

  If the CONTAINER runs into resource problems while the Bean must be kept alive, the CONTAINER is free to PASSIVATE the instance and activate it again when required.
- For Entity Beans, the client calls *remove()* on the Remote or Local Interface when the logical instance is no longer needed. The entity is then removed from the database and the current remote references become invalid. Internally, for physical instances, the server uses INSTANCE POOLING, which means it will only keep those instances active that are actually used.

Note that no real distributed garbage collection problem exists in EJB. For Stateless Session Beans, garbage collection is pointless anyway. It only makes sense to delete all instances in the INSTANCE POOL if the services of a Stateless Session Bean will never again be required. Less frequently-used Stateless Session Bean instances can also be deleted if the CONTAINER runs out of resources. A strict garbage collection process is therefore not necessary.

For Stateful Session Beans, distributed garbage collection is not necessary, because Stateful Session Beans are defined to be accessed only by one client. If the client no longer needs the Bean, it can be discarded. A reference counter to determine whether *nobody* needs the instance any more is unnecessary. It could be argued that Stateful Session Beans use a kind of lease to solve the garbage collection problem – they are deleted if they are not used for a time longer than the set timeout. In other words, they will be deleted if not used frequently. This is much like an object that has a lease that must be renewed regularly.

Entity Beans are logical singletons, and removing an Entity Bean is considered to be business logic – if one client removes it, the entity is gone for all clients. Garbage collection thus makes no sense for the logical entities. However, the physical instances of the Entity Beans, temporarily assigned to the logical entities, might very well be removed if they are no longer needed. In this case the number of concurrently held instances can be regulated much as for Stateless Session Beans – if the CONTAINER runs out of resources, it can delete some of the instances, and if many concurrent request occur for the same type of Entity Beans, more instances can be added. Therefore, again, no strict garbage collection is needed.

15 More EJB Container Implementation

We have already looked at how an EJB CONTAINER works internally and what this means for Bean developers. In this chapter we complete the picture and look at some missing points. From a Bean developer's perspective, the most important issue might be how the container handles errors, so we start with that.

System Errors

In EJB there are basically two kinds of errors, as described in the SYSTEM ERRORS pattern:

- Errors that result from problems in the business logic (application errors)
- Errors that result from technical problems (system errors)

Note that the distinction between these two kinds of error resembles the SEPARATION OF CONCERNS, as implemented by the CONTAINER. The two kinds of error have completely different characteristics. Let's first have a look at system errors.

System errors

System errors are usually caused by the CONTAINER or have other technical sources. The following are typical examples:

- A low-level database problem occurs when trying to communicate with the database from within a Bean
- The CONTAINER runs out of resources
- A network communication problem occurs when the client communicates with the APPLICATION SERVER

These kinds of error may require action by the CONTAINER. The client can still catch the respective exceptions and react accordingly, but before that the CONTAINER must be given a chance to deal with the problem.

These kinds of errors can happen at any time, during any method invocation on any Bean instance. This has two consequences: first, there has to be a kind of generic *system exception* that must always be caught by clients. Second, because this type of exception can happen in any method implementation, it would be tedious to declare it in each implementation method.

RemoteExceptions

Every operation declared in the COMPONENT INTERFACE must contain a declaration for a *RemoteException* in its *throws* clause. This exception must be caught by the client code. It plays two distinct roles:

- Throwing a *RemoteException* to signal that the CLIENT LIBRARY has encountered a network communication problem with the APPLICATION SERVER, without any involvement of the remote APPLICATION SERVER itself.
- Delivering the *EJBExceptions,* raised by the CONTAINER for technical reasons. Whenever the CONTAINER runs into problems that have to be communicated to the client, the CONTAINER wraps the root cause exception in a *RemoteException* and propagates it to the client.

All operations in the Remote Interface therefore have to declare the *RemoteException*. The code below gives details for EJB 2.0 Local Interfaces:

```
public interface Account extends EJBObject {
 void credit( double amount ) throws RemoteException,
  InsufficientBalance;
}
```

The client code has to catch the exception and handle it:

```
try {
 account.credit( 1000.00 );
} catch ( InsufficientBalance neb ) {
 double balance = account.getBalance();
  //...
} catch ( RemoteException rex ) {
  // handle it...
}
```

Note that it is not always simple to handle such a problem in the client. For example, what should the client do when the APPLICATION SERVER has problems with its database?

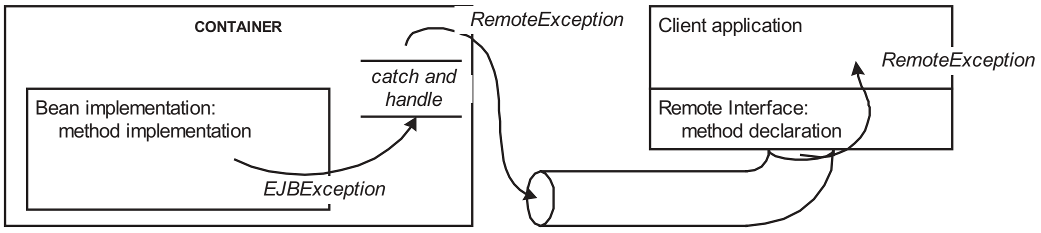
Typically, there are a few options:

- Notify the user and do nothing more
- Retry the operation, hoping that the APPLICATION SERVER has solved the problem
- In an environment where several (similar) APPLICATION SERVERS are available, try another

EJBExceptions

Technical errors can happen at any time, not just with client communication with the remote APPLICATION SERVER, but also in the method implementations. Examples include a database exception, a JNDI exception or problems with a message queue.

To avoid having to declare a *RemoteException* in every method's implementation, and to allow the CONTAINER to intervene, a different approach is taken here – the implementation throws an *EJBException*, wrapping the original exception:



EJBException is a subclass of *RuntimeException*, which means that it does not need to be declared in the *throws* clause of the operations. The CONTAINER can still catch it and act accordingly. Whenever an implementation throws an *EJBException*, the CONTAINER aborts the current transaction. It also makes sure that no further operations are invoked on the instance that threw the *EJBException*, and cleans up resources – the Bean instance is discarded. It then propagates the exception to the caller: if the caller had initiated the transaction, it propagates a *TransactionRolledBackException*, otherwise a *RemoteException*. The CONTAINER usually also logs the exception, making sure that the administrator is informed so that they can solve the problem.

If an *EJBException* is thrown in a method implementation that has been invoked through a Local Interface, the *EJBException* is just propagated to the caller – which must be another Bean, because only they can invoke Local Interfaces. If this operation has been invoked locally, the exception is further propagated. If it has been invoked through a Remote Interface, it is handled as described above – the CONTAINER catches, handles, and propagates a *RemoteException*. This is why operations in Local Interfaces need not (and must not) throw a *RemoteException*.

Note that if a Bean catches a *RemoteException* from another Bean, it can re-throw the exception embedded in an *EJBException*. Thus even a Bean that only has a Local Interface can propagate the encapsulated *RemoteExcetions*. However, they are not propagated automatically, they have to be wrapped manually by the developer.

Application errors

Application errors are errors that can be explained in terms of the functional requirements of the system. Typical examples of application errors might be:

- In a *withdraw()* operation in a financial application, an account balance is lower than the requested amount. The amount therefore cannot be debited. The client has to be informed about this error condition and must resolve the problem.
- You want to send an e-mail to an employee. However, no e-mail address is specified for the employee. Again, the client/user has to be notified about this error condition, and again it must provide a solution.

Application errors are normal exceptions, declared as a subclass of *java.lang.Exception*. Operations are declared to throw the exceptions in the Remote Interface. Look at the following example:

```
public class InsufficientBalance extends Exception {
 public InsufficientBalance() {
  super();
 }
 public InsufficientBalance( String message ) {
  super( message );
 }
}

public interface Account extends EJBObject {
 void withdraw( double amount ) throws RemoteException,
  InsufficientBalance;
}
```

Because these exceptions are part of the business logic, they are not handled by the CONTAINER. In particular, throwing such an exception has no consequences on the current transaction – it is not rolled back automatically. If you need to make sure that the transaction is rolled back in such a case, you have to call *setRollbackOnly()* on the COMPONENT CONTEXT. This is because the CONTAINER cannot be expected to know what to do if such an application error occurs. The client therefore has to catch it and react accordingly.

In many cases, the client application is the only 'authority' that is able to determine a useful reaction, for example by querying the user for a resolution strategy. In the case of our financial example, a maximum possible amount might be permitted:

```
Account account = // ...
try {
 account.credit( 1000.00 );
} catch ( InsufficientBalance neb ) {
 double balance = account.getBalance();
 if ( balance > 0 ) {
  try {
   account.credit( balance );
  } catch ( InsufficientBalance neb2 ) {
   // cannot happen here, but has to be caught...
  }
 }
}
```

Predefined application errors

There is a set of predefined exceptions that are thrown by specific methods:

- A *findByXXX()* operation has to include a *FinderException* in its *throws* clause to signal an application error in the finder method, such as an illegal search condition specification. An *ObjectNotFoundException*, a subclass of *FinderException*, can be thrown by single-object finders to specify the case in which no suitable instance was found.
- A *create()* operation has to throw a *CreateException* to signal the case that something has gone wrong when the instance was created. Entity Beans can also throw the more specific *DuplicateKeyException*, to signal that the Bean was not created because the supplied PRIMARY KEY was already available in the database.
- *remove()* operations have to declare a *RemoveException*. For example, a *RemoveException* could be thrown by an *Account* instance to signal that a non-closed account cannot be removed.

These exceptions are merely defined to set a standard for certain types of errors that might occur in these standard methods, so they are predefined, but are otherwise no different from other application errors. The transaction is not marked for rollback if one of these exceptions occurs.

Component Introspection

EJB is based on Java, which provides reflective features as part of the language. Using the classes and interfaces in *java.lang.reflect* it is possible to query a class object for:

- Its attributes – visibility, name and types
- Its operations – visibility, name, return type, parameters
- Its superclass
- The interfaces it implements

This serves as a good base on which to implement COMPONENT INTROSPECTION in EJB. A deployment tool can use these Java features to easily access this information. For example, in the J2EE Reference Implementation [SUNRI], the *deploytool* introspects the COMPONENT and provides a GUI to the deployer that allows the selection of security settings or transaction attributes for specific operations. The tool already presents the list of operations to the user, because it queries them from the COMPONENT, and generates the Deployment Descriptor based on user input.

When the COMPONENT is actually installed, the CONTAINER can validate the ANNOTATIONS against the COMPONENT IMPLEMENTATION and can detect if, for example, an operation has no transactional attributes defined.

Some of this information is available to clients at run time. They can call *getEJBMetaData()* on the Remote Interface, receiving an instance of *EJBMetaData*. This object provides the following operations with obvious meanings:

- *getEJBHome()* returns the reference to the instance of the Home that is responsible for this particular instance
- *getHomeInterfaceClass()* returns the class of the Home Interface
- *getPrimaryKeyClass()* returns the class that is used to construct PRIMARY KEYS

- *getRemoteInterfaceClass()* returns the class of the Remote Interface
- *isSession()* returns *true* if the current Bean is a Session Bean
- *isStatelessSession()* returns *true* if the current Bean is a Stateless Session Bean

Note that the results of these operations, such as whether a Session Bean is stateless, cannot usually be obtained by using Java reflection. They can only be obtained from the Deployment Descriptor and represent the properties declared there. However, this is still COMPONENT INTROSPECTION, because it provides data about the COMPONENT – the data is just from a different source.

At Bean run time the situation is a little different. As detailed in the section on IMPLEMENTATION RESTRICTIONS, Java's Reflection API must not be used freely on the server side. Access to private fields, for example, is forbidden. However, these implementation details should not be relied on by other COMPONENT IMPLEMENTATIONS anyway.

Implementation Plug-In

There is no standard in EJB to support plugging in arbitrary code generators. However, there is one use of IMPLEMENTATION PLUG-INS in EJB environments that is typical – to create code to support persistence for CMP Entity Beans. Whereas Bean-Managed Persistence requires the programmer to write the persistence code, in Container-Managed Persistence (CMP) this job is handled by a suitable IMPLEMENTATION PLUG-IN in the CONTAINER.

To manage persistence for Entity Beans, the following lifecycle operations have to be implemented:

- *ejbCreate(…)* creates a new entity in the database
- *ejbStore()* stores the current entity in the database
- *ejbLoad()* loads the state of the current entity from the database
- *ejbRemove()* removes the current entity from the database
- *ejbFindByXXX(…)* finds entities in the database and returns PRIMARY KEYS

In the case of BMP, the programmer has to implement these operations in the COMPONENT IMPLEMENTATION. Usually, the programmer uses MANAGED RESOURCES to access the database and writes the database code manually. The CONTAINER calls these operations whenever the lifecycle of the instance requires it.

In the case of CMP, the programmer can simply leave these operations unimplemented. The functionality for access to the persistent store is either directly implemented in the GLUE-CODE LAYER or by a separate class to which the GLUE-CODE LAYER delegates.

Ideally, the interfaces between the CONTAINER and the persistence code should be standardized – however, they are not. Persistence providers and CONTAINER therefore have to be adapted specifically to each other.

In EJB 1.1, the Bean programmer uses a CONTAINER-specific tool to inspect the COMPONENT IMPLEMENTATION and provide persistent storage for its fields. The fields that should be stored persistently are declared in the ANNOTATIONS. Note that the EJB 1.1 specification does not specify how the data should be stored, which is up to the APPLICATION SERVER. In addition, the finder operations' semantics have to be specified, for example by providing the necessary SQL-fragment (usually, a *where* clause). Again the exact mechanism is not specified.

In EJB 2.0 the persistence model has been changed significantly. A COMPONENT IMPLEMENTATION class of a Bean that uses CMP must be declared *abstract*. The persistent fields are not declared at all – the programmer instead must declare abstract getter/setter operations according to the JavaBeans convention. The programmer must specify the persistent fields with *cmp-field* elements in the Deployment Descriptor. The CONTAINER will then create a concrete subclass and implement the abstract operations accordingly.

To implement finders, EJB 2.0 provides EJB QL, the EJB Query Language. This language provides a CONTAINER-independent way to specify queries for entities in finder operations. Its syntax resembles SQL or OQL.

In addition, EJB 2.0 provides facilities to specify relationships among Entity Beans declaratively. EJB 2.0's features described above provide complete CONTAINER-independent CMP. It is even independent of the type of storage engine used by the CONTAINER. This makes Entity Beans fully portable across different CONTAINERS.

16 Bean Deployment

At this stage, we have almost completed the discussion about implementing the patterns using EJB. The only missing piece is the deployment and packaging of COMPONENTS. In this chapter, we take a deeper look at how deployment works with EJB.

Component Installation

The COMPONENT INSTALLATION is usually done with a command-line or GUI-based tool delivered with the CONTAINER. This tool is often also used for COMPONENT PACKAGING. For this example, we will again use the J2EE Reference Implementation.

After you have created a COMPONENT PACKAGE or imported an existing one into the APPLICATION SERVER, you can install the contained COMPONENT, usually done by choosing a menu item. You can also choose whether a CLIENT LIBRARY should be generated and set NAMING parameters. The installation itself then takes place: – the COMPONENT PACKAGE is transferred to the APPLICATION SERVER and the GLUE-CODE LAYER is generated. All of this is then stored in a repository for installed COMPONENTS.

If you look behind the scenes into this repository, you can find all the generated and installed files. For a *ShoppingCart* Bean, for example, three files would be generated:

- A file with a name such as *ShoppingCart1000581589562.jar*. This file is almost the same as the COMPONENT PACKAGE, but includes a J2EE-RI specific Deployment Descriptor that specifies the security information for the environment. This file is *sun-j2ee-ri.xml*, and can be found in the *meta-inf* directory. The original COMPONENT PACKAGE is also included.
- A file with a name such as *ShoppingCart1000581589952Server.jar*. This includes the generated GLUE-CODE LAYER, containing the stubs that are used by clients, *_ShoppingCart_Stub.class* and *_ShoppingCartHome_Stub.class,* both located in the directory *shoppingcart*. This *.jar* file also contains the files required for server-side communication, *_ShoppingCartEJB_EJBObjectImpl_Tie.class* and *_ShoppingCartHomeImpl_Tie.class*. This is not too different from what you would expect from a CORBA IDL compiler, for example. The 'real' GLUE-CODE LAYER is located in the files *ShoppingCartEJB_EJBObjectImpl.class* and *ShoppingCartHome-Impl.class*. The first is the class that implements the interfaces

required by the Remote Interface and forwards the calls to the COMPONENT IMPLEMENTATION, the second is the implementation of the Home Interface.

- A file called *ShoppingCartClient.jar*. This includes everything that is needed by the client: the interface classes, the stubs and all other helper classes necessary for the client, for example exception classes. For the Sun J2EE reference implementation [SUNRI], it also includes the COMPONENT IMPLEMENTATION, the Deployment Descriptor and the original COMPONENT PACKAGE, even though this is not needed by the client.

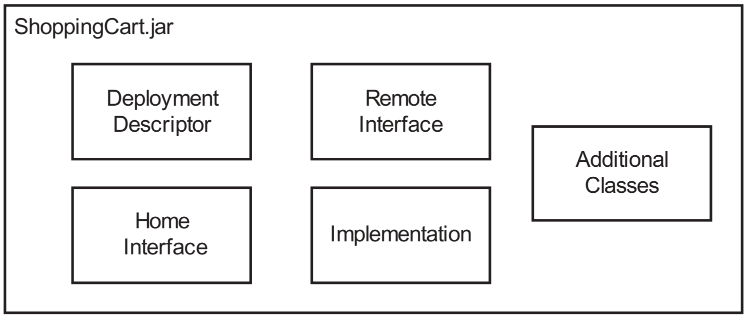
What is not visible here is the set-up of NAMING and MANAGED RESOURCES. These are prepared by the CONTAINER and are not represented in the file structures.

Component Packaging

The packaging format for an EJB is a *.jar* file. This is a *.zip* archive that includes a *manifest* file. As well as listing the contents of the *.jar* file, this file also contains additional metadata about the included files. The directory structure for the *.jar* file is also predefined.

Java's JAR command-line tool can be used to create such files from a set of individual files. However, each CONTAINER usually comes with a GUI tool that can also create *.jar* files. In some cases, the development environment also contains such a tool. These more specific component-packaging tools 'know' that they are about to create a *.jar* for an EJB and can thus carry out additional tests on the validity of the package, making them preferable to the command-line JAR tool.

We will use the *ShoppingCart* example COMPONENT to take a closer look at the *.jar* file. Conceptually, the COMPONENT PACKAGE of this COMPONENT has the structure shown in the figure[1]:



The *ShoppingCart.jar* file contains the following parts:

- *Deployment Descriptor.* Located in the *meta-inf* directory along with the manifest file, this file *ejb-jar.xml* includes the information described in the ANNOTATIONS. Other EJB implementations

1. Note that in this example the *ShoppingCart* Bean supports only Remote Interfaces. If Local Interfaces are needed, these would also be part of the *.jar* file.

might also contain an APPLICATION SERVER-specific Deployment Descriptor in the same directory.

- *Remote Interface*. The Remote Interface is the class *ShoppingCart* located in the *shoppingcart* package. It is available as *ShoppingCart.class* in the *shoppingcart* directory. This directory is used because the class is placed in the package *shoppingcart*.
- *Home Interface*. This is represented by the file *ShoppingCartHome.class* in the *shoppingcart* directory.
- *Implementation*. The implementation can be found in *ShoppingCartEJB.class,* also in the *shoppingcart* directory.
- *Additional Classes*. In our shopping cart example, the class *Product-Description* is used to represent a product. This is used in the Remote Interface as well as in the implementation, and is located in the file *ProductDescription.class* in the directory *shoppingcart*. Other classes that are either used in one of the interfaces or in the implementation are also included. You might also include other resources such as text files or icons.

In this example the shopping cart has only a Remote Interface. A Local Interface and Local Home Interface must of course also be part of the *.jar* file.

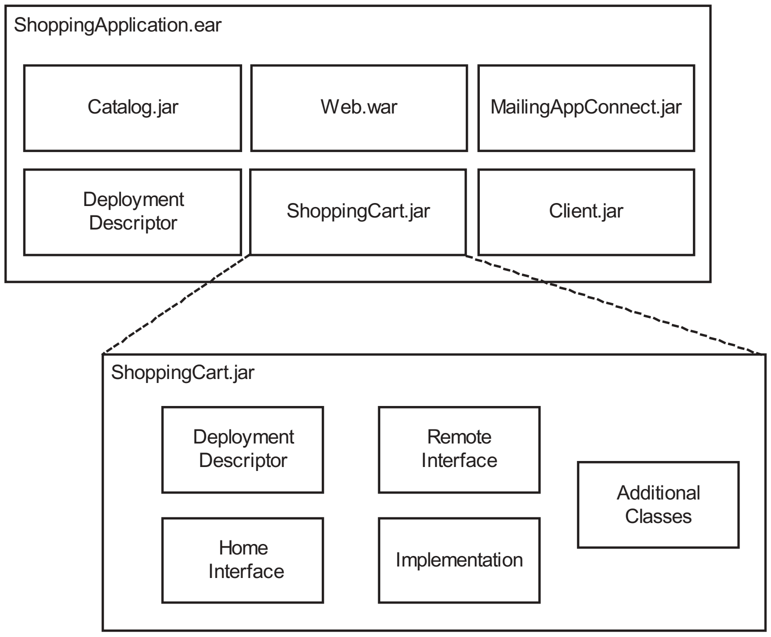
Note that a *.jar* file can contain more than one COMPONENT. In this case, multiple Remote Interfaces, Home Interfaces and implementations will obviously be present. However, there is still only one Deployment Descriptor, which includes entries for all Beans.

Assembly Package

Strictly speaking, an ASSEMBLY PACKAGE is not part of EJB. Instead, J2EE provides *.ear* (Enterprise Archives) files. As with *.jar* files, these are *.zip* files with a different extension to simplify discrimination. They include other archives and a Deployment Descriptor.

Because EJB is a part of J2EE, EJBs can be included in *.ear* files by including their *.jar* COMPONENT PACKAGE files in the *.ear* file. Besides EJBs, other types of COMPONENT can also be included in *.ear* files, including for example *.war* (Web Archives) files. These are used to package Servlets, JSPs and other web-related files such as images or static HTML pages. Other possible file types are application clients – the part of the distributed application that runs on client computers – and Resource Adapters, as described in the example for PLUGGABLE RESOURCE.

The illustration below shows an example of the contents of an *.ear* file.



The *ShoppingCart.jar* file – the COMPONENT PACKAGE – is therefore included in a bigger *.ear* file. Only some possible parts of the *.ear* file are shown here: it includes other EJB COMPONENTS in *Catalog.jar,* which might for example be a catalog COMPONENT from a freelance developer, the web parts of the application in *web.war*, the connector to the mailing application in the file *MailingAppConnect.jar* and so on. Finally, the client part for users in your company is included in *client.jar*.

This structure will also be apparent in the Deployment Descriptor of the *.ear* file:

```
<?xml version="1.0" encoding="Cp1252"?>

<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN' 'http://java.sun.com/j2ee/dtds/
application_1_2.dtd'>

<application>
 <display-name>Shopping</display-name>
 <description>The shopping application.</description>
 <module>
  <ejb>Catalog.jar</ejb>
 </module>
 <module>
  <connector>MailingAppConnector.jar</connector>
 </module>
 <module>
  <ejb>ShoppingCart.jar</ejb>
 </module>
 <module>
  <java>Client.jar</java>
 </module>
</application>
```

This resembles the structure of the *.ear* file itself, but may include additional information such as icons or descriptions and definitions of security roles for the complete application.

Application Server

An APPLICATION SERVER for the EJB platform might contain one or more CONTAINERS. As each kind of Bean (Service, Session, Entity) has different requirements for the CONTAINER, usually one CONTAINER exists for each kind of Enterprise JavaBean. All these CONTAINERS usually run in one process.

An exception to this would be the use of a cluster of APPLICATION SERVERS, in which case the Beans might run on different computers and therefore in different processes. Even multiple instances of an APPLICATION SERVER on the same machine are possible. This is also the only case in which the distinction between different CONTAINERS is really visible. If all CONTAINERS run in the same process, the distinction is purely conceptual. For the COMPONENTS, the difference is visible because different NAMING service and different MANAGED RESOURCES are available.

All other aspects of the APPLICATION SERVER have been described in other patterns' examples and are not repeated here.

Note that J2EE (but *not* EJB) defines the term *component*, and therefore also the term *container*, differently to our definitions in this book. In J2EE, a 'component' can not only be an Enterprise JavaBean, but also a client-side application, Java Server Page or Servlet. Only Enterprise JavaBeans components fit our definition of COMPONENT. Containers in J2EE are therefore defined as the environment that a certain type of component needs to run. Thus not only Enterprise JavaBeans containers exist, but also client containers and web containers. As with components, only Enterprise JavaBeans containers fit our definition of CONTAINER.

This also has an impact on the term APPLICATION SERVER. In J2EE, an APPLICATION SERVER is also a set of containers, but those containers are the different kinds of containers defined above, as opposed to our definition of CONTAINER.

Part III A Story

This part of the book documents a real success story. It describes the actual implementation of a system based on a component infrastructure, EJB in this case. We use an imaginary dialog between two people to illustrate the concepts. One is a Java programmer from a large company's development staff – his statements are given in *italics*. You met him in the introduction to Part I, *A Server Component Patterns Language*. The other is a consultant.

❊ ❊ ❊

John meets Eric again

Hi, haven't seen you in quite a while. How are you? Still doing programming?

Yes – I'm doing component stuff now.

Oh, really? We're working on that too. Because we don't really know much about it, we hired a consultant. But I don't get what he's talking about. It's way too abstract. I mean, how am I supposed to build a system when he doesn't even show us example code? And which technology should I look at? He talks about EJB, CCM and COM+. I can only use one of them!

Ok, here's what we could do. We've implemented an e-commerce system – a shopping system – with EJB. Why don't we go through all the abstract stuff he gave you and I'll try and tell you how things are done in EJB and how we made use of it in our system?

That's a great idea – thanks! I'll just pick up the things he gave me so you can read it, too.

❊ ❊ ❊

After Eric read through the consultant's material, they meet again.

❊ ❊ ❊

Eric explains the e-commerce system…

Well, before we delve into the details of the design and implementation of our shopping system, let me give you some information about the system itself, so you know the context we're in. We developed a shopping system for a huge on-line toy store, but it wasn't a completely new system.

So it had to be integrated with other systems?

Yes. But what's more important – we developed it as a replacement for existing software. The shopping system was previously implemented as a set of monolithic CGI scripts using Perl. So by comparing the old application and the new, the benefits of using a component-based approach are nicely illustrated. The functional requirements were pretty clear, we just have to look at what the old system does.

Ok, go ahead…

From the non-functional point of view, performance, scalability and fault-tolerance were most important. In fact, that was the reason for the redevelopment of the application using EJB – the old application was too slow. A migration to a server cluster was tried, but due to the unwieldy design of the old application this didn't result in much performance improvement. It also led to problems with fault tolerance – it wasn't possible to have any fail-over, because you couldn't get another server to replace the failed one dynamically. After the shop site became more popular, this led to big problems – we had very long response times and sometimes even long downtime. That meant revenue drop due to lost customers. So something had to be done.

And that was? What kind of magic did EJB bring to your business?

…and the plan to improve it

Well, actually, we really expected that most of the problems in the area of scalability and fail-over would be solved by EJB, that was the reason for choosing EJB as the basic infrastructure. We *could* have developed a solid architecture for the Perl system, but we found that using EJB would be more efficient, because it already provided such an architecture and it's a standard solution, so there are many implementations available.

So EJB is the silver bullet here? How are these benefits achieved?

Well of course there is no such thing as a silver bullet, but actually it helped us a lot to reach our goals. But you're trying to deal with the most complex issues first… we'll come to that later. Generally speaking, there's something called a CONTAINER that handles most of these rather technical issues, and there are some established techniques for implementing COMPONENTS too – they provide the actual functionality, or business logic.

Ok, lets talk about that later. So what does the application actually do, functionally?

Eric explains the legacy system

Well, basically it had to support on-line shopping, selling products over the web, right? In the old system we identified the following building blocks:

- A catalog to choose items from. This had to store information about the items such as product id, descriptions, price, and pictures. Also, information about availability had to be present.
- A shopping cart to assemble an order, that customers could use to add and remove items and create their actual order.
- Order processing, which included status information for past orders and processing of new ones. Basically, after you have chosen what to buy, a shopping cart *becomes* an order. Then the order is processed and after some time the items are sent out – or an error situation occurs and a message is generated. The information about the orders is stored for later reference. For example, the customer can look at the status of an order he placed, using a web form.
- Information about the users is stored, which includes payment information, the user's address, and some internal ratings.

So as a customer you basically start by browsing the catalog to look for the items you eventually want to buy, then place them into the shopping cart. When you've finished filling your cart you can create an order from its contents. To do that, you have to register if you haven't already done so. The order is completed with your address and payment information. Later, you can retrieve the status of the order.

The different bits of functionality are - as I told you – rather independent parts. But unfortunately that didn't mean they had been implemented to be independent in the old software! In fact, they had been so interwoven with each other that you couldn't really identify what belonged to what in the code. So they were only distinguishable from a user's point of view, by looking at the shopping site.

Also, all these parts had to have a web front-end to make them accessible to the user, a GUI, but let's leave that aside for now – it's not something you do with EJB, there are other technologies available in J2EE for those kinds of things, like Servlets for example. They're a standard Java technology for the dynamic creation of web pages, much like CGI scripts.

Hmm – I understand. But there is nothing special so far. These parts are what you would expect from any application like this…

Yes, sure. Of course there are things happening behind the scenes. For example, the whole actual order processing is done with the help of a legacy mailing application that supports the packing of the orders and sends them to customers. It's based on a mainframe and has been used ever since even before the web interface was considered. It's quite complex, because it has to handle all the warehousing issues – it has to know where each product is stored, it even calculates the shortest path through the warehouse to collect all the items in an order.

So because it's very complex and works quite well, there's no need to change it. But it has to be integrated into the new shopping system. However, we can consider it to be a 'black box' with a well-defined interface, and this also reduces the complexity quite a bit. This was already done in the old system, of course, so at least we had a rough idea how to do it for the new system as well.

Eric outlines the different kinds of component…

Ok. Let's get a bit more specific: how did you make this into a component-based system? What did you do first?

The first step for the redevelopment of the application was to identify the COMPONENTS. Basically this is what I just told you: the application is broken down into smaller parts such as a shopping cart, a catalog and so on. At least, this the first naïve approach.

But that's rather obvious, isn't it? What does such a COMPONENT actually look like? Isn't it just a kind of module, a separate file? We've been doing these kinds of things for ages, haven't we?

Well, first of all, each COMPONENT has an explicit COMPONENT INTERFACE. And there's a separate COMPONENT IMPLEMENTATION. Maybe we should look at one of the COMPONENTS in more detail?

Yes – how about the shopping cart?

…and how to select them

Before we can look at the actual code, we should first talk about the step that comes after the definition of the functionality for a COMPONENT, which we didn't before. We have to decide which of the three kinds of COMPONENT that EJB offers should be used. It's obvious that the shopping cart needs to have state – the items you put into it must be stored somehow. But a closer look tells us that it's enough for this state to exist only for a single session within the APPLICATION SERVER. At the end of the session, the contents of the shopping cart is not needed any more, because it is either abandoned or stored in an order.

At a more abstract level, you're trying to develop a simple workflow: the first step is that the user selects items and puts them in the cart. This can occur more than once. The second step is to place the order with the selected items. This workflow has state, namely the items that should eventually be ordered. This is a rather typical situation, and EJB provides a special kind of COMPONENT for this. Do you know which one?

Well, you don't need an ENTITY COMPONENT, because that one is persistent across sessions and that isn't required. Maybe a SERVICE COMPONENT?

John begins to get the idea

But a SERVICE COMPONENT cannot have any state at all.

Oh, yes, I forgot. So it has to be a SESSION COMPONENT!

Yes. To have state just for a session and to model complex workflows, a SESSION COMPONENT should be used. EJB calls these *Stateful Session Beans*. Note that this kind of Bean has another important property: it assumes that an instance is only accessed from one client at a time, so it doesn't provide any synchronization.

Why is a Stateful Session Bean only accessible from one client?

Eric explains the limitations of Stateful Session Beans…

Usually workflows like these will be accessed from one client only, the one that initiated it and that therefore also performs it. Of course this is not always true. Imagine an order within a company – an employee wants to buy something, so he starts the workflow, then it continues to his boss for approval and eventually it reaches the purchasing department. Session Beans cannot handle workflows like these. Instead they can only be accessed by one client, and because of this limitation there's no need for synchronization of concurrent accesses. There's just one client that can access an instance, and therefore no parallel accesses can happen by definition. Omitting these features makes Stateful Session Beans more lightweight, and therefore they perform better.

Ok, I understand the synchronization and concurrency aspects. But what about the use of transactions? I read that EJB handles transactions for you automatically. To place an order, you must have a transaction on the database of some kind. How is this handled in the shopping cart? Is there a transaction for the whole lifetime of the Stateful Session Bean?

…and the approach to transaction handling

No. Usually, modern relational database systems are not tuned for transactions of this length. A transaction is active only while the order is actually created. The data for this transaction is collected during the lifetime of the shopping cart. It makes no sense to do this complete job in one transaction on the database, because this would be a very long transaction and that's usually not very efficient. For the shopping cart it would mean that each product to be ordered will be locked until the order is completed. This is because the stock total for each ordered product is decreased as the order is fulfilled, and this write access means you have to lock the data till the end of the transaction. So the product isn't available for another concurrent order. This is not acceptable, of course. It would also require lots of locks to be held, and that's usually not good for the performance of a database.

However, some database use optimistic locking. In that case the order would not be locked. The checking would take place at the end of the transaction: if the product were accessed by a different order as well, the transaction would fail at the end. Note that a simple write access is enough to break the ordering process, even if the product is still available.

So it is much less trouble to do the work in a single transaction at the very end of the order processing. Using only short transactions is in fact a benefit of the shopping cart COMPONENT.

John asks about persistence and ruggedness…

So there's no persistent state during the lifetime of the shopping cart in the database, it's only stored in the memory of the APPLICATION SERVER. *So what happens if, or rather when, the* APPLICATION SERVER *crashes?*

…which Eric explains

That depends very much on the APPLICATION SERVER you use. Usually if the server crashes the state of the Stateful Session Beans is lost. Also, timeouts might cause loss of data. However, we decided this was no big problem, as the shopping cart need not be fully protected. In other words, it's a trade-off between the less safe but more lightweight Stateful Session Bean and the safer but more heavyweight Entity Bean. If we experience a high failure rate that regularly results in lots of items getting lost from shopping carts, we might decide to use an Entity Bean instead. It really depends on the feedback from the customers. If they are bugged by this problem, we'll fix it. Never fix a problem you don't actually have…

So I pick up a hundred items in the shop, then the system crashes and I have to start all over again?

It's a trade-off. We believe this sort of failure won't occur so often that we need to use a different kind of Bean. And we gain a lot of performance! Things might be different in other applications such as stock trading, for example. Of course we had the chance of using a different kind of COMPONENT and making the shopping cart persistent. Some APPLICATION SERVERS provide clustering with automatic fail-over to a standby machine to solve this kind of problems, as well.

John asks about what happens inside a COMPONENT

I guess I roughly understand how to choose the component type. Can we now look at the code?

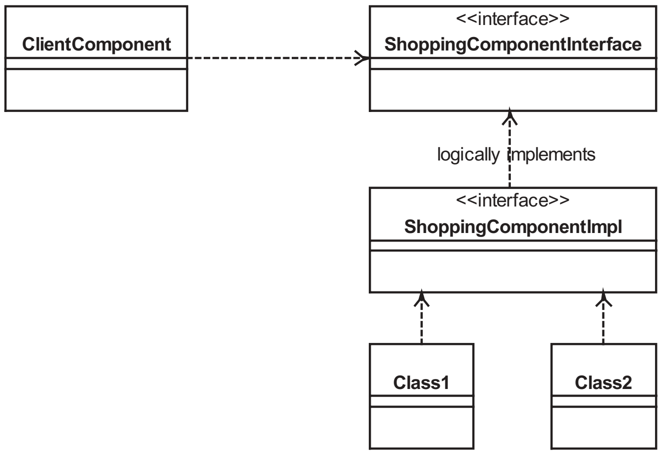
Ok. First of all, there's the COMPONENT INTERFACE. This includes all the methods that are intended to be accessed by clients.

How do you decide what to put there?

Eric explains reverse engineering the legacy system

Well, it was rather hard to define them because we couldn't take them directly from the old application. Our approach was to analyze the interactions between the parts of the old application and define *facades* for more or less complete subsystems. As I told you, we'd already identified the responsibilities for the COMPONENTS themselves, so it was just a matter of finding the right operations for their COMPONENT INTERFACES.

Here's a generic example: let's say the old application used classes such as *Class1* and *Class2* as shown in this diagram. We found the COMPONENT INTERFACE by taking those parts from the classes that are accessed by other classes. As a consequence, the *ClientComponent* uses a much smaller interface and fewer other COMPONENTS, so the dependencies are reduced. The old application wasn't object-oriented, so to find these classes we had to do a lot of analysis of the old code and additional design as well. It wasn't as easy as it could have been if we'd had a clear object-oriented design to build on.



Why would that have been helpful?

Because the COMPONENT INTERFACE, and thus the COMPONENT itself, is easier to understand if it has a simple interface and a well-defined responsibility. There are just fewer methods and therefore less to understand. This is a prerequisite for reuse: you cannot reuse a COMPONENT you don't understand.

They compare an O-O design to a COMPONENT-based design…

But that's the same with object-oriented systems. The public methods form an interface that has to be as simple as possible to make use and reuse easy. Why should I have a separate interface?

An important benefit is that we were able to exchange implementations easily. The client only sees the COMPONENT INTERFACE, so other COMPONENT IMPLEMENTATIONS with the same COMPONENT INTERFACE can be used without the need to change the client.

Did you actually use that benefit?

…and Eric outlines a major advantage of COMPONENT-based design

Well, the first implementation of the shopping cart had a major performance problem. Every time something was put into a shopping cart or removed from it, the action was logged in a special database table. We did this so that we could learn more about the behavior of the customers with a view to optimizing the system. This back-end was not sufficient for the amount of data that was produced and didn't offer enough statistical evaluation methods either. So we exchanged the implementation completely to use a different back-end. This was greatly simplified because of the COMPONENT INTERFACE – we only had to exchange the implementation and then we were done. The same is true for the evolution of COMPONENTS that takes place day by day, for example through fixing bugs or by modifying minor parts of the business logic. This made the software much more maintainable.

Didn't you have any problems at all?

…and some snags

Well, there are some issues you have to take into account. For example, you are out of luck if the client of a COMPONENT makes implicit assumptions about properties of the COMPONENT that are not part of the COMPONENT INTERFACE. Response times and performance in general are such issues. These are non-functional aspects that are not covered by the COMPONENT INTERFACE, so they might change, as they did when a new version of the shopping cart was installed.

Sorry, but I don't get it…

If, for example, a client relied on the fact that the shopping cart took a certain amount of time to add an item, it would have had problems with the newer, faster variant.

Other aspects might change too. The COMPONENT INTERFACE does not define what is actually done - the semantics of an operation. For example, there must be a method to add something to a shopping cart. However, this method could be changed so that it doesn't add the item any more, but instead orders it immediately. A client might have problems in that case. The only hint about the semantics of a method is usually its name or some kind of documentation – if you have it.

So the exchange of a COMPONENT is *technically* simplified compared to normal objects, but it is still not without risk. But you can't blame the infrastructure for that.

Ok. Back to the shopping cart. What does the interface actually look like? Show me code.

Eric demonstrates the code for the cart…

It's quite simple. Look:

```
public interface ShoppingCart extends javax.ejb.EJBObject {
  public void add(ProductDescription pd, int number) throws
   java.rmi.RemoteException;
  public void remove(ProductDescription pd) throws
   java.rmi.RemoteException;
  public float totalValue() throws java.rmi.RemoteException;
  public void order() throws java.rmi.RemoteException;
}
```

Basically, there are methods to add and remove products. These take a description of a product that can be retrieved from the catalog. The class *ProductDescription* only contains information about the price, name and manufacturer of a product, plus a unique identifier.

The total value of all products in the shopping cart can be calculated using the *totalValue()* operation, and finally the order can be placed with the *order()* operation. This is usually the last method that's called within the workflow. After that the shopping cart is discarded.

So that's the interface of the shopping cart. But how do I create a shopping *cart instance? There can't be a constructor, some kind of a factory must be used, I guess. You know, like the one described in the GoF book [GHJV94].*

…and explains how it gets instantiated

Yes, that's true. In fact, EJB uses this pattern, too. This means there must be a method somewhere that returns instances that somehow implement the required COMPONENT INTERFACE. This method is provided by the COMPONENT HOME. For our shopping application, the COMPONENT HOME appears to be a new burden EJB brings to us. However, in the old application we also made use of the *Factory* pattern [GHJV94] you already mentioned. These factories did almost the same as the COMPONENT HOME: for persistent objects they either created new entries in the database or searched in the database for a certain entity.

Of course, some of the transient objects also had Factories, but most often we had to introduce them during the transition to EJB. So basically the COMPONENT HOMES mean that we have to implement these Factories for every type of COMPONENT, and we have to do that in the way specified by EJB. It's an investment that gains us the benefits of COMPONENTS, because the performance optimizations especially aren't possible without a COMPONENT HOME. Only in this way can the CONTAINER hook between the COMPONENT INTERFACE and COMPONENT IMPLEMENTATION to do its tricks there.

What does a COMPONENT HOME look like?

Oh, for our shopping cart, this interface, the so-called Home Interface, is very simple:

```
import java.rmi.*;
import javax.ejb.*;

public interface ShoppingCartHome extends EJBHome {
 public ShoppingCart create() throws RemoteException,
  CreateException;
}
```

Just one method is defined to create new shopping carts and it looks almost like a simple Factory.

So after you define all these methods, where do you put the actual code that should be executed?

In the COMPONENT IMPLEMENTATION.

What does it consist of?

Eric introduces lifecycle management…

Well, of course the code implements the business methods defined in the Remote Interface. This part of the code has the same functionality as the code of the old shopping application. Additionally, we had to implement the methods defined in the *SessionBean* interface. These define the LIFECYCLE CALLBACK operations, for example.

LIFECYCLE CALLBACK? What's that?

LIFECYCLE CALLBACKS are necessary to manage an instance's lifecycle. That includes not only creation and deletion, but also persistence. This might be considered overhead, but in fact it has several advantages. In big applications, these issues have to be dealt with anyway. Using COMPONENTS, there is a well-defined approach that also allows some optimizations. It's important to note that the style of programming has changed: in the old application, the application itself decided when to store data. Now this is decided by the CONTAINER and the COMPONENTS just have callbacks in place for it.

For a Stateful Session Bean like the shopping cart there won't be much code of this sort. Most of the LIFECYCLE CALLBACKS are left empty because the shopping cart is so simple that not much lifecycle-dependent stuff needs to be implemented.

So what does the actual code look like?

…and outlines the implementation

Here's a rough sketch of what the implementation of the shopping cart looks like:

```
import java.rmi.*;
import javax.ejb.*;

public class ShoppingCartEJB implements SessionBean {
 private SessionContext sessionContext;
 // The state of the shopping cart is left out.
```

```
  // callbacks for the Container:
  public void ejbCreate() {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }

  public void setSessionContext(SessionContext sessionContext) {
   this.sessionContext = sessionContext;
  }

  // business logic
  public void add(ProductDescription pd, int number) {
    //...
  }
  public void remove(ProductDescription pd) {
    //...
  }
  public float totalValue() {
    //...
  }
  public void order() {
    //...
  }
}
```

The LIFECYCLE CALLBACKS for the CONTAINER we were talking about are the methods that start with *ejb* and the *setSessionContext* method. The CONTAINER calls the methods *ejbCreate()* and *ejbRemove()* when a new instance is created or an instance is deleted. They've been left empty, because there is nothing for them to do in this case.

Eric describes COMPONENT PASSIVATION

The *ejbActivate()* and *ejbPassivate()* operations are used for resource optimization. If there are too many shopping cart instances in the memory of the APPLICATION SERVER, some of them are temporarily stored to hard disk. They must be serializable for this to be possible. This means they may not have any references to non-serializable objects when the instance is actually passivated.

What does that mean? Can you give me an example?

Well, database connections, for example, are usually not serializable. So before the instance is written to disk, or *passivated* as it is called in EJB-speak, the state of the instance must be changed accordingly. So the connection is closed and the reference to it set to *null ejbPassivate().* You could also declare the variable as transient. The connection is re-acquired in *ejbActivate()*. *ejbPassivate()* is called by the CONTAINER just before PASSIVATION and *ejbActivate()* after the instance is activated again.

Usually *ejbActivate()* and *ejbPassivate()* are empty, but they allow for reaction to certain events in the lifecycle, for example by acquiring or releasing resources.

What is this setSessionContext() method used for?

… explains how a COMPONENT can examine its environment…

It's called by the CONTAINER, and an object is passed to the Bean that it uses to access certain functionalities of the CONTAINER. It's an implementation of the COMPONENT CONTEXT. We'll see more of this later on.

So this is all stuff that's related to the fact that the shopping cart is a COMPONENT. The implementation of the actual business logic looks very much like normal Java code…

…and some implementation restrictions

Yes, basically it's normal Java code. But COMPONENT-based development also has an impact on that code: during the implementation you mustn't use certain features provided by the Java language or the libraries, because the CONTAINER imposes some IMPLEMENTATION RESTRICTIONS. Basically these restrictions are in place to enable optimization by the APPLICATION SERVER that hosts the Beans. That can only be accomplished if the COMPONENTS don't make full use of the features that the programming environment usually offers, so as not to interfere with the assumptions the APPLICATION SERVER makes as a basis for its optimizations.

A specific example is multi-threading. The old shopping system used sophisticated threading techniques. You're not allowed to use these inside your Beans. However, they're also unnecessary, because the APPLICATION SERVER makes sure that concurrent requests are handled efficiently. It's in charge of multithreading because it's a technical concern. And because your COMPONENTS mustn't interfere with this, threading is simply disallowed for COMPONENTS.

Another example is file access. The old shopping application used files in some cases to store data. In EJB access to files is not allowed for COMPONENTS. The main reason is that file handling is quite difficult if you're using a cluster of APPLICATION SERVERS – you have to synchronize read and write operations. So this is a real limitation. However, to solve the problem, file access is now replaced by a database. As a side effect, this results in a more reliable handling of the data, even in a clustered environment.

…which John doesn't much like…

But I want to control threads, I want to access files. I've always done this!

You don't need to, because the APPLICATION SERVER takes care of most of these issues. Isn't that a relief for you? Be honest – many of the problems you had in the applications in your company were due to these technical concerns: deadlocks in threading, inefficient use of resources, memory problems and so on.

I have to admit you're probably right. But I still need to access files…

…so Eric explains safe workarounds…

In that case you have to compromise. For example, you can build an external server that takes care of these things and is accessed by the EJBs. Some APPLICATION SERVERS also provide files as MANAGED RESOURCES, and you can access them under the control of the CONTAINER. There are practically-usable ways in which you can work around these restrictions.

But still I have to give up control over some issues?

…and highlights the advantages of COMPONENTS

That's true. But you must think about COMPONENTS in a different way than you think about 'normal' software. COMPONENTS live inside an APPLICATION SERVER. Therefore you give up a lot of control. But on the other hand, the APPLICATION SERVER handles a lot of nontrivial aspects for the COMPONENTS. Most programmers struggle with these things!

So far so good. Let's take a look at the next COMPONENT. Maybe the user?

They then move on to discuss the user COMPONENT

Fine. Let's again start with a discussion of the type of COMPONENT that can be used to represent users. There are two primary characteristics. First, user data has to be persistent. It has to be available for several sessions, over long periods of time. Second, it is possible that a specific user is accessed by several clients at the same time: for example, a new shopping session could be created for them while another COMPONENT updates some statistics. These concurrent accesses to a specific user have to be synchronized. As such, a user is a typical example of a business entity: persistent and concurrently accessible. It is therefore implemented as an ENTITY COMPONENT, or Entity Bean, as they are called in EJB.

So, let me see. There must be a Home and Remote Interface again and an implementation. Are there any differences?

Well, the Home Interface has some additional methods to find existing users in the persistent store. And the LIFECYCLE CALLBACKS in the implementation are different. But that's about it.

But you said the users are persistent. The shopping cart was not. So there must be more differences. How are the users stored in the database?

…and how it differs from the shopping cart COMPONENT

Basically there are two ways in EJB to do this. One is using Bean-Managed Persistence, or BMP for short. In this case, specific LIFECYCLE CALLBACKS must be implemented that load and store the state of a user. In addition, you have to provide operations that query the database to find already-stored users.

So I write the usual database code, just at a different place in the code?

Yes, if you use BMP. However, note that you don't get to call that code. It's the responsibility of the APPLICATION SERVER to decide when to call this 'store' operation. But there is another possibility that is a bit less work.

And that is?

…by being persistent

Entity Beans offer an out-of-the-box solution for persistence called *Container-Managed Persistence,* or CMP for short. With this, loading and storing the data is handled by the CONTAINER. You just have to specify a mapping from attributes to table columns, or a *where* clause the CONTAINER should use for querying. How this works exactly depends on the APPLICATION SERVER you use – that is not standardized. Actually, we used CMP to implement our user Entity Bean. This means that the code we've written is clean of any database handling.

How is this done?

Eric explains how the persistence code is generated

Generally speaking, for recurring development tasks, a typical simplification is to use code generators. That's how the code for the persistence is created here. An APPLICATION SERVER usually contains a specific section of code that can create the persistence-handling code from abstract specifications. Such a tool is usually called an IMPLEMENTATION PLUG-IN. The generated persistence code is part of the GLUE-CODE LAYER and used by the APPLICATION SERVER.

So APPLICATION SERVERS use code generators packaged as an IMPLEMENTATION PLUG-IN, I understand. Did you add your own code generators for specific, repetitive aspects?

In EJB it's hard to plug in arbitrary code generators. We had no chance to integrate code generators directly into our EJB project to make it a real IMPLEMENTATION PLUG-IN. However, in some cases it makes sense to generate code in the project, outside of the APPLICATION SERVER. For example, the Remote and Home Interface can most often be directly generated from the Bean Implementation. We did that quite frequently in our project and so we provide a generator for that. However, it isn't integrated in the APPLICATION SERVER, it's used manually each time the implementation is changed in a way that requires a new Remote or Home Interface. We couldn't integrate this into the APPLICATION SERVER, but we made it an integral part of our build process.

Ok, I guess I don't want to see the code of the user COMPONENT. Let's continue to the order…

Next they discuss the order COMPONENT…

That's an interesting one, because at first sight it appeared to be one single COMPONENT, but in fact we later realized that it was several different COMPONENTS. First of all, there's a COMPONENT that manages the information about the order.

Hmm, it has persistent state. Another Entity Bean, I guess?

No, in fact it is a Stateless Session Bean.

But Stateless Session Beans don't have any state at all!

…and Eric explains why it does not need state

This one doesn't need to. It directly accesses the database and returns the results immediately.

What does that look like?

Only a few methods are required to get detailed information about an order. The Bean can also include business logic working on the data. For example, the cancellation of an order is implemented in the same Stateless Session Bean that gives access to the order data:

```
public interface Order extends EJBObject {
 public OrderInformation getInformation(int orderNumber)
   throws RemoteException;
 public void setInformation(int orderNumber, OrderInformation
   orderInformation) throws RemoteException;
 public void cancel(int orderNumber) throws RemoteException;
 // ....
}
```

The *OrderInformation* class includes all the information necessary about an order directly from the database. It can be retrieved or changed by using a unique identifier, the order number. With this identifier it's also possible to call business logic, for example the *cancel()* method shown. Of course this does not look very object-oriented. The method *cancel()* is not called on an object representing an individual order, but rather on the complete set of all orders – the specific order is identified by the provided parameter.

Let me get this. This COMPONENT *has no state of its own, it just accesses the database and returns the data from it without any further manipulation. A sort of data conduit?*

Well, more or less. Let's call it a nice, componentized facade for the database instead of a conduit. But besides piping data, it also offers some simple business logic.

Strange. Is that the only COMPONENT *that's implemented this way?*

The interface to the legacy mailing system was done the same way

No, the access to the old mailing system is similar. We implemented the interface on the EJB side as a Stateless Session Bean. This Bean just needs to forward data to the legacy system and receive data from it. The Bean is just an adapter and needs no state of its own. Therefore it makes sense to implement it as a Stateless Session Bean. The benefit here is that the complex code for accessing the legacy system is hidden from the client, and the client just gets a simple API

…which gives benefits of complexity-hiding…

with the required functionality. If the legacy system is only accessed through this facade, it becomes much easier to eventually replace the old system with a newer one, as it wouldn't affect the rest of the application.

How is the mailing COMPONENT *implemented? How do you actually access the legacy system?*

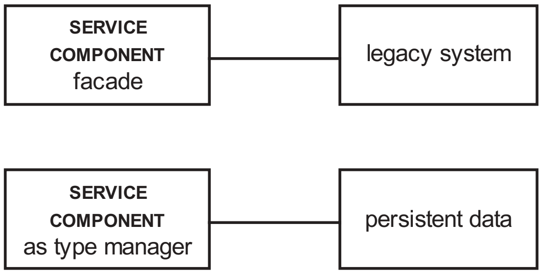
…and J2EE provides APIs to implement it…

This is done using the J2EE Connector Architecture, which provides a kind of PLUGGABLE RESOURCE. In other words, you can define something that behaves roughly like a database driver inside your APPLICATION SERVER and provides access to a legacy system. This results in the same benefits as for database connections or other MANAGED RESOURCES: transactions, security and pooling are handled automatically.

Ah, so it provides some kind of homogenous access to connection-oriented back-end systems?

…which make life a lot easier for the developers

Sort of. The APIs should be very similar, and there's a Common Client Interface that every resource adapter should provide. So ideally you just have to learn a single API to communicate with different legacy systems. This is also a different form of reuse. So the connector to the mailing system itself can be reused if other COMPONENTS need to access the same system.



To sum up, access to the legacy mailing system is provided via a *Service Component Facade*. This hides the complexity of access to the legacy system and therefore is considered a *Facade*. The order status uses direct access to the database. This is called a Type Manager because it manages all data of the same type. Both are implemented as Stateless Session Beans and have similar interfaces. You can get more information on these uses of Stateless Session Beans from [VSW02].

John remains sceptical…

I don't see the point. Stateless Session Beans are very much like normal remote objects and you don't use Entity Beans except for the user data. So what is the point in EJB? I could use plain CORBA or RMI!

First of all, there are other advantages that we'll discuss later on. These are primarily related to the existence of the CONTAINER, and they give us lots of benefits in the areas of scalability and fault tolerance. And you forgot that there are also Stateful Session Beans…

But what is the reason for not using Entity Beans?

Performance and the uniform access to legacy and database data.

Performance? How's that?

…so Eric goes into some detail

As mentioned in the ENTITY COMPONENT pattern, Entity Beans synchronize for concurrent access, and that imposes some performance overhead. We did some tests for our shopping application, and even though the APPLICATION SERVER has a cache for the state of Entity Beans, so that it need not be read from the database for each access, Stateless Session Beans were still much faster. As some of the legacy systems were also accessed using Stateless Session Beans, we decided to use a similar approach for the data that's stored in the databases as well. That allowed us to use a common programming model to access both types of storage: relational databases for the new entities and legacy systems for the old ones. I admit that modeling business entities as Entity Beans would be usually more natural, because one COMPONENT instance represents one business entity instance in that case. The performance depends very much on the APPLICATION SERVER and the way in which the entities are accessed, though: how often, read/write, concurrently or not and so on.

After we implemented and installed the shopping system, we did some testing with a newer version of the APPLICATION SERVER and also with other APPLICATION SERVERS. We tried a change from Stateless Session Beans to Entity Beans and noted better performance. That's because of better caching and other improvements in EJB 2.0.

So now you would implement the system using Entity Beans?

Yes. Especially with the new features of EJB 2.0. With Local Interfaces and the new persistence features, it's much more powerful.

And legacy access?

You can have Entity Beans that store their data in a legacy system implementing your own persistence. Remember BMP. Using a suitable connector to your back-end as a PLUGGABLE RESOURCE makes this a rather straightforward thing to do.

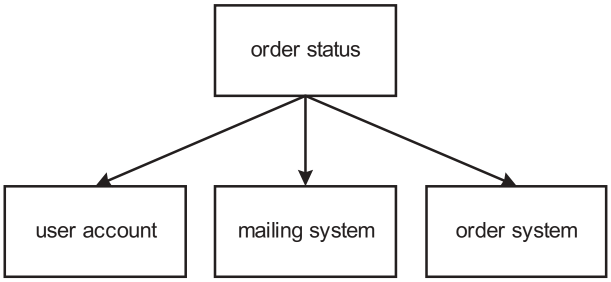
Why didn't you implement the user as a Stateless Session Bean as well?

We decided to pay the price for synchronization in this case, because multiple different processes might concurrently access the user information and because usually only very few user's data is accessed simultaneously. The caching is also very efficient in this case. There aren't that many users logged in concurrently, so the data of all the logged-in users is available from the cache of the APPLICATION SERVER. The caching was also much more efficient than a prototype we built based on a Stateless Session Bean.

… and John is reluctantly convinced.

Well I think this whole Stateless Session Bean instead of Entity Bean stuff is not very obvious, but reasonable. When we started discussing the order, you said that one COMPONENT *was not enough for the implementation of the order. What else did you do?*

They move on to the rest of the order COMPONENT



As I mentioned at the very beginning, the order COMPONENT should include some means of giving information about the current state of an order. The COMPONENT we discussed above only gives basic information, such as which items were ordered. However, the order status should return an object that contains information on which user issued the order, whether and when it was shipped and what the order contains. This information must be assembled from several sources: the user account…

…that is the Entity Bean we discussed earlier on, right?

Yes. Additionally the mailing system must be used to retrieve the status of the delivery.

That's the other Stateless Session Bean we discussed, I guess.

Yep. And also the other Stateless Session Bean – the order system – must add its information. Access to these three COMPONENTS is hidden in the *OrderStatus.*

Why did you introduce another COMPONENT just for the status of an order?

The benefit of this is that we hide the complexity of assembling all this information from the client. If this COMPONENT did not exist, the client would have to invoke all these operations on the three COMPONENTS itself. This would make the client much more complex and make reuse of this code harder, because it would be hidden somewhere in the client code.

Eric highlights the impact of COMPONENT-based development on reuse

Reuse of the code might be useful, for example, if a call center needed to access the order status as well. For a call center, the client would possibly be a Java application and not a web application like the one for the customers. It's much easier to reuse a COMPONENT than a part of the client code that must be manually cut out.

Also, putting the code on the client would result in more network traffic. With the COMPONENT, only one remote call is needed for the order status – otherwise calls to several COMPONENTS would be necessary.

Is there no way to avoid this additional COMPONENT?

Well, in EJB 2.0 you can have additional operations in the Home Interface. So the functionality could be implemented there instead of in additional COMPONENTS. However, this would make the order depend on the user and mailing system. COMPONENTS are introduced to built software from independent parts.

Makes sense. But what type of COMPONENT will this be? It's a workflow like the shopping cart, so I believe it should be a Stateful Session Bean.

… and discusses the order status COMPONENT.

Close. In fact it's a Stateless Session Bean.

Ok, I see. It just has to be given the specification of an order and then the information for that order must be collected from the other COMPONENTS. *This can be done in one method call and therefore no state is needed!*

Exactly. This Bean just takes an order number and returns the respective data to the client. You can see this functionality also by looking at the Remote Interface of the order status:

```
import java.rmi.RemoteException;
import javax.ejb.*;

public interface OrderStatus extends EJBObject {
 OrderStatusInformation getOrderStatus(int orderNumber)
  throws RemoteException;
 OrderStatusInformation[] getOrderStatusForUser(int
  userNumber) throws RemoteException;
}
```

So, for a given user or order number, the *OrderStatusInformation* is assembled and returned. The *OrderStatusInformation* is a serializable Java class with the following definition[1]:

```
public class OrderStatusInformation implements java.io.Serializable
{
 private boolean shipped;
 private boolean singnedFor;

 private boolean processed;

 private AddressInformation shipToAddress;
 private NameInformation shipTo;

 // public setters and getters go here
 // ...
}
```

The first two attributes show whether the order has been shipped and whether the recipient has signed for it. This information is provided by the mailing system, for example. The next attribute *processed* is *true* if the order was processed, which is also information provided by the mailing subsystem. Information about to who and to where the order was shipped is provided by the user account sys-

1. This is called a Data Transfer Object in [VSW02].

tem, because only the user's id number is stored in the order.

John thinks this is the end of the story…

So the order is in fact two COMPONENTS?

Well, you haven't seen the full picture yet. There is one more COMPONENT that deals with orders.

What is its purpose?

…but it isn't

The same that applies to the order status is also true for the processing of an order. Several things have to be done. The address must be collected from the user, the mailing system must be activated, some error states such as out-of-stock products or invalid addresses must be handled and so on.

But there is the order() method in the shopping cart…

…but that just delegated the real work to the *OrderProcessing* COMPONENT. And that's where the interaction with the mailing system, the order system and the user information takes place.

Almost like the OrderStatus we talked about…

Yes, but there is a difference. This time the information is not collected, but rather changed in the COMPONENTS. As we just saw, there are two good reasons to do information gathering in a COMPONENT on its own. One is that the complexity is hidden, the other is better performance due to less network traffic. Reuse is also easier. When you consider updating data, another reason is added: transaction demarcation. Transactions can be started and committed in the Session Beans even if the Bean accesses several Entity Beans or other Stateless Session Beans. If this wasn't hidden inside a COMPONENT, the client would need to start and commit transactions, something you certainly don't want clients to be responsible for.

How is this important for our current problem?

Eric explains database rollback issues

There are a lot of different database tables that must be modified. The number of items in store must be changed, and a new entry in the order table must be added. There are also a number of reasons why this might fail. For example, the customer's credit card authorization might fail, some items might not be in stock and so on. In such cases all changes must be rolled back. So you have to use a transaction that can be rolled back in error situations like these.

John sums up the design decisions for the order COMPONENT

Ok, let me rephrase it in my own words: we have the order, which in fact consists of three COMPONENTS *for different aspects of the order. Both to process an order and to retrieve the status of an order, an additional* COMPONENT *is used. Because those are very simple workflows, a Stateless Session Bean is used. To access the persistent data of an order, direct access to the database is made, again in a Stateless Session Bean. The reason for that is performance. And also the access to the mailing system was done in this way, but with a J2EE Connector to a legacy system instead of the database.*

Yes, but these would now be done with Entity Beans in EJB 2.0. From the design point of view it's a compromise, because of the limitations of EJB1.1.

Ok. Let me continue: the shopping cart is a Stateful Session Bean because the workflow it models is more complex and therefore needs state. The user component is also just for access persistent data, but in this case an Entity Bean is used because concurrent access occurs and that can be more efficiently handled by an Entity Bean.

They move on to discussing the CONTAINER…

So I have at least a basic understanding about COMPONENTS. *But our consultant told us about the* CONTAINER *as well…*

Yes, a CONTAINER is the environment in which the COMPONENTS run. In the case of EJBs, a CONTAINER is usually a part of a J2EE APPLICATION SERVER you buy from some vendor.

Why should I use one?

That's simple. You have to!

But why? Can't I just run my COMPONENTS *as standalone applications?*

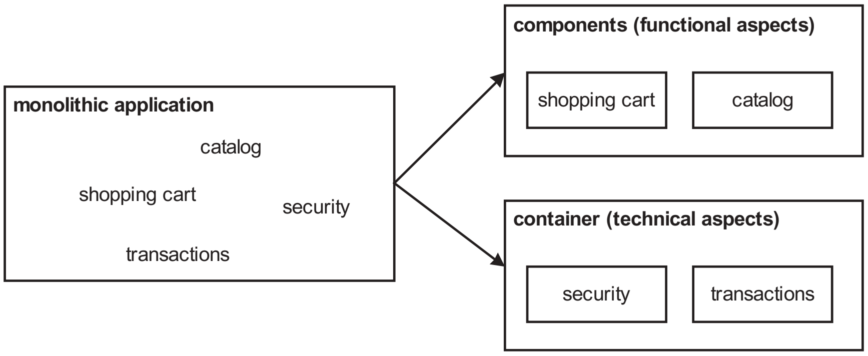
…and Eric explains why they exist…

No, that isn't possible, as Enterprise JavaBeans require a CONTAINER and you wouldn't even want to live without one. Let me start by telling you what happens if you have no CONTAINER, as was the case in the old shopping application. This implemented some functionality for the business logic, and also took care of security features, as well as persistence, transactions across multiple databases and database connection pooling. These technical concerns were mingled with the functional parts, inside the same source entities.

But that's just how things are. How is component-based development different here? And what has this to do with the CONTAINER*?*

…and the advantages they confer…

In a COMPONENT-based system, technical and functional concerns are cleanly separated. This has several benefits, but primarily it promotes reuse of the technical concerns. In fact, the goal of the EJB component model is to define a standard for technical concerns, allowing these to be reused efficiently. They can even be bought as an off-the-shelf package in the form of an APPLICATION SERVER. Developers don't need to worry about implementing these things themselves. They can focus on their business logic.



That sounds like a great advantage…

Yes, it made the development of the new shopping system much easier. For example, authentication and authorization are now completely done by the CONTAINER.

And I guess the infrastructure is much better than the one you developed yourself…

…such as separation of fail-over from COMPONENT *implementation*

You bet! One benefit in fact was the clustering features of the APPLICATION SERVER, which take care of the thread management on a multi-processor machine. This makes the software much more scalable without any further modifications in the COMPONENTS. Also, it deals with fail-over: an APPLICATION SERVER might be distributed across multiple machines, with one taking over if another one fails. This makes the application more fail-safe, and this is an important point, especially for an Internet application with 24/7 requirements. Of course you have to do some configuration to enable this feature, but your COMPONENTS remain untouched.

Are you bound to a specific APPLICATION SERVER?

No. You can choose between different APPLICATION SERVERS if you use EJB. Therefore you can get different levels of Quality of Service through buying the APPLICATION SERVER that best fits your requirements and your budget.

But shouldn't I have more control over these technical concerns? I don't feel good giving away all this control I was used to have in the past…

Remember back in the days when databases weren't mainstream, as they are today, and everybody had to do their own persistence? Then databases became mainstream, we gave up control over persistence, and systems were built using off-the-shelf databases. Well, a similar thing is currently happening here.

So I don't need to worry about that any more?

It is still a challenge to build a scalable system, but now the challenge is at a different level. It's now more of a design issue. For example, you should use Session Beans for workflows. Then the actual processing is done on the server and you have less client-server communication and hence improved performance.

But how can all this magic happen? I mean the APPLICATION SERVER cannot know about all the technical things I need.

Eric introduces ANNOTATIONS

It can't, of course. You have to give some hints. ANNOTATIONS are used to tell the CONTAINER how it should handle the technical concerns. In other words you still have to *specify* these things, but you don't have to code them. These ANNOTATIONS are called Deployment Descriptors in EJB.

So the Deployment Descriptor is much like a configuration file…

Yes. It's a configuration file that tells the CONTAINER how to handle the technical aspects for a specific COMPONENT.

…and demonstrates an example Deployment Descriptor

Can we look at a typical Deployment Descriptor then?

Yes, of course. Deployment Descriptors use XML – like everybody else these days.

Here’s the Deployment Descriptor for the shopping cart:

```
<?xml version="1.0" encoding="Cp1252"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-
jar_1_1.dtd'>

<ejb-jar>
 <enterprise-beans>
  <session>
   <ejb-name>ShoppingCart</ejb-name>
   <home>shoppingcart.ShoppingCartHome</home>
   <remote>shoppingcart.ShoppingCart</remote>
   <ejb-class>shoppingcart.ShoppingCartEJB</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Container</transaction-type>
  </session>
 </enterprise-beans>
 <assembly-descriptor>
  <security-role> <role-name>user</role-name>
  </security-role>
  <security-role> <role-name>administrator</role-name>
  </security-role>
  <security-role> <role-name>guest</role-name>
  </security-role>
  <method-permission>
    <role-name>user</role-name>
    <role-name>administrator</role-name>
    <method>
      <ejb-name>ShoppingCart</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
  <method-permission>
    <role-name>guest</role-name>
    <method>
      <ejb-name>ShoppingCart</ejb-name>
      <method-name>add</method-name>
    </method>
    <method>
      <ejb-name>ShoppingCart</ejb-name>
      <method-name>remove</method-name>
    </method>
    <method>
      <ejb-name>ShoppingCart</ejb-name>
      <method-name>totalValue</method-name>
    </method>
  </method-permission>
```

```
    <container-transaction>
      <method>
        <ejb-name>ShoppingCart</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
   </assembly-descriptor> </assembly-descriptor>
  </ejb-jar>
```

After the actual definitions of the parts of the shopping cart, the next part defines the security policy. The roles *user*, *administrator* and *guest* are defined. Then the access rights are specified for these roles: every *user* and every *administrator* may call any method on a shopping cart. A *guest*, however, is only allowed to create new shopping carts, fill them and calculate the total value of the contents. If a *guest* registers with the shopping system, he will be given the role of a *user*. Then he can actually order the items he's put into the shopping cart.

Hmm, surely a user can't access any shopping cart other than his own? No further details are given…

No, it 's not possible to limit a user to his own cart through the Deployment Descriptor. This has to be done manually in the code.

So no magic here. Can we look at this code later? Tell me what the other parts of the Deployment Descriptor are about.

The rest deals with transactions. It just specifies that every method must be called inside a transaction: all methods have the transaction attribute *Required*.

Why is that?

This is useful to ensure that access to databases is done inside transactions. Accessing a database with undefined transaction status might lead to loss of changes in the case of concurrent access, and other typical synchronization problems.

Eric describes the security code in the COMPONENT CONTEXT

You mentioned that code must be written to limit access to a shopping cart to a single user, the user who created the cart. Can you show me how?

Yes, sure. To implement it, we first have to think how the environment set up by the CONTAINER can be accessed by the COMPONENT.

This is done by using the COMPONENT CONTEXT. It gives access to information such as the caller for the current operation, or the current transactional status. In the old shopping application, there was a data structure that returned the current user for each session. The user was identified by a cookie in his web browser. Here's how we used this COMPONENT CONTEXT to implement the security check we talked about:

```
public void checkUser() {
 if ( sessionContext.isCallerInRole("user") &&
      sessionContext.getCallerPrincipal()
      .getName().equals (userAccount)) {
    throw new SecurityException(
       "User may only access his own shopping cart!");
 }
}
```

Wait a second! There's a bug: your if *statement is wrong – you didn't check for the other roles.*

Let me see. No, it's correct.

But if it's coded that way, a user in the role of guest can do anything, surely?

No, because he's already blocked by the constraints defined in the Deployment Descriptors and enforced by the CONTAINER.

But that's not defensive programming, that's for sure. I mean if the guy who deploys the application defines a new role that can call the method, you may get a security hole.

Yes indeed. But look at the alternative: if we check all the groups within the code, that problem can't arise any more. But on the other hand, you also lose the ability to add a new role without changing the code of the COMPONENT. So you have to choose between being able to administer your COMPONENT more flexibly, or being more defensive against administration failures. We chose the first route, but of course you also can take the other. The important thing is to remember that both the programmatic and the configured permission management must match. The rule is to let the CONTAINER do as much as possible. That was why we used a CONTAINER in the first place, after all.

John asks about the SessionContext

Ok, I see. And where does this SessionContext come from?

The *SessionContext* is provided to the COMPONENT instance using a specific LIFECYCLE CALLBACK operation, namely *setSessionContext()*.

Right, I've seen that. It just takes the Session Context and stores it in an instance variable, right?

Yes, that's the idea.

But I'm confused. Who calls setSessionContext()?

We already discussed that. LIFECYCLE CALLBACK operations are called by the CONTAINER at the appropriate times during the lifecycle. Remember that COMPONENTS are passive – they're only called by the CONTAINER if it judges that it's the right time.

I guess the COMPONENT CONTEXT also gives access to transactions?

Yes.

Did you use that?

Yes. We talked about the *OrderProcessing* COMPONENT before. That does the actual processing of the order, remember? A lot of data in different locations must be updated. These updates mustn't occur if, for example, the customer's credit card payment fails. So the whole process is done in a transaction and rolled back if an error occurs. To mark the transaction to be rolled back, code like this is used:

```
// ...
if (...) { // Credit card payment failed
 sessionContext.setRollbackOnly();
}
```

That's all…

Where is the transaction started?

The transaction attribute of this method is set to *Required* in the Deployment Descriptor. So a transaction is either started at the beginning of the method, or the caller started it. The code above just specified that the transaction should be marked for rollback, which means that whatever happens subsequently, the transaction is rolled back.

Is there anything else about the Deployment Descriptor I ought to know?

Eric describes how the Deployment Descriptor aids reuse…

Sure. The Deployment Descriptor is very important for reuse as well. After all, a major benefit of component-based development is reuse. This is also the reason why COMPONENTS are called 'components': the idea is to build software from pre-manufactured parts – the COMPONENTS – by assembling them. Deployment Descriptors are very important for that!

Why is that?

They keep the technical concerns out of your code. You can even argue that they are a prerequisite for reuse and a COMPONENT marketplace.

Uh, why?

Because COMPONENTS must be usable in different contexts, different APPLICATION SERVERS and different system environments. This can only be accomplished by customization. And this can only be done outside the code, because usually you don't get the source code for the COMPONENTS you buy. So there has to be another way – a Deployment Descriptor.

So if you have a Deployment Descriptor, everything is ok?

In fact the main precondition for reuse is to have a defined base, like EJB for example, that each COMPONENT can rely on. The Deployment Descriptor makes customization, and therefore reuse, much easier. So it's an important part of the big picture.

Can you give an example?

Yes, we forgot one COMPONENT, the catalog. We didn't develop this ourselves, we bought it from another vendor instead.

So what problems did you have reusing this COMPONENT?

The catalog COMPONENT we bought had to be configured. The vendor we bought it from is located in the US, so all prices were in US dollars, using the currency symbol *USD*. As we're located in Germany, this had to be changed to Deutschmarks (*DM*) and later Euro (*EUR*).

How did you do that?

Just by putting some values in the Deployment Descriptor. It's not just parameters to control the technical and deployment aspects that can be placed there, but also parameters for the functional parts of the COMPONENT. To set the currency symbol, for example, a CONFIGURATION PARAMETER was added to the Deployment Descriptor. All we had to do for this is to change the Deployment Descriptor accordingly, to change *USD* to *DM* and later *EUR*. This is the part where the changes were made:

```
<session>
 ...
 <env-entry>
  <env-entry-name>currencySymbol</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>EUR</env-entry-value>
 </env-entry>
...
</session>
```

Ok, so Deployment Descriptors are essentially a prerequisite for reusing COMPONENTS in different environments. They appear to enable small modifications and adaptations without touching the source code. Just like configuration files…

Yes, but there's more to Deployment Descriptors than that. Server-side COMPONENTS, especially, usually have more complex configuration requirements: they use other COMPONENTS and access MANAGED RESOURCES such as database connections. To get a COMPONENT to run in a different environment means that these things must be configured as well.

Hmm. How are those resources accessed anyway?

There's a part of the APPLICATION SERVER called NAMING. It returns a reference to a MANAGED RESOURCE or to a COMPONENT if you pass it the searched item's name. All parts of a component-based system are registered there under specific names: COMPONENTS, of course, and databases connections, and other MANAGED RESOURCES.

What does the Deployment Descriptor help here?

The developer of the catalog COMPONENT for example had his own database that was configured differently to ours and had a different name in NAMING. We had to set up the catalog COMPONENT to use our database. This can't be accomplished with NAMING alone, because the names for the database are different in the two systems. So the database would have to be reconfigured for each reused COMPONENT that had a different hard-coded database name.

So somehow the database must be configured to have different names for each COMPONENT?

…and how NAMING assists in this…

Yes. This is done by the COMPONENT-LOCAL NAMING CONTEXT. The COMPONENT uses a name in the COMPONENT-LOCAL NAMING CONTEXT to look up the database, and this name can be mapped to a different name in the global NAMING system. So using a different database in the catalog COMPONENT just meant a change in the Deployment Descriptor.

How is a database accessed, then?

Let's take a look at the code:

```
 // get a reference to the naming system
InitialContext iContext = new InitialContext();
 // get a reference to the component local naming context
Context local = (Context)iContext.lookup("java:comp/env");
 // get the DataSource from the JNDI service
DataSource ds = (DataSource)local.lookup("jdbc/shoppingDB");
 // get the Connection from the DataSourceConnection
connection = ds.getConnection();
```

Basically, first the NAMING system is accessed. All parts of the COMPONENT-LOCAL NAMING CONTEXT are placed below the *java:comp/env* context. So it makes sense to store a reference to this context in a variable on its own. Afterwards a concrete name relative to this context, the name *jdbc/shoppingDB* is accessed. So the database must be registered with the name *java:comp/env/jdbc/shoppingDB* for this code to work: *java:comp/env* as the prefix for the COMPONENT-LOCAL NAMING CONTEXT and *jdbc/shoppingDB* as the name of the database relative to that. Afterwards a concrete connection to the database is created.

I still don't really get it… What's the purpose of all this?

The advantage is that the name *java:comp/env/jdbc/shoppingDB* can now be bound to different real database names, so the database that was used by the developer can easily be exchanged for the database we used in our shopping application. Part of this is done in the Deployment Descriptor:

```
<session>
 ...
 <resource-ref>
   <res-ref-name>jdbc/shoppingDB</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
 </resource-ref>
  ...
</session>
```

Here the COMPONENT declares that a connection to the database is expected. However, this doesn't specify which database should be bound to that name – the actual binding is done in an APPLICATION SERVER-specific way.

But you said it was also possible to access other resources in the same way…

…as well as accessing other resources

Yes. The same technique can be used to resolve a COMPONENT's references to other COMPONENTS, the REQUIRED INTERFACES.

So how are these other COMPONENTS accessed?

Again the REQUIRED INTERFACE references are looked up in the COMPONENT-LOCAL NAMING CONTEXT and can therefore be exchanged easily. This simplifies the composition of applications from individual COMPONENTS. Note however that a COMPONENT's reference to another COMPONENT can only be changed to use other type-compatible COMPONENTS. Say we sold one of the COMPONENTS of the shopping application that makes use of the third-party catalog, our customer must also have the same catalog COMPONENT, or a COMPONENT with a compatible COMPONENT INTERFACE.

Can you give an example of how REQUIRED INTERFACES are actually used?

Eric demonstrates REQUIRED INTERFACES

Sure. The *OrderStatus* Bean references the *Order* Bean. Let's have a look at the Deployment Descriptor of *OrderStatus* first. This is the part of the Deployment Descriptor that's important for REQUIRED INTERFACES:

```
<session>
 ...
 <ejb-ref>
  <ejb-ref-name>ejb/Order</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>shoppingcart.OrderHome</home>
  <remote>shoppingcart.Order</remote>
  <ejb-link>Order</ejb-link>
 </ejb-ref>
</session>
```

This part specifies that the required *Order* Bean should be located at *java:comp/env/ejb/Order* in the COMPONENT-LOCAL NAMING CONTEXT. The Remote and Home Interface, as well as the type of Bean, are also defined. Finally it specifies that *java:comp/env/ejb/Order* should point to the name *Order* in the global NAMING.

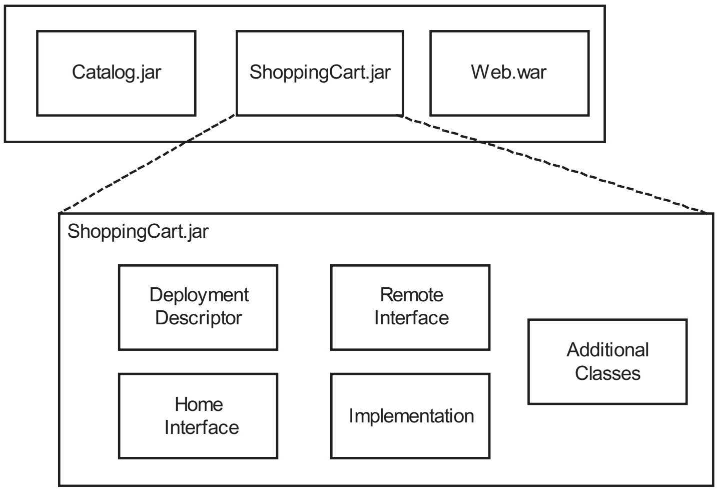
…so java:comp/env/ejb/Order points to the global name Order, but can easily be reconfigured to use a different name.

Exactly.

So that's how I declare that another COMPONENT is used. But how do I actually install this COMPONENT? What does the developer of the original COMPONENT give me?

Eric describes COMPONENT PACKAGES…

COMPONENTS are distributed in COMPONENT PACKAGES, a specific file format. For EJBs, each COMPONENT is distributed as a *.jar* file. 'Jar' is an acronym for 'Java archive'. The layout and contents of a *.jar* file is standardized, it's basically a *.zip* file with added meta-information. So it was also quite easy to get the catalog COMPONENT we bought to work with our APPLICATION SERVER.



What's in a .jar file?

As you can see in the diagram, the file includes the Remote Interface, the Home Interface, the implementation, the Deployment Descriptor and all the other necessary classes. One or many COMPONENTS might be put in a *.jar* file. In our application we decided to have a *.jar* file for each COMPONENT.

Why did you put every COMPONENT in its own file? Does that have any benefits?

First there are some drawbacks: if some COMPONENTS share parts of their ANNOTATIONS, it means that the same ANNOTATIONS must be copied and pasted to several different locations. This makes changes somewhat harder. Also some of the additional classes are used in several different COMPONENTS, again potentially leading to update problems and version conflicts. Alternatively, the common classes could also be kept in a common, additional *.jar* file.

However, *.jar* files make the reuse of COMPONENTS in other applications much easier – you can simply reuse each COMPONENT by using the corresponding *.jar* file. Another advantage of this approach is that you can change and redeploy each COMPONENT separately without any changes to other parts of the application. In our old script-based application this wasn't possible, because lots of functionality was combined in the CGI scripts. For example, the shopping cart and the catalog were handled in a single CGI script, so changes to these two parts had to be synchronized: if two developers had to do changes on these parts, they had to work on the same source file, and therefore they had to cooperate closely to prevent conflicting changes. The component-based approach enabled better and faster changes to the software due to decomposition into parts. We didn't want to compromise this by using *.jar*s with several COMPONENTS inside.

And the application is then assembled from those parts?

Yes. Actually, a collection of COMPONENT PACKAGES packed together for the purpose of forming an application is called an ASSEMBLY PACKAGE. This allows easy deployment of complete applications. The ASSEMBLY PACKAGE also has a Deployment Descriptor. It can be used to 'wire' the individual COMPONENTS together and other stuff, such as defining security roles for the complete application. ASSEMBLY PACKAGES can do more to simplify assembling applications from existing parts, too.

How?

Well, for example we can use uniform security information across the complete application. So in each COMPONENT the same notion of groups or roles can be used. This can only be accomplished because the information is stored centrally in the Deployment Descriptor for the application, so need not be stored in the Deployment Descriptor of each individual COMPONENT.

And that's all put in the CONTAINER, then?

Not exactly. A CONTAINER is responsible for one kind of COMPONENT. For example, there's a CONTAINER for Stateless Session Beans, and one for Entity Beans. But the result of using ASSEMBLY PACKAGES is an application consisting of several different kinds of COMPONENTS. So something has to be defined that can handle such applications. This is the role of the APPLICATION SERVER. An APPLICATION SERVER contains multiple CONTAINERS for the different kinds of COMPONENTS, as well as extra functionalities such as NAMING.

I see. So I have an application, install it and that's it. What happens during this installation process?

...and how COMPONENT INSTALLATION helps

Before I answer that question, let me tell you how COMPONENT INSTALLATION helped at this point in the reimplementation of our shopping application. In the old application, we had to work out a way of installing a new version of the software ourselves. First we did it by copying files on the server and restarting the server by hand. This was quite complicated and sometimes resulted in significant downtime of the server, so we developed scripts that automated the installation of new versions. In the EJB-based system, this procedure was well-defined from the beginning, so almost no additional effort was needed to specify a standard installation procedure. Of course, some parts of the predefined procedure needed to be modified slightly.

This is true even though the installation of the new system is much more complicated. This is because the application now contains multiple different COMPONENTS as well as other parts for the web server, for example. Each of these bits must be installed correctly.

So are the COMPONENTS just copied to the right server – or what else is done?

Several things. The Deployment Descriptors are read and the environment of the COMPONENTS is set up in the way defined there. As I've already said, they contain simple CONFIGURATION PARAMETERS, but also references to MANAGED RESOURCES or references to other COMPONENTS as described in the REQUIRED INTERFACES. So using this information the CONTAINER can set up the environment for the COMPONENTS, most notably the COMPONENT-LOCAL NAMING CONTEXT. The COMPONENTS are made available to the clients as well.

Are clients located on different computers? Like clients of database servers?

Yes. It's a distributed system we're talking about here. Every system where clients talk to remote servers is – by definition – a distributed system. So server-side component systems are also distributed systems.

So the old system was a distributed system because it's a web application?

No, I don't think so. The old shopping application was just a set of CGI scripts running on the web server. It's a distributed system in the sense that its clients – web clients - are remote. But the term *distributed system* here means that the business logic itself is – or at least can be – located on a different machine to the stuff that generates HMTL. So we added a presentation tier in our EJB application. And this presentation tier is separated from the business tier. This is also called a *three-tier architecture*: a database tier, a business logic tier and a presentation tier. All of these tiers can be located on different computers. The CGI application was more monolithic, in the sense that everything, the presentation and the functional stuff were concentrated in the CGI layer. Actually the presentation and the functional stuff run on different machines right now.

Why did you do that? It just means performance penalties because of the relatively slow network, surely?

Eric discusses distributed architectures

There's an advantage to having a distributed system: if you use a remote server, you can easily add another, and you just have to hide which server actually serves a request. So it's the other way round: distribution is a precondition for scalability or fault tolerance, which is a technique to make the system perform adequately even for high loads. That's something that was important to us.

But you can add additional web servers – why should I have a separate server for the business logic?

Because you want to scale them separately. And because it isn't necessarily accessed from the web. Remember, we want to reuse the COMPONENTS also in other applications. So there must a separate level of scalability: a set of business logic servers. And there's another reason for distribution.

Which is?

An important justification for component-based systems is often the need for integration with legacy systems, such as the mailing system in our shopping application. Those naturally run on separate servers, so you have to support a distributed system anyway.

How is distribution accomplished?

The first building block for distribution is the COMPONENT BUS. This is the basic infrastructure for making calls over the available network protocols. This is pretty low-level and was of no big concern to the reimplementation of the shopping application.

Why not?

Because you hardly noticed it. It's concerned with low-level network communication and that's well hidden from the programmer. So you didn't really notice it being around.

I guess it had no advantages then…

Well, the only visible benefit of the COMPONENT BUS as it is implemented in EJB is that the communication layer is the same as the one used for CORBA, the IIOP protocol. This simplifies the integration of EJB and CORBA, because at least RPC calls can be sent from CORBA clients to Beans. The integration of transactions, security and other features is still an issue beyond the COMPONENT BUS, but at least the fundamentals are there. For the shopping application, this means that easy integration of applications in other programming languages is possible as long as the language offers support for CORBA.

As you said, you didn't really deal with this low-level stuff. Why is that?

That's because of the next level above the COMPONENT BUS, the CLIENT-SIDE PROXY. This tries to hide the existence of distribution – something almost every technology for distributed systems tries. But you can't hide distribution from programmers completely. Distributed calls can cause problems than local calls can't, such as network failures that are represented as SYSTEM ERRORS.

What are SYSTEM ERRORS?

A distributed system has a more complex infrastructure with many additional possible errors. You have to deal with them, so they are represented by a separate class of errors that don't occur in 'normal' applications: SYSTEM ERRORS.

Have you got any example code?

Yes. SYSTEM ERRORS are typically represented as *RemoteExceptions* in

EJB. So each call to a method must catch this exception:

```
try {
 ...
 cart.add(new ProductDescription("iPod","apple",1,500F),1);
 ...
} catch (RemoteException ex) {
 ...
}
```

Besides SYSTEM ERRORS*: What else is done in the* CLIENT-SIDE PROXY*?*

Marshaling and unmarshaling of parameters and results are done there. That is, the encoding and decoding of data in a way that can be used for the transfer across the network. Think of it as a way in which complex objects are turned into a stream of binary data. The CLIENT-SIDE PROXY also hides the actual management of the network connections from the user.

Hmm. That should be all there is to know about distribution – we can communicate across a network, and the lurking complexity is hidden from the developer.

Not exactly. The next important concept for distribution is NAMING. We were talking about each COMPONENT having a name already? In fact this is the next step beyond CLIENT-SIDE PROXY. While the CLIENT-SIDE PROXY offers distribution transparency, the benefit of NAMING is location and migration transparency. This means that the COMPONENTS can still be located correctly even if their physical location has changed.

In fact, a COMPONENT's physical location is not even known to the client. At first sight this seems to be no big advantage. However, when we migrated the application to a bigger server the benefits became clear: absolutely no changes were needed to the code. Only the entries in NAMING got updated to reference the new server. These changes were done automatically at deployment. Later we migrated the application to a cluster of servers. Still the code needed no changes! Again the binding of the names were changed transparently during deployment.

Ah. But that's it for distribution, right?

Yes.

Now, what I wonder is, how are all these things put into place? How can a client communicate with a COMPONENT?

Eric describes how the CLIENT LIBRARY and COMPONENT access is managed

Of course, code and other artifacts are needed on the client side. This is a problem, because it might be hard to decide which code is actually necessary on the client side. A CLIENT LIBRARY is used to solve this problem. This contains all the code necessary to access the COMPONENTS on the APPLICATION SERVER and the NAMING system. For example, it includes the COMPONENT INTERFACE, the CLIENT-SIDE PROXY, and code to access the COMPONENT BUS. But in some applications, the CLIENT LIBRARY never needs to be generated.

Why? It appears to be very important…

Because the client is a web application that runs inside the same J2EE APPLICATION SERVER. The CLIENT LIBRARY would only be generated during the deployment for an external client, and it would then be copied to the clients manually. But even in that case the classes might be loaded dynamically from the server. J2EE even defines the Client Container that automatically deals with these issues, so only the APPLICATION ASSEMBLY is needed on the client. As our presentation layer runs on a different machine but in the same J2EE APPLICATION SERVER, the web part of course needed the CLIENT LIBRARY and the server made it available to the server hosting the presentation layer.

John is still unsure about transactions and security in a distributed system…

What I still wonder is: you said that transaction and security can be handled across COMPONENTS. Now if I log in or start a transaction somewhere, this information must be forwarded to the other COMPONENTS. How is this done if everything is distributed? Can't be put in a global variable, can it?

Global variables are evil anyway. This problem is solved as follows: when a remote invocation is executed, an INVOCATION CONTEXT is passed in addition to the normal invocation data, such as method name, target instance, parameters and results. It includes – among other things – transaction information.

…so Eric explains in more detail

First of all, this enables distributed transactions, which are an optional part of J2EE, although it's implemented by most APPLICA-

TION SERVERS. This concept is a prerequisite for having multiple COMPONENTS on different APPLICATION SERVERS that all share one transaction. So in the case in which the system runs on a cluster of servers but uses only one database server, like our shopping application, distributed transactions are necessary. And so the transaction information must be passed between the COMPONENTS using this INVOCATION CONTEXT.

And what about my log-in information?

This is also handled there. The security data in the INVOCATION CONTEXT makes sure that the identity of the caller is available in the APPLICATION SERVER during each method invocation. This means that the APPLICATION SERVER can make permission checks for every invocation. This is only possible because of the INVOCATION CONTEXT that passes this information from the client to the COMPONENTS. Of course we used this feature in the re-implemented shopping application. But the INVOCATION CONTEXT is only interesting because the system is now distributed. In the old shopping system the user information was stored in a cookie and was accessible from anywhere in the code through a user data structure, because all the code ran inside the CGI script on the web server. Because the business logic is now on a separate server, this is not that simple any more.

And now for something completely different… What happens if a client crashes? It would be nice if resources on the server were freed if they aren't used any more, just like garbage collection in normal Java…

Eric discusses distributed garbage collection…

Distributed garbage collection isn't that easy. Something called CLIENT REFERENCE COOPERATION is used there. This means that Stateful Session Beans are destroyed automatically after they've not been used for a specific period of time. In the shopping application we have the example of a user who doesn't do anything with his shopping cart for a time and gets his cart destroyed. This results in reduced resource consumption, but might also lead to problems because users might become frustrated if an unexpected timeout occurs. This could be a reason to replace the Stateful Session Bean with an Entity Bean.

So there's a lot to do just to make distribution easier!

You can come up with different component architectures that don't use distribution. But we're talking about *server* components here. And those *are* distributed.

I did some CORBA and RMI before. These support the same patterns. Are CORBA and RMI themselves already component architectures?

No. It's true that these technologies solve the distribution problems in a similar way. However, they solve *only* these problems and don't implement the rest of the component infrastructure. So at least they can be used as a basis for component architectures according to our definition here. Actually they are used that way: RMI as a basis for EJB and CORBA as a basis for CCM, the CORBA Component Model.

So far you've not shown me how you can use the COMPONENTS*…*

…and how to actually access COMPONENTS *using references…*

To use a COMPONENT, you have to retrieve a reference to it. This is not as trivial as it might seem. I'll go over each step and show you why it's present.

Let's look at the shopping cart again. First the actual code:

```
1   Context ctx = new InitialContext();
2   Object ref = ctx.lookup("ShoppingCart");
3   ShoppingCartHome shoppingCartHome = (ShoppingCartHome)
      PortableRemoteObject.narrow(ref,ShoppingCartHome.class);
4   ShoppingCart cart=shoppingCartHome.create();
5   cart.add(new ProductDescription("iPod","apple"),1);
6   cart.order();
```

The first line creates a reference to the NAMING system, and the second line uses NAMING to locate the COMPONENT HOME for the shopping cart Bean. The third line makes sure that the classes for the GLUE CODE LAYER that was generated during deployment and the COMPONENT PROXY are in place. Then a new individual instance can be created – line 4 – and methods can be called on it in lines 5 and 6. Note that a lot of work is done behind the scenes here, such as distribution or lifecycle handling.

This is a lot more complicated than using a normal class!

Well, if you use a Factory instead of a constructor, your code would be almost identical.

But that's just an explanation for one line. There are many more lines!

First of all, take into account that a lot is happening behind the scenes. Also note that most of the calls are remote calls and you don't notice anything – so you get distribution and therefore fault tolerance and scalability without much code. Basically this code is just the price you have to pay, and it's not that much, is it?

Ok, fair enough. This code shows how a Stateful Session Bean can be used. Are there differences for other COMPONENT *types?*

Yes. The *create()* operation for Stateless Session Beans mustn't take any arguments, because all instances are equal – remember they are stateless. Thus there is no point in passing initialization data to create a *specific* instance using *create()*. The *create()* method of Stateful Session Beans and Entity Beans, however, might take such arguments.

Right, the example above is just a creation. But often you want to access existing COMPONENTS, *I guess.*

… to access COMPONENTS *using* PRIMARY KEYS*…*

Well, for Stateless Session Beans this makes no sense, because all instances are equal, so you just call *create()*. But for Entity Beans you're right. A mechanism must exist to locate a specific logical entity. This is done using the PRIMARY KEY, a technique well-known from the relational database world, where each row has to be identified by a unique value. In our shopping application, the only case in which an Entity Bean was used is the representation of user data. Each user has an integer as its PRIMARY KEY, the user number.

Have you got an example?

Sure – the following code shows how the PRIMARY KEY can be used:

```
UserHome userHome = (UserHome)
  PortableRemoteObject.narrow(ref, UserHome.class);
int userNumber = ...; // get a useful user number
User user=userHome.findByPrimaryKey(new Integer(userNumber));
  // do something with the user
```

Each Entity Bean must have a PRIMARY KEY, used to identify individual entities. Note that you can also use the concept of PRIMARY KEYS to develop Stateless Session Beans that access a database directly, as we've already discussed.

Hmm. But what about Stateful Session Beans? How do I re-locate a Stateful Session Bean?

Oh, I forgot: another way of re-locating COMPONENT instances is by using HANDLES. HANDLES can be used to restore all type of Beans, not just Entity Beans. The code is very simple. Say you have a reference to a shopping cart stored in the *cart* variable. Then the HANDLE can be created as follows:

```
Handle myCartHandle=cart.getHandle();
```

… and using HANDLES

The nice thing about HANDLES is that they can be passed to other parts of a system, or they can be stored on disk, sent by e-mail, whatever. This is because HANDLES are serializable. Later, or in another bit of code, the original instance can be reacquired:

```
ShoppingCart cart=(ShoppingCart) myCartHandle.getEJBObject();
```

In the EJB-based shopping application, we rarely used HANDLES. If we had to refer to an Entity Bean, we passed around the PRIMARY KEY. For Stateful and Stateless Session Beans, usually the remote reference is enough.

By the way, you can also use HANDLES for Home Interfaces. To create a handle for a Home Interface, you can use following code:

```
 HomeHandle homeHandle = shoppingCartHome.getHomeHandle();
```

Later on, the HANDLE can be sent across a network or stored persistently. Then the original Home Interface can be restored:

```
ShoppingCartHome shoppingCartHome =
  (ShoppingCartHome)homeHandle.getEJBHome();
```

This application of HANDLES might be useful, because Home Interfaces remain valid for a long time and are also unique.

Hmm. I think I now even know how to access COMPONENTS. So I know how to develop them, how they are installed and how they can be used. But I'm not certain why should I use them.

Well, I told you about some benefits: they are easy to reuse and they are flexible and easier to maintain, because it is easy to exchange implementations. Also they make development of a distributed system easier.

Eric describes how COMPONENTS *help with performance and fail-over*

Another reason for choosing an APPLICATION SERVER and a component-based approach is performance and scaleability, as I mentioned before. An additional benefit is fail-over: an APPLICATION SERVER is supposed to support high availability, which means that it should be possible to configure the system such that if one APPLICATION SERVER fails, another takes over its responsibilities automatically.

You already mentioned that. Can you give me more details?

Well, in the old application a lot of tricks were used to ensure good performance. For example, caching was used to make sure that the data of currently logged-in users was always held in memory, instead being reread from the database for each access. This was already a minimal form of lifecycle management: the objects weren't deleted after they had been accessed, but were put in the cache for later reuse. So, these objects could be in two states: cached and in-use. The problem was that the implementation of this had to be done manually, and making this work reliably was not a trivial thing to do.

How did this change with EJB?

After we made the transition to EJB, this manual mechanism was no longer needed: the lifecycle is managed by the CONTAINER and we just have to implement the respective LIFECYCLE CALLBACK operations in our COMPONENT IMPLEMENTATION. The lifecycle support has also become much more sophisticated, with different pooling strategies for each Bean type, configurable limits on the number of concurrent object instances, and so on.

So how does the CONTAINER *actually help here?*

To achieve the caching and pooling and make it transparent to the user, the CONTAINER uses the concept of a VIRTUAL INSTANCE. This means that a client's reference to a Bean does not reference the Bean instance itself – it's rather just a reference to some CONTAINER-provided proxy, and the Bean instance is prepared only when it is really accessed. For example, in the case of an Entity Bean, its state could be loaded from a database just before it is accessed.

Why does that help?

This concept is important to enable the implementation of several other patterns. For example, INSTANCE POOLING: Entity Beans need fewer physical instances than there are logical entities. Every time a specific logical entity, for example a certain user, is accessed, the user data is loaded into memory, into a pooled physical COMPONENT instance, just in time. So a physical instance of a user COMPONENT represents several different logical instances in its lifetime. For us, this again replaced some of the sophisticated caching techniques we had to implement manually in the old implementation.

And what about other Bean types? Are there optimizations for Stateless Session Beans, for example?

Things are different for Stateless Session Beans – the consequences of INSTANCE POOLING are not obvious for this Bean type. In the old application, some parts such as the order status were required to be thread-safe because they were accessed from different threads concurrently, each of them serving a different customer. We decided to implement these parts as Stateless Session Beans. These need not be thread-safe, because instead several instances are created, one for each thread. Development was simplified significantly, because we didn't need to consider thread-safety during the implementation. To reduce the number of instances at run time, we used INSTANCE POOLING as an optimization: a limited number of physical instances are given to the different users.

And Stateful Session Beans?

The optimization for Stateful Session Beans is called PASSIVATION. In the old application, the state of all the shopping carts was held in memory, all the time. Now that they are implemented as Stateful Session Beans, they are PASSIVATED automatically: their state is written to disk if an instance isn't used for a while. This results in a more scalable application, because it consumes less memory due to the PASSIVATION of rarely-used Stateful Session Bean instances. Again, we only had to implement the LIFECYCLE CALLBACKS – the rest is done by the APPLICATION SERVER.

And how are these LIFECYCLE CALLBACKS called?

They're different for each Bean type, and the implementation classes are all different too.

But, how can the CONTAINER handle all these different COMPONENTS efficiently? I mean, each has other operations in their interfaces, has other security configurations, different requirements for transactions, and so on?

Eric describes how the CONTAINER adapts to different Bean types

The CONTAINER uses a GLUE CODE LAYER to provide an adaptation from the rather generic CONTAINER implementation to each specific Bean. This layer implements the COMPONENT INTERFACE and uses the COMPONENT IMPLEMENTATION for business logic. The important point is that the GLUE CODE LAYER calls the LIFECYCLE CALLBACKS at the appropriate times. It also provides the transaction and security handling required by the ANNOTATIONS. VIRTUAL INSTANCE – by PASSIVATION or INSTANCE POOLING – is also implemented as part of this layer.

But you can't do much about other resources apart from memory, say database connections for example…

Oh yes you can! For example, in a further performance optimization the CONTAINER provides MANAGED RESOURCES, such as database connections. During the implementation of the new EJB shopping application, we didn't really see any consequence of MANAGED RESOURCES. We were still doing our usual database handling, but the CONTAINER automatically provided connection pooling and sophisticated transaction management. As COMPONENT developers we didn't bother about these issues – but performance has increased significantly. In the old application we had to take care of this stuff ourselves and, as it turns out, in some cases we didn't implement it all that well, so the performance wasn't too good. MANAGED RESOURCES are also needed to implement PASSIVATION or INSTANCE POOLING. While an instance is passivated, its resources should be given back to the CONTAINER. This is easier when MANAGED RESOURCES are used, because connections needn't be closed each time.

So a COMPONENT*-based application is much faster with all those performance optimizations around…*

(*Laughs*) Actually, that's how things should be. But in reality, well…

Does that mean a COMPONENT*-based application is slower?*

Sometimes it is. There's an overhead involved for each remote invocation, and all the 'server magic' doesn't come free. But on the other hand, there are other advantages of COMPONENT-based systems such as reuse, additional flexibility and so on. And especially fault tolerance and higher availability, these were major reasons to use EJB.

So you have to pay in performance for the other advantages?

Hmm. Let me ask you a question: what do you actually mean by performance?

Well, for a shopping application, there are issues such as response time or maximum concurrent users. I think that's my definition.

Do you expect the same performance from a desktop PC and an SMP server?

Of course not!

So, you have to take hardware into account when talking about performance. The easiest thing to change in a system, usually, isn't the software, but the hardware. Adding another computer to a set of servers is much easier than rewriting software, and often less expensive.

So the hardware is important, sure. But with infinite hardware resources, every application can be as fast as needed. In other words, if you throw more computer power at the problem, every application can be made arbitrarily fast.

Yes and no. This would be true if you could have one arbitrarily fast computer. But in fact more computer power most often means more CPUs or more computers, not one faster computer.

I still don't see the point.

The point is: performance isn't everything. It's probably more important that your architecture allows you to add more CPUs or more computers if needed and benefit from that. This is called *scalability*. And this is where COMPONENT systems have their real advantages if used correctly. APPLICATION SERVERS are usually very scalable, they can use the additional computing power efficiently.

It's a business matter: what is cheaper? Making the application faster or buying another computer? The latter option is always available with a well-designed component-based application.

And that's all free. You just use an EJB server and your system can get as fast as you like!

Not quite. It's still a distributed system. You can create a lot of problems – you have to be careful. Remember that we even had our design influenced by performance considerations: the decision to make the order a Stateless Session Bean and not an Entity Bean was a performance decision. Putting workflows like *OrderStatus* or *OrderProcess* on the server was also a decision at least influenced by performance. But a lot of basic prerequisites for scalability and performance are built into EJB, while they would have to be programmed explicitly in CORBA or RMI. But even with EJB, if you do a bad design, the application might even become slower if you add more servers, because the network traffic increases. But besides better scalability, you also get fault tolerance and better availability, because the system is built of an array of servers that can replace each other in case of failure.

So did you find implementing the system using COMPONENTS better?

Well, there was just no way we could get the CGI scripts to offer the same kind of scalability and fault tolerance as easily as with the component-based design. And buying an APPLICATION SERVER, as opposed to implementing all the technical stuff manually, was also an important argument.

So there was no real choice, right?

Yes, that's the point. Using a scripting language gives you quick results at first, but there are also limits to these technological foundations. That's why we changed.

Do you think it was more complicated?

It was – let's say – different. Distributed, component-based systems are designed differently from quickly-hacked scripts of course, but it's also different from good object-oriented design. So we had a lot to learn in this respect. The EJB environment gives you a lot of basic functionality to achieve good scalability and good design, but you still have to learn how to use it efficiently.

I see. So – would you do it the same way again?

I would use the same technologies, but with the experience I gained I would design an even better system, I believe. Or should I say I hope?

Literature and Online Resources

[ACM01] Deepak Alur, John Crupi, Dan Malks: *Core J2EE Patterns*, Sun Microsystems Press, Prentice Hall, 2001

This book presents patterns on J2EE application design. It gives a very good 'kick-start' to J2EE application design and is currently a 'must read' for J2EE developers and architects.

[AL77] Christopher Alexander, Sara Ishikawa, Murray Silverstein et al.: *A Pattern Language – Towns • Buildings • Construction*, Oxford University Press, New York, 1977

No pattern book can pass without a reference to Alexander's books. This book contains a 253-pattern language describing how architectural patterns can create spaces in which people can live and work in harmony.

[AL79] Christopher Alexander: *The Timeless Way of Building*, Oxford University Press, New York, 1979

The introductory volume in the Center for Environmental Structure Series, which also includes [AL77]. Presents a new theory of architecture, building, and planning that lead to the original ideas for software patterns.

[AOP] The Aspect Team: *The Aspect-Oriented Programming Homepage*
http://aosd.net/

Here you will find all relevant information on Aspect-Oriented Programming. The collection of links to all kinds of technologies using aspect-oriented techniques is especially useful.

[BCK98] Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice*, Addison-Wesley, 1998

This book illustrates basic concepts of software architecture. It starts from the goals you want to reach with an architecture, explains the basic building blocks, and provides a collection of useful architectural styles (or templates) that you can use in practice. A collection of interesting case studies completes the book.

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern-Oriented Software Architecture, A System of Patterns*, John Wiley & Sons, 1996

The perfect companion to [GHJV94]. Every software architect and software developer has to read this book.

[BR00] Alan W. Brown: *Large-Scale Component-Based Development*, Prentice Hall, 2000

An introduction to Component Based Development.

[CCG00] James Carey, Brent Carlson, Tim Graser: *SanFrancisco Design Patterns*, Addison-Wesley, 2000

IBM's SanFrancisco is one of the largest software projects based on Java. But it's also one of the largest component projects, so a lot can be learned from it. This book presents design patterns from the SanFrancisco project.

[CD01] John Cheesman, John Daniels: *UML Components*, Addison-Wesley, 2001

Component-Based Development needs a process. This book describes a simple process for CBD projects. It uses UML to express specifications for components. A lightweight point to start from if you seek a CBD process.

[CE00] Krzysztof Czarnecki, Ulrich Eisenecker: *Generative Programming*, Addison-Wesley 2000

[CETUS] Schneider, et al.: *Cetus-Links*
http://www.cetus-links.org

A very useful catalog of software development URLs.

[CN91] Brad J. Cox, Andrew J. Novobilski: *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley, 1991

Many of the ideas of component based development are not really new. Does anybody remember the Software-IC? Here you can find explanations of some of the roots of CBD.

[DSW99] Desmond F. D'Souza, Alan Cameron Wills: *Objects, Components and Frameworks with UML*, Addison-Wesley, 1999

This book introduces the Catalysis approach and is an important work for Component Based Development.

[ED00] W. Keith Edwards: *Core Jini*, Prentice Hall, 2000

http://www.kedwards.com/jini/corejini.html

You need an alternative to EJB? JMS didn't do it? How about Jini?

[FJ00] Mohamed E. Fayad, Ralph E. Johnson: *Domain-Specific Application Frameworks*, John Wiley & Sons, 2000

Part of the monumental framework trilogy made up by [FSJ99b] and [FSJ99].

[FO96] Martin Fowler: *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997

A book about patterns that focus on reusable object models.

[FO99] Martin Fowler: *Refactoring, Improving the Design of Existing Code*, Addison-Wesley, 1999

A book totally unrelated to ours, but nevertheless one of the most important programming books ever written. It shows how quality code should look and how to achieve it.

[FS98] Martin Fowler, Kendall Scott: *UML Distilled*, Addison-Wesley, 2000

A quick introduction to UML, containing everything you need to understand the figures in this book, and of course much more.

[FSJ99b] Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson: *Building Application Frameworks*, John Wiley & Sons, 1999

A book that shows how to build object-oriented frameworks.

[FSJ99] Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson: *Implementing Application Frameworks*, John Wiley & Sons, 1999

Forty case studies of different frameworks.

[FV00] Astrid Fricke, Markus Voelter: *SEMINARS – A Pedagogical Pattern Language on how to Teach Seminars Efficiently*
http://www.voelter.de/publications/seminars.html

This URL describes a pattern language intended for instructors in industry who are not trained educators. We wrote it especially for those who feel that something is going wrong with their seminars, who are perhaps frustrated and who don't know what's going wrong or what to change. This pattern language gives some hints about how the situation can be improved, i.e. how to run better seminars. In addition to describing the patterns themselves, the pattern language also tries to explain some of the biological, anthropological and pedagogical background, covering all aspects of a seminar from preparation to exams.

[GE99] David Hillel Gelernter: *Machine Beauty: Elegance and the Heart of Technology,* Basic Books, 1999

Gelernter is considered one of the masterminds of computer science in general. In this book he argues that there is more to engineering than just creating solutions. Computer scientists should also learn design, art and music and take these into account, something often forgotten in day-to-day work.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

The 'Gang of Four' book, probably the most famous pattern book. What else can we say about this, it rocks…

[GO00] Alan Gordon: *The COM and COM+ Programming Primer*, Prentice Hall, 2000

This book provides an easily-understandable introduction to the concepts and techniques used in Microsoft's COM and COM+ technologies. It does not provide too much detail and requires no previous knowledge of the Microsoft programming world.

[HA00] Eric Hall: *Internet Core Protocols: The Definitive Guide*, O'Reilly & Associates, 2000

RMI is not deep enough for you? This book digs deeply into the TCP/IP layers.

[HDSC] IBM Corporation: *Multi-Dimensional Separation of Concerns with Hyperspaces*
http://www.research.ibm.com/hyperspace/

Hyperspaces constitute IBM's approach to multidimensional separation of concerns. The respective implementation for Java is called Hyper/J and is available from the above web site.

[HILL] James O. Coplien: *A Pattern Definition*
http://hillside.net/patterns/definition.html

This web page contains some of the classic definition of software patterns.

[HS00] Peter Herzum, Oliver Sims: *Business Component Factory*, John Wiley & Sons, 2000

This book provides a comprehensive overview of component-based enterprise business systems. It starts out from the business case for components, and touches on technical details about their implementation. Analysis, application design and process issues are also covered.

[HV99] Michi Henning, Stephen Vinoski: *Advanced CORBA Programming with C++*, Addison-Wesley, 1999

This is *the* book on CORBA programming with C++. It provides a good introduction, but does not stop there, delving well into the technical details. The authors are well-respected experts in their field.

[ISE] ISE: *An Introduction to Design by Contract*

http://www.eiffel.com/doc/manuals/technology/contract/page.html

Bertrand Meyer introduced the idea of design by contract with the programming language Eiffel, and here the basics of this concept are explained.

[JBOSS] *The JBoss Homepage*
http://www.jboss.org

News, documentation, downloads and discussion on this Open Source Application Server.

[JINIPL] Michael Kircher, Prashant Jain: *The Jini Pattern Language*
http://www.cs.wustl.edu/~mk1/AdHocNetworking/

This pattern language constitutes a collection of patterns that apply in ad-hoc networking environments such as Jini or UPnP.

[KE00] Nicholas Kassem, The Enterprise Team: *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition*, Addison-Wesley, 2000

The famous J2EE blueprints introduce best practices for J2EE development. The 'pet store' application is introduced in this book.

[KJ00] Michael Kircher, Prashant Jain: *Lookup Pattern*, EuroPLoP 2000 Conference, Irsee, Germany

The *Lookup* design pattern originated in software architecture, in the field of distributed object computing, and describes how to locate and recover references to distributed objects and services. Services register references to themselves with a defined access point, and are then retrieved by clients wishing to access the services.

[LD97] Chris Loosley, Frank Douglas: *High Performance Client/Server,* John Wiley & Sons, 1997

A discussion about the performance aspects of middleware, databases, transaction monitors and high performance in distributed systems in general.

[LE00] Doug Lea: *Concurrent Programming in Java, Design Principles and Patterns* (Second Edition), Addison-Wesley, 2000

If you do concurrent programming in Java and you still haven't read this book, stop now, go to the nearest book store and buy the book! An exceptional book.

[LG00] Craig Larman, Rhett Guthrie: *Java 2 Performance and Idiom Guide*, Prentice Hall, 2000

If you have you ever wondered why most Java applications are slow, this is a practical text on Java performance and how to optimize it.

[MA00] Klaus Marquardt: *Patterns for Plug-Ins*, Proceeding of the EuroPLoP '99 Conference

This pattern collection helps to define, implement and package plug-ins specific to a configurable application. Central patterns are the *Plug-In* and the *Plug-In Contract* between the *Plug-In* and the application: patterns for packaging and registration of plug-ins are also explored. Process and organization patterns complement the general technical patterns, and support the technical flexibility introduced with plug-ins. The patterns in the last section focus on implementation techniques and show how other design patterns can be used for plug-ins.

[MH00] Richard Monson-Haefel: *Enterprise JavaBeans* (Second Edition), O'Reilly & Associates, 2000

The classic EJB beginner's book. Introduces EJB and its API and surely deserves its position as one of the most important EJB books published. A new version has been published recently covering EJB 2.0.

[MHC01] Richard Monson-Haefel, David Chappell: *Java Message Service*, O'Reilly & Associates, 2001

Monson-Haefel did it again, this time for JMS. Another classic written together with David Chappell, leader of Progress Software's engineering team. Explains all you need to start with JMS.

[ML01] Brett McLaughlin: *Java and XML* (Second Edition), O'Reilly & Associates, September 2001

Ever wondered what this DTD thing within the Deployment Descriptors means? This book gives you a quick introduction to XML and shows how to use it in Java.

[MM97] Thomas J. Mowbray, Raphael C. Malveau: *CORBA Design Patterns*, John Wiley & Sons, 1997

A book on design patterns for CORBA.

[MSCOM+] Microsoft Corporation: *COM+ specification*
Available from http://www.microsoft.com

[MSDCOM] Microsoft Corporation: *DCOM specification*
Available from http://www.microsoft.com

[MV02] Markus Voelter: *Small Components*

http://www.voelter.de/smallComponents

[NO97] James Noble: *Basic Relationship Patterns*
http://www.riehle.org/europlop-1997/p11final.pdf

A paper presenting the basic patterns that describe how objects can be used to model relationships within programs. Relationships between objects are almost as important to design as the objects themselves, and most programming languages do not support relationships well, forcing programmers to implement relationships in terms of more primitive constructs.

[ÖB01] Rickard Öberg: *Mastering RMI: Developing Enterprise Applications in Java and EJB*, John Wiley & Sons, 2001

Written by one of the JBoss architects, this book explains everything there is to know about RMI.

[OH98] Robert Orfali, Dan Harkey: *Client/Server Programming with Java and CORBA* (Second Edition), John Wiley & Sons, 1998

Another classic, this time on Java and CORBA. A good beginner's-level introduction to using CORBA with Java and the vision of a better world – a world with distributed objects everywhere.

[OMGCCM] The Object Management Group: *CCM specification* available from http://www.omg.org

[OMGCORBA] The Object Management Group: *OMG's CORBA Homepage* http://www.corba.org

[OMGOTS] The Object Management Group: *The catalog of OMG specifications* (including the OTS specs) http://www.omg.org/technology/documents/spec_catalog.htm

[OPENCCM] *The OpenCCM Platform* http://corbaweb.lifl.fr/OpenCCM/

OpenCCM stands for the Open CORBA Component Model Platform: a publicly available Open Source implementation of the Object Management Group's CORBA Component Model.

[OPENEJB] *The OpenEJB homepage* http://openejb.exolab.org/

OpenEJB is another Open Source EJB implementation.

[PO01] Shelley Powers: *Developing ASP Components* (Second Edition), O'Reilly & Associates, 2001

A practical introduction to component usage with ASP showing component-oriented programming with Visual Basic, Visual C++ and Visual J++ in the Microsoft environments.

[PPP] The Pedagogical Patterns Project http://www.pedagogicalpatterns.org

The PPP collects and re-factors patterns that cover aspects of teaching and learning.

[PRGNR99] Marco Pistoia, Duane F. Reller, Deepak Gupta, Milind Nagnur, Ashok K. Ramani: *Java 2 Network Security (*Second Edition), Prentice Hall PTR, 1999

Explains many aspects of the Java 2 (and earlier version) security concepts. As security is a very important aspect of distributed systems, Java's security lays the foundation for secure EJB applications.

[RI98] Dirk Riehle: *Bureaucracy in Pattern Languages of Program Design 3*, Edited by Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998.

The *Bureaucracy* pattern is used to implement hierarchical object structures that are capable of maintaining their own inner consistency. It is a composite pattern based on the *Composite*, *Observer* and *Chain of Responsibility* patterns.

[SI00] Jon Siegel, *CORBA 3 Fundamentals and Programming*, John Wiley & Sons, 2000

A comprehensive introduction to CORBA, also covering the advanced concepts of CORBA 3 and the CORBA Component Model in particular.

[SN95] Snoopy: *Ansichten eines Beagles*, Bernd-Michael Paschke Verlag, 1995

Ever felt this heavy *deja-vu* thing as a developer? This is a collection of articles from the *GUUG Nachrichten*, the German Unix User Group's magazine.

[SO99] Peter Sommerlad: *Configurability*, Proceedings of the EuroPLoP '99 Conference, Irsee, Germany

Configurability is a collection of patterns on data-driven programming, consisting of *Configurable System*, *Configurable Component System*, *Bootstrapper* and *Named Configurable Object*.

[SR98] Peter Sommerlad, Marcel Rüedi: *Property List*
http://www-poleia.lip6.fr/~razavi/aom/papers/oopsla98/sommerlad.pdf

The *Property List* design pattern is used to attach attributes to an object at run time. Each attribute is given a name represented by a data value, as opposed to an identifier in the programming language, and attributes can be added or removed on a per-object basis.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*, John Wiley & Sons, 2000

Again a 'must read'. This patterns book deals with networking and concurrency. It covers the technical foundations necessary for the material in our book.

[SU01] Mark Wilcox, Alex Toussaint, Sameer Tyagi, Subrahmanyam Allamaraju et al.: *Professional Java Server Programming* (Edition 1.3), Wrox Press, 2001

A 1600+-page monster covering the whole of the J2EE, with lots of information and many source code examples. A good place to start if you need a quick introduction to a part of the J2EE or if you need sample code.

[SUNEJB] Sun Microsystems: *EJB Specification, Reference Implementation*
http://java.sun.com/products/ejb

The specification, reference implementation and everything else you might need for EJB directly from Sun.

[SUNJ2EE] Sun Microsystems: *Java 2 Enterprise Edition*
http://java.sun.com/j2ee/
Sun's J2EE home page.

[SUNRI] Sun Microsystems: *J2EE Reference Implementation*
http://java.sun.com/j2ee/download.html

[SUNRMI] Sun Microsystems: *RMI Homepage*
http://java.sun.com/products/jdk/rmi/index.html

[SZ99] C. Szyperski: *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1999
The classic book about components. It introduces the fundamentals of components like reuse and compares Sun's JavaBeans with Microsoft's DCOM and ActiveX and with OMG's CORBA components.

[VL98] John Vlissides: *Pattern Hatching – Design Patterns Applied*, Addison-Wesley, 1998
Have you ever wondered how the GoF book [GHJV94] was written, how to hatch patterns and what it's all about? This book gives you all the answers, and, even better, it's great fun to read.

[VSW02] Markus Voelter, Alexander Schmid, Eberhard Wolff: *Server Component Patterns – Book homepage*
http://www.servercomponentpatterns.org
This URL contains errata and additional patterns to complement this book, together with the part of the book we did not have time to include by the publication date – the application patterns.

[WE85] Gerald M. Weinberg: *The Secrets of Consulting*, Dorset House Publishing, 1985
If you are a consultant and sometimes think life is strange, keep calm – not only yours is strange. Learn more about consulting from one of the most experienced consultants, Gerald Weinberg. An entertaining book that teaches everything there is to know about consulting.

[YO97] Edward Yourdon: *Death March – The Complete Software Developer's Guide to Surviving 'Mission Impossible' Projects*, Prentice Hall, 1997
Not a pattern book, not a component book, not even an EJB book. But if you're part of a 'death march' project, you will be glad to see this.

Glossary

API

Application Programmer Interface, a collection of entry points that allow programmatic control of an application or other software system.

Business Object

A Business Object is an object with a meaning in the business context, for example a customer. It is motivated rather by the business than the technical domain. The goal is to let the developer focus on the business domain rather than on technical issues. Business Objects can be seen as a root of COMPONENTS.

CCM

CCM (CORBA Component Model) is a COMPONENT system influenced by ➢EJB but based on ➢CORBA. It therefore provides the same general architecture as EJB, but with the programming language independence of CORBA, as well as other modifications.

Code Generation

Generally every kind of compile process can be viewed as code generation. This term means that certain standard code that differs only slightly for different purposes is generated, instead of being typed in. This might cover technical details. For example, you could develop a COMPONENT in a technology-independent language that is translated to EJB or CCM by a code generator

COM+

COM+ is the COMPONENT technology developed and marketed by Microsoft. It is the successor of COM and DCOM, incorporates the

Microsoft Transaction Server (MTS) and is available on every Microsoft Windows platform.

CORBA

CORBA (Common Object Request Broker Architecture) is a platform- and language-independent standard for ➢Distributed Objects. Using CORBA, therefore, allows objects on different computers written in different languages to communicate. CORBA also defines facilities and services to support the development of ➢Distributed Object Systems and CORBA Domain Interfaces as standards for ➢Business Objects in certain domains.

Distributed Objects

Distributed Objects are the next step after ➢RPC, taking the paradigm of object-orientation to the network. That is, objects that are usually only accessible in an application might also be accessed from other computers through the network.

See *Foundations* for more details.

DLL

Dynamic Link Library. While an executable program can be statically linked – i.e. all parts are in one file – it is possible to use dynamic linking as well. In this case some functions are located in a different file, a library. This file then need only be loaded once even if several programs use functions from the library.

In Windows these libraries are called DLLs (Dynamic Link Libraries).

EJB

EJB (Enterprise JavaBeans) is a ➢Java-based framework for COMPONENTS. It includes services for transactions, security and persistence. As a part of ➢J2EE it is responsible for hosting business logic. It makes use of ➢RMI for the implementation of distributed calls.

Framework

A framework is a collection of classes. The main idea is that only some parts of the framework need to be inherited from in order to build an application or make use of the framework. These are called 'extension points'. There are technical frameworks for building graphical user interfaces, for example, and there are domain frameworks for building applications for specific business domains.

Functional Variability

A ➢Principle that describes the goal that each part of a software application should be easily changeable, possibly by configuration.

Garbage Collection

While in some systems memory and resources must be freed explicitly, garbage collection is an alternative. Garbage collection determines which parts of memory and which resources will no longer be accessed because no references are held to it, allowing memory and resources to be freed. Garbage collection is implemented in ➢Java. Garbage collection is hard to implement in distributed systems, because some of the host computers might fail.

IIOP

Internet Inter-ORB Protocol, the protocol standardized as part of ➢CORBA, also used for ➢RMI and ➢EJB. IIOP is an implementation of GIOP (General Inter-ORB Protocol) based on TCP/IP as network protocol. GIOP itself is independent of the underlying network protocol.

Java

Java is a programming language in the tradition of C and C++. It includes ➢Garbage Collection and is compiled into an intermediate code ('Byte Code') that is interpreted at run-time.

J2EE

Java 2 Enterprise Edition, an edition of ➢Java for enterprise business applications. J2EE includes ➢EJB for business logic, Servlets and Java

Server Pages for the web front-end and several other ≻APIs for transactions (JTA, the Java Transaction API), database access (JDBC, Java Database Connectivity) and ≻MoM (JMS, Java Message Service).

MoM

Message Oriented Middleware. As opposed to ≻RPC or ≻Distributed Objects, MoM tries to make an access to a remote system explicit instead of transparent. In an MoM architecture it is therefore necessary to create a message, put information in it and send it to the receiver. Message delivery is usually decoupled from message sending, so calls in MoMs are usually asynchronous. Queuing and guaranteed delivery is easier to implement in an MoM system than in a tightly-coupled system based on a RPC mechanism, and many major software systems rely on this technology.

Multitier Architecture

This term describes the usual approach to designing distributed systems. Usually a distributed system is layered in three tiers: the presentation, the business logic and the database tier. Further sublayers might be used, for example the business logic tier might use layers for work flow and for ≻Business Objects.

Pattern

A pattern is a three-part rule that expresses a relationship between a specific context, a specific system of forces that occur repeatedly in that context, and a specific configuration that allows these forces to resolve themselves [HILL].

See *Foundations* for a more complete discussion.

Pattern Language

A collection of ≻Patterns with a language-wide goal. Applying the patterns in a defined sequence generates a solution to the general problem the patterns are designed to address.

See *Foundations* for a more complete discussion.

Principle

A guideline or high-level goal for a software architecture. For example, the Principle might be the goal that ≻Patterns help to reach.

RMI

Remote Method Invocation, a system for ≻Distributed Objects based on Java. RMI includes naming and distributed ≻Garbage Collection.

RPC

Remote Procedure Call, a technology for sending a procedure call from one computer to another. Technologies that implement RPCs are Sun RPC and DCE (Distributed Computing Environment). The successor to RPCs are ≻Distributed Objects.

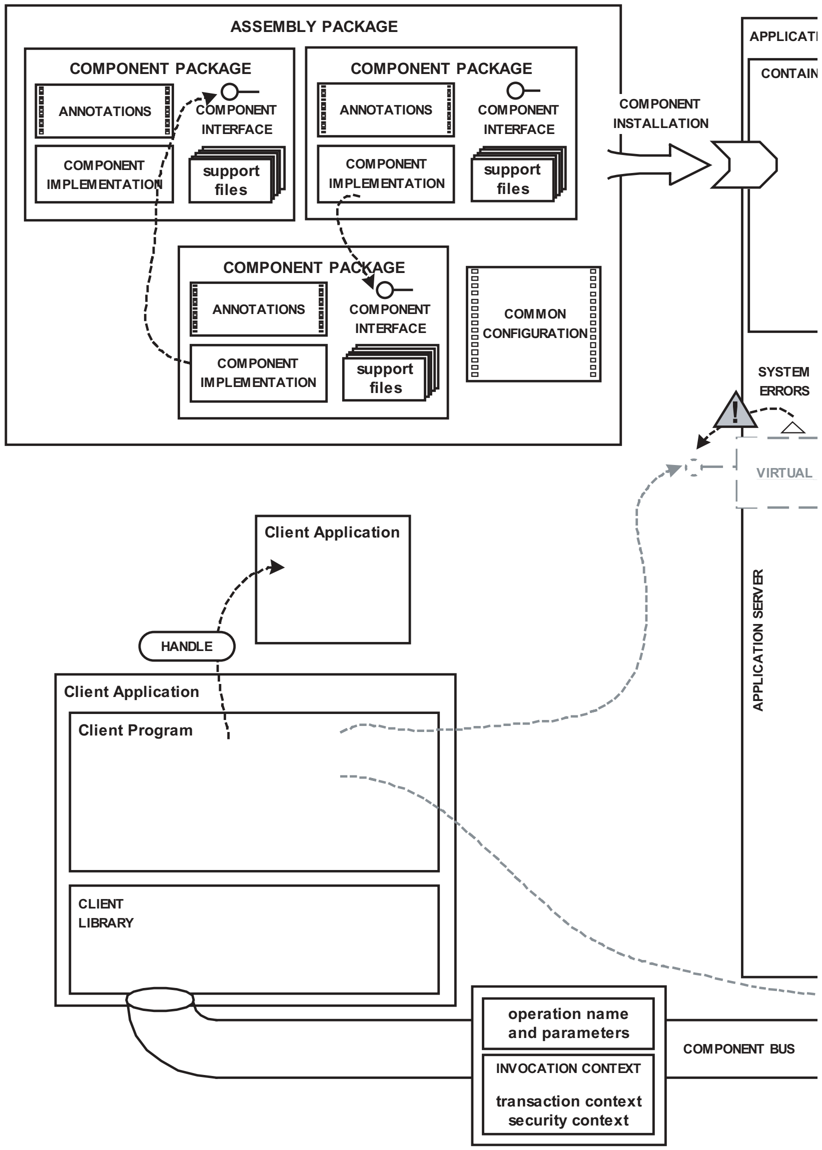
Separation of Concerns

One of the ≻Principles of COMPONENT infrastructures. Through the separation of technical and functional concerns, a developer of a business application can focus on the business logic, while the system developer can focus on basic services for transactions and so on.

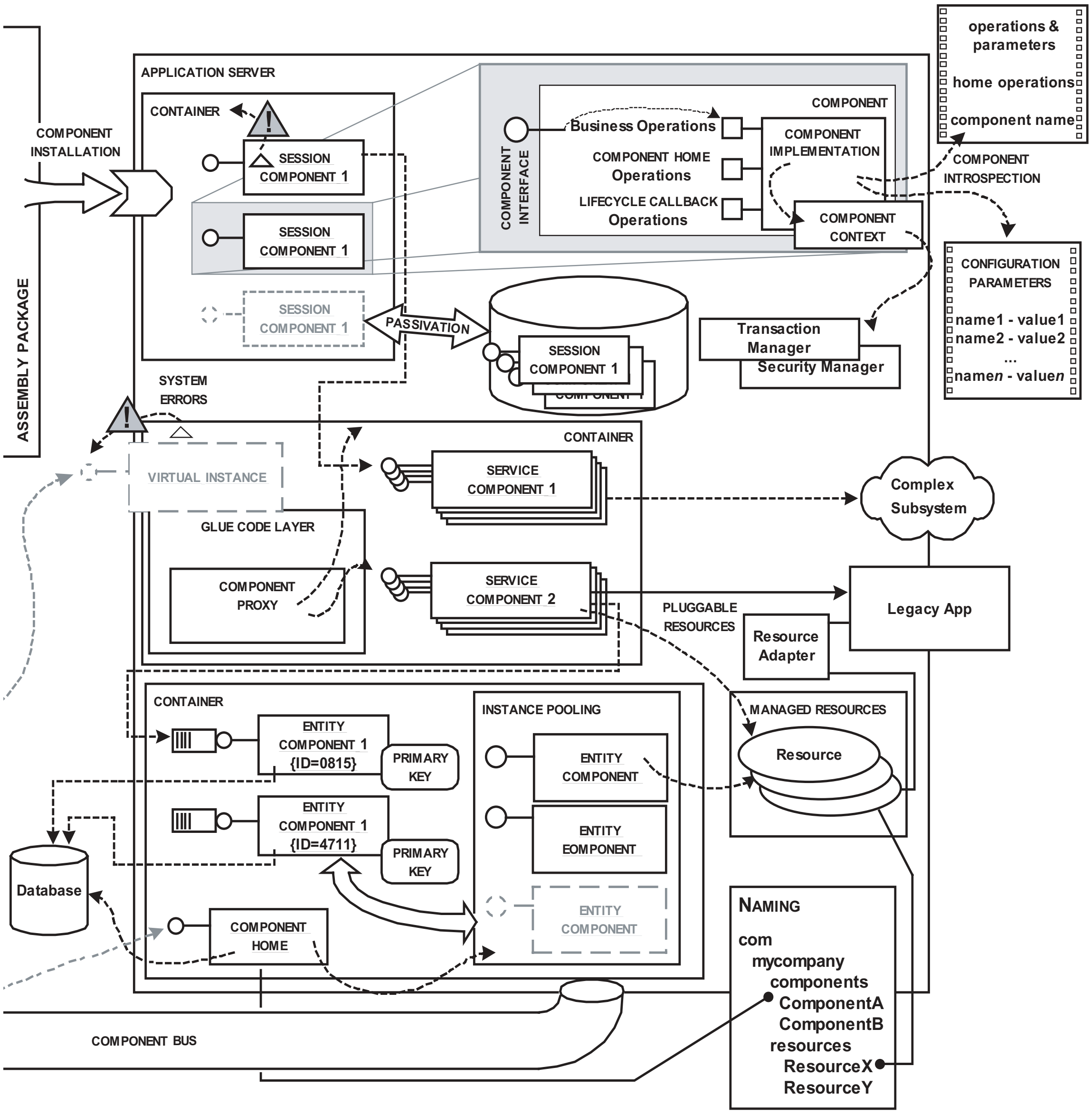
Thread

A Thread is a 'lightweight' process – a process without its own memory and executable. Multiple threads can therefore use the same code and data without the substantial overhead interprocess communication (IPC) involves.

Index



ASSEMBLY PACKAGE
COMPONENT PACKAGE
ANNOTATIONS
COMPONENT INTERFACE
COMPONENT IMPLEMENTATION
support files
COMPONENT PACKAGE
ANNOTATIONS
COMPONENT INTERFACE
COMPONENT IMPLEMENTATION
support files
COMPONENT PACKAGE
ANNOTATIONS
COMPONENT INTERFACE
COMPONENT IMPLEMENTATION
support files
COMMON CONFIGURATION
COMPONENT INSTALLATION
APPLICATI
CONTAIN
SYSTEM ERRORS
VIRTUAL
APPLICATION SERVER
Client Application
HANDLE
Client Application
Client Program
CLIENT LIBRARY
operation name and parameters
INVOCATION CONTEXT
transaction context
security context
COMPONENT BUS



ASSEMBLY PACKAGE
COMPONENT INSTALLATION
APPLICATION SERVER
CONTAINER
SESSION COMPONENT 1
SESSION COMPONENT 1
SESSION COMPONENT 1
COMPONENT INTERFACE
COMPONENT
Business Operations
COMPONENT HOME Operations
LIFECYCLE CALLBACK Operations
COMPONENT IMPLEMENTATION
COMPONENT CONTEXT
operations & parameters
home operations
component name
COMPONENT INTROSPECTION
CONFIGURATION PARAMETERS
name1 - value1
name2 - value2
...
namen - valuen
PASSIVATION
SESSION COMPONENT 1
Transaction Manager
Security Manager
SYSTEM ERRORS
VIRTUAL INSTANCE
CONTAINER
GLUE CODE LAYER
COMPONENT PROXY
SERVICE COMPONENT 1
SERVICE COMPONENT 2
Complex Subsystem
Legacy App
PLUGGABLE RESOURCES
Resource Adapter
MANAGED RESOURCES
Resource
CONTAINER
ENTITY COMPONENT 1 {ID=0815}
PRIMARY KEY
ENTITY COMPONENT 1 {ID=4711}
PRIMARY KEY
INSTANCE POOLING
ENTITY COMPONENT
ENTITY EOMPONENT
ENTITY COMPONENT
Database
COMPONENT HOME
COMPONENT BUS
NAMING
com
mycompany
components
ComponentA
ComponentB
resources
ResourceX
ResourceY